Sugar for OCL

Jörn Guy Süß

Information Technology and Electrical Engineering
The University of Queensland, St. Lucia, 4072, Australia
jgsuess@itee.uq.edu.au

Abstract. Examples of OCL use often do not exceed a few lines. Larger examples are rare, because the concrete syntax of OCL is verbose and based exclusively on ASCII encoding. This makes it easy to edit OCL in any environment, but hard to layout in a readable manner. A minor issue like presentation affects use in a major way. This paper proposes three shorthand notations, or syntactic sugars, for laying out OCL in the Latex, HTML, and Unicode encoding systems. To avoid splitting the available OCL source code base any further, flavours are convertible via the base syntax. To allow benefit across the community, the representations are OCL version-independent. To support recognisability, the representations are visually very similar. To simplify reuse, definitions are based on POSIX regular expressions and Unicode.

1 Introduction

While OCL today offers an substantial number of tools, its adoption as an industry standard is still limited. Usage issues due to language semantics and tooling, like the lack of modularisation, non-deterministic evaluation, the missing and insufficiently formalized transitive closure operation and collection flattening have been addressed successfully in the past[2,**?**] and seem to consolidate. OCL's concrete syntax still seems to be an impediment:

Working with a UN-CEFACT group on the metamodel for the business language UMM[8], which is formalized as a UML profile, I had to introduce the workgroup to the use of OCL. It turned out that even simple examples quickly filled the whiteboard. Points of emphasis were hard to make, due to the expansive syntax. Thus I resorted to short symbols, borrowed from math, logic or improvised in the process, but made clear that these were *not* standard compliant. After collecting the workshop notes, I found that most participants had adopted the shortened ad-hoc syntax and that those that had used it, had generally tried out more and different formalisation solutions, and hence come up with better ones on average, then those that had used the standard syntax.

After this experience, I applied the shorthand to the OCL contained in my work of creating a UML Profile for small-scale enterprise integration, to save print space. Section 4 shows two Well-formedness Rules from the UML 1.4.2

standard. I then showed parts of the work to some colleagues who had previously criticised OCL as 'to bulky' and 'not mathematical', to find increased acceptance, purely because of a syntactic sugar.

Trying to apply this encouraging result, some underlying challenges and requirements surfaced. Foremost, OCL is not a single language, which conforms to a single grammar or meta-model, but a collection of languages with an overlapping concrete syntax. The published grammar in the UML 1.4.2 standard for example cannot be directly converted into an LALR1 parser[12], which prompts several dialects. The OCL 2.0 standard, although a great improvement, only contains a non-normative concrete syntax section. Consequently, the use of a parser to analyse source code is not very promising. Most OCL tools however adhere to keyword and syntax conventions laid down in OCL 1.6 and to the standard library of collection functions shown there. This proposal thus uses OCL token patterns, rather than full parser analysis. This allows its application even to unstructured or broken OCL code, which broadens applicability and robustness of the approach. A technically and semantically viable form to specify such 'approximate matches' is the use of regular expressions (REs). REs are an established means to parse constructs from a token stream and have been formalized, standardized and implemented to the degree of commoditisation.

The overall approach involves three steps: Determining *what to abbreviate*, *choosing a set of glyphs* or symbols as abbreviations, which is both intuitive and available within the different technical systems used to display and print OCL, and determining *how to find* matching constructs. The paper is structured as follows. The next section introduces the abbreviation syntax, treating both the keyword selection and glyph system. Section 3 describes the portable technical specification using REs, and section 5 investigates related approaches. Section 6 presents an outlook.

2 Abbreviating OCL

An abbreviation is a mapping from the range of OCL texts using keywords to the domain of OCL texts using symbols. In section 2.1, we choose a common symbol set for the domain and discuss some typeface conventions. The subsequent sections 2.2 to 2.7 discuss the range of abbreviations for structural parts, collections, enumerations types, simple and collection operations, followed, each arguing for the choice of symbol used. As a guideline symbols are introduced for OCL constructs that are mandatory, like the structural parts, or used frequently, like the zero-arity collection operations. Some operations are given shorter textual names. Each section finishes with a summary table.

2.1 Glyph System, Font and Typography

OCL source code intended for reading currently appears in two contexts: As specification text in documents and as part of (UML) models. The notation must cater to these contexts. Academics tend to use the Latex typesetting compiler to author specification documents; in the industrial context Microsoft Word dominates. In addition, specifications are occasionally rendered as HTML documents. Modelling tools these days are either based on Java or directly on the Windows operating system. We thus have to define abbreviations for two glyph systems: Latex and Unicode[1].

To present OCL more like a formula and less like a program, the notation uses a proportional serif font, like 'Times', to typeset all text. This saves space compared with a fixed-width typewriter font. However, serif fonts often do not provide math symbols, so switching of fonts may be necessary. In Microsoft Word for example, the 'Times New Roman' font supplies ASCII glyphs and a few symbols, while the 'MS PMincho' font supplies the complete Unicode 1.4 Mathematical Operators[5] glyph range from hexadecimal 2200-22FF.

The summary tables in the following section give the literal OCL token that is to be replaced, the Unicode abbreviation, symbol number and font, and the Latex code and package, if required. To allow a large degree of portability, the Latex symbols are derived from the mapping of ISO 8879:1986 entities to Latex by Vidar Bronken Gundersen, Rune Mathisen[14]. The leftmost column shows the required latex package or mode. The following sections suggest certain frequently used parts of OCL for abbreviation.

2.2 Structural Parts

OCL source code breaks into packages, which contain context statements holding invariants, definitions or pre- and post-conditions. This structure is abbreviated as follows: Open and closed boxes define the boundaries of a package. The closed box symbol further alludes to the symbol for the end of a mathematical proof. The copyright symbol represents a context. The Greek lowercase lambda represents the 'let' abstraction, as in lambda calculus. Global definitions ('def') are shown as a plus in a box, as they add derived features to the context. The three state restricting stereotypes – invariant, and pre- and post- condition – use a graphic metaphor of a program flow from top to bottom: A rhombus symbol represents an invariant. The symbol alludes to the fact that invariants have to hold before and after a change in the system. The rhombus widens, as invariant conditions are broken and narrows again, as they are restored. Pre- and Post-conditions appear as guards before (above) and after (below) the body of the method, contained in the box. The 'self'-reference of the instance is shown as an arrow reverting to its origin. The summary can be found in Table 1.

[1] Unicode is usable in HTML 4.01, Microsoft Word, the Windows operating system and the Java Virtual Machine.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	Package
package	□	25A1	T	\square	amssymb
endpackage	■	25A0	T	\block	Isoent
context	©	00A9	T	\copyright	-
inv	♢	25CA	T	\diamond	mathmode
let	λ	03BB	T	\lambda	mathmode
def	⊞	229E	M	\boxplus	amssymb
pre		2552	T	\boxDr	Isoent
post		2558	T	\boxUr	Isoent
self	↻	21BB	M	\circlearrowright	amssymb

Table 1. Structural Parts

2.3 Collections

Definition constraints increase the power of OCL through modularisation. This also leads to the usual challenges encountered in object-oriented programming languages [6]. Although the definition of type signatures is optional in OCL, explicit types are an invaluable aid in discovering errors and their use should be encouraged. To this end the lengthy syntax for collection types is abbreviated using three types of braces, common for sets in mathematics, lists in functional programming languages and bags in the Zed[3] notation. The same notation can also be used to express construction of a collection within OCL body text. The summary is found in Table 2.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	Package
Set(X)	{X}	007B / 007D	T	\{ / \}	mathmode
Sequence(X)	[X]	005B / 005D	T	[ ]	mathmode
Bag(X)	⟨X⟩	3008 / 3009	M	\langle / \rangle	mathmode

Table 2. Collections

2.4 Enumerations

Metamodels, like that of the UML, contain enumerations, which often have long names to provide clarity, like VisibilityKind or ChangeableKind. The OCL syntax requires that an enumeration value be declared with the full type and value identifier. With long names, this takes up a lot of space. In fact, within the UML standard's own OCL well-formedness rules the type-name is generally left out. We adopt this simplification and use a typewriter font to mark the enumeration

value as something extraneous to the model. This convention is obviously restricted to models, in which the labels representing the enumeration values are unique. Otherwise a mechanism is needed, which, if provided with the context, returns the intended value to the replacement mechanism.

2.5 Simple Object Operations

Many simple operators have equivalent mathematical symbols, such as the basic Boolean operators, absolute value function and string concatenation.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
<>	≠	2260	T	\not=	mathmode
and	⋀	22C0	M	\wedge	mathmode
or	⋁	22C1	M	\vee	mathmode
not	¬	00AC	T	\lnot	mathmode
implies	⇒	21D2	M	\Rightarrow	mathmode
.abs(x)	|x|	007C	T	\vert x \vert	mathmode
.concat(x)	&(x)	0026	T	\&(x)	

Table 3. Simple Object Operations

2.6 Collection Operations

Different types of arrows distinguish collection operations without parameters from those with parameters. Zero-arity operations are represented by a hook arrow, and leave out the following brace; all other operations are shown with a regular arrow with parameters inside the brace. To shorten the Latex notation further, the operation name is shown atop, rather than behind, the arrow in those notations.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
->x()	↩x	21A9	M	\atop{x}{\hookleftarrow}	mathmode
->x(y)	→x(y)	2192	T	\atop{x}{\rightarrow}(y)	mathmode

Table 4. Simple Object Operations

Collection Operation Names The following operations from the areas of set theory and predicate logic, functional programming and list manipulation, and relational-calculus are used frequently in the definition of well-formedness rules.

The operations for set-theory use customary mathematical symbols. Only the abbreviation for symmetricDifference is a composition. The two last operations -'excluding' and 'including' - use an exclamation mark to indicate that the collection on which the operation is invoked is 'changed'.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
union	$\bigcup$	22C3	M	\cup	mathmode
intersection	$\bigcap$	22C2	M	\cap	mathmode
symmetricDifference	$\bigcup - \bigcap$				mathmode
isEmpty	$\emptyset$	2205	M	\emptyset	mathmode
includes	$\in$	2208	M	\in	mathmode
excludes	$\notin$	2209	M	\not\in	mathmode
forAll	$\forall$	2200	M	\forall	mathmode
exists	$\exists$	2203	M	\exists	mathmode
excluding	$\subset$!	2282	M	\subset	mathmode
including	$\supset$!	2283	M	\supset	mathmode

Table 5. Collection Operation Names

The 'select', 'collect' and 'iterate' operations are equivalents of basic operations from the field of functional programming. The operation 'select' also appears in the Relational calculus, where it is abbreviated as the lowercase Greek letter sigma. This convention is also used here.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
select	σ	03C3	T	\sigma	Mathmode
collect	map		T		Mathmode
iterate	fold		T		

Table 6. Collection Operation Types

The 'count', 'one', 'isUnique' and 'sum' operations often occur in contexts were cardinalities need to be enforced, like in database modelling. The question mark used for the first three is meant to indicate that these are *query* operations, which either query a variable (infix use) or a Boolean property (postfix use).

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
count	|?|	007C	T	\vert ? \vert	mathmode
one	|1|?	007C	T	\vert 1 \vert ?	mathmode
isUnique	key?				
sum	$\sum$	2211	T	\sum	mathmode

Table 7. Database and cardinality operations

Sequence operations are common when building constraints on access structures and in functional programming. The 'any' function is renamed, as it introduces non-determinism, which probably deserves greater recognition.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
append	◁	22B2	M	\triangleleft	mathmode
prepend	▷	22B3	M	\triangleright	mathmode
subsequence	sub		T		mathmode
at	#		T	#	mathmode
any	rnd!		T		

Table 8. Sequence and list operations

2.7 Types, Casts and States

Multiple inheritance and *def*ined additional attributes and operations often require the use of type operators in OCL programs. Unfortunately, the type operators are relatively unwieldy, making it harder to write type-safe operations. We abbreviate the type and kind concepts with Greek lowercase letters Tau and Kappa, followed by a question mark for a predicate and an exclamation mark for a cast. Similarly, the state and creation predicates are shown as a Greek lowercase sigma (end of sentence variant) and nu. These are shown in Table 9.

3 Mapping Mechanism and Strategy

In order for the approach to work, the keywords and syntactic constructs outlined in the previous section cannot be used as variable names, as REs are not aware of the context of occurrence. As all abbreviation patterns are disjunct, the mappings are bijective. Thus, the concrete syntax notation can be used as a pivot to translate, for example, Latex representation to HTML representation. Each mapping is set up as a set of RE search-and-replace pairs.

Syntax	Unicode			Latex	
	abbrev.	Code	Font	Code	package
oclIsKindOf(X)	κ?	03BA	T	\kappa ?(X)	mathmode
oclIsTypeOf(X)	τ?	03C4	T	\tau ?(X)	mathmode
oclAsType(X)	τ!	03A4	T	\tau !(X)	mathmode
oclInState(X)	ς?	03C3	T	\varsigma ?(X)	mathmode
oclIsNew()	ν?	03BD	T	\nu?	mathmode

Table 9. Types, Casts and States

3.1 Unicode

Translation between concrete syntax and Unicode does not have any issues. Since both glyph systems do not allow font information, the abbreviation of enumerations cannot be performed.

3.2 HTML

HTML 4.01 is used as the translation target standard. Translation from concrete syntax to HTML uses the numeric codes for Unicode entities. To express enumerations, the HTML 'code' tag is used. The advantage of this tag is that it is less presentation oriented, and hence less likely to be affected by presentation mechanisms like Cascading Style Sheets. The abbreviation of enumerations must be specifically fashioned for each meta-model. The HTML syntax does not use the Math-ML standard. Math-ML is intended for the exchange and presentation of mathematical equations, while OCL is a computer language. The HTML syntax also does not use textual entities, although they could be mapped in an additional step.

For the mapping from concrete syntax, the text is first converted to HTML without any change, then, the regular expression are applied. In order to work on the encoded text, the regular expression also have to be encoded to use HTML conventions. For all textual matches, the result is identical. The only clashes with the HTML syntax arise for the equality ('<>') and collection operation tokens ('->'), which use the entities < and > instead.

For the mapping to concrete syntax, the process is reversed. First, the abbreviations are expanded to regular HTML, than HTML is converted back to concrete syntax.

3.3 Latex

Latex 2e and the ISO entity package with its transitive dependencies are the basis for the Latex mapping. All glyphs used in the abbreviations described above are

available in math mode. Thus, the OCL source code is translated to be included in a 'displaymath' environment. This allows simplified later editing of documents, because symbols do not have to be escaped. Further, it enables simple line numbering, using the 'equation' environment. Latex command syntax does not clash with OCL syntax, as the backslash character is not allowed in any identifiers; It is in the 'inhibitedChar' character set of the OCL 1.6 grammar[11]. Like in the case of Unicode, the translation from concrete syntax is straightforward. It can be reversed easily, after any additional latex formatting, like additional line breaks, has been removed.

3.4 Implementation

The translation is implemented as a Java class, which is based on the Java Regular expression facility, which provides an implementation based on the POSIX standard. The Microsoft .Net runtime also provides a regular expression engine. The translation tables described above are laid down in comma separated value files based on the ASCII character set. Unicode characters are abbreviated using the character escape mechanism. The Java class is provided with an input stream, encoding and direction, and provides an output stream. The class only applies the regular expression specified. If special processing steps are required, as is the case with HTML, the class is wrapped up with an additional layer, which provides the processing. In this implementation, a utility class from the Java version of the W3C HTMLTidy project is used to perform the encoding.

4 Examples and Savings

As stated in the introduction, the examples below were taken from the UML 1.4.2 standard. Obviously, the quality of the notation cannot be assessed by a simple count of original and reformatted glyphs. In our experience, page space required to fit all of the UML's WFRs was reduced about a quarter, while legibility seemed improved. This observation is obviously subjective. Especially the question, whether and when the advantage of brevity offsets the additional learning cost for the notation in novice users, probably cannot be decided without a larger work-efficiency study based on sound methods of organisational or didactic psychology.

On the examples below, it is worth noting, that in the layout of the UML standard, the WFRs take up 9 and 10 lines respectively. To allow a fairer comparison, the original source was reformatted not to break the page margin. Access space in the pretty printed version was used to convey logical structure of the statement.

4.5.3.10 Component [3] A Component may only have as residents DataTypes, Interfaces, Classes, Associations, Dependencies, Constraints, Signals, DataValues, and Objects.

$\circlearrowleft$.allResidentElements$\xrightarrow{\forall}$(re | re.κ?(DataType) $\vee$ re.κ?(Interface) $\vee$ re.κ?(Class) $\vee$ re.κ?(Association) $\vee$ re.κ?(Dependency) $\vee$ re.κ?(Constraint) $\vee$ re.κ?(Signal) $\vee$ re.κ?(DataValue) $\vee$ re.κ?(Object))

```
self.allResidentElements->forAll( re |
re.oclIsKindOf(DataType) or re.oclIsKindOf(Interface) or
re.oclIsKindOf(Class) or re.oclIsKindOf(Association) or
re.oclIsKindOf(Dependency) or re.oclIsKindOf(Constraint) or
re.oclIsKindOf(Signal) or re.oclIsKindOf(DataValue) or
re.oclIsKindOf(Object) )
```

4.10.3.4 Collaboration [1] All Classifiers and Associations of the ClassifierRoles and AssociationRoles in the Collaboration must be included in the namespace owning the Collaboration.

$\circlearrowleft$.allContents$\xrightarrow{\forall}$(e |
(e.κ?(ClassifierRole)$\Rightarrow$ $\circlearrowleft$.namespace.allContents$\xrightarrow{\in}$(e.τ!(ClassifierRole).base))$\wedge$
(e.κ? (AssociationRole)$\Rightarrow$ $\circlearrowleft$.namespace.allContents$\xrightarrow{\in}$(e.τ!(AssociationRole).base)))

```
self.allContents->forAll(e|(e.oclIsKindOf(ClassifierRole) implies
self.namespace.allContents->includes(e.oclAsType(ClassifierRole).base))
and (e.oclIsKindOf(AssociationRole) implies self.namespace.allContents
->includes (e.oclAsType(AssociationRole).base)))
```

5 Related Approaches

The Object-Z community uses a similar mechanism for dealing with different representations within the set of the Common Zed Tools (CZT)[10]. Here, a standard Unicode representation is used as the pivotal representation of the Zed and Object-Z languages to produce other renderings. However, that approach depends on a complete parse of the source code. Also, Unicode encoded Zed sources are quite rare and latex representation is not easily convertible into it. The B language also offers a similar facility within the jBTools suite[13], which allows conversion to HTML. However, in this suite, the full complexity of the B language is restricted on input from concrete syntax in order to allow conversion to an XML intermediate format known as B-XML. This is due to the fact that the aim of the suite is to provide further services beyond presentation, like type checkers and provers.

6 Summary and Outlook

In this paper we have shown how a flexible mechanism to abbreviate OCL concrete syntax of different versions can be defined, used to transfer code between main areas of use and flexibly implemented. Beyond this, the RE replacement approach could further be used to remedy some shortcomings within the standard and fix tool incompatibilities.

In this context it acts much like a macro-processor would, transforming an concrete syntax with abbreviations into an expanded form. Three examples for such a

scenario would be a templating mechanism for transitive closures, the avoidance of non-deterministic behaviour caused by the 'any' and 'asSequence' operations and fixing parser incompatibilities. We will focus on the last two examples here.

6.1 Non-deterministic Behaviour

As OCL is an expression language, and makes intensive use of iterators, optimisation options for its execution strongly depend on determinism of sub-expressions. Although expressive in theory[2], the 'any', 'asSequence' and 'iterate' operations violate execution determinism. 'any' yields a (potentially different) element of a collection on each execution. For this reason, the operation has been renamed 'random' in the above listing. With a macro mechanism as explained in the preceding section, it would be possible to replace the any operation with a deterministic variant specified (*def*ined) in OCL. As a result, there would not be ambiguities in the computational semantics. The 'asSequence' operation returns the content of a collection as a sequence. Order is non-deterministic. Any order-dependent operations based on the resulting list may hence yield different results. Here, the replacement mechanism could require the use of a sorted operation, whose comparison predicate definition has to be provided by the author. The predicate could be written to accept the most general type 'OclAny', and then list case choices for each class for which a comparison is implemented. Finally, the 'iterate' operation is defined on collections, which leads to the same problem as indicated for the 'asSequence' operation. With replacement, uses of iterate could generally be restricted to be prefixed with a cast to a list, using the safe version of 'asSequence' previously outlined.

6.2 Parser incompatibilities

OCLE of Babes-Bolyai University [4], the Dresden OCL Toolkit[9] and the Kent Modelling Framework [1] are three major tool suites for OCL. Except for the Dresden Toolkit, all of them use textual notation to interchange OCL and all use parsers with slightly different grammars. The differences between those notations are often minor. For example, OCLE uses an optional 'model' construct to denote the underlying model in a file, while the Dresden toolkit insists that a specification should start with a 'package' statement. KMF does not expect any structural parts, but works with OCL Expressions only. Such minor inconsistencies could be resolved with the same infrastructure used above to achieve the abbreviation markup.

References

1. David H. Akehurst and B. Bordbar. On querying UML data models with OCL. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 91–103. Springer, 2001.

2. Thomas Baar. Non-deterministic constructs in OCL – what does any() mean. In *Proc. 12th SDL Forum, Grimstad, Norway, June 2005*, volume 3530 of *LNCS*, pages 32–46. Springer, 2005.
3. S. M. Brien and J. E. Nicholls. Z base standard. Technical Monograph PRG-107, Oxford uiversity computingh Laboratory, November 1992. Accepted for standardization under ISO/IEC JTC1/SC22.
4. Dan Chiorean, Mihai Pasca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring UML models consistency using the OCL environment. *Electr. Notes Theor. Comput. Sci*, 102:99–110, 2004.
5. Unicode Consortium. *The Unicode Standard, Version 2.0*. Addison Wesley Publisher, Reading, Mass., 1998.
6. Alexandre L. Correa and Cláudia Maria Lima Werner. Applying refactoring techniques to UML/OCL models. In Thomas Baar, Alfred Strohmeier, Ana M. D. Moreira, and Stephen J. Mellor, editors, *UML*, volume 3273 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2004.
7. Mark Davis. Unicode regular expression guidelines. Unicode Technical Report 18, The Unicode Consortium, P.O. Box 700519, San Jose, CA 95170-0519, USA, Phone: +1-408-777-5870, Fax: +1-408-777-5082, E-mail: `unicode-inc@unicode.org`, 1999.
8. Birgit Hofreiter, Christian Huemer, and Werner Winiwarter. OCL-constraints for UMM business collaborations. In Kurt Bauknecht, Martin Bichler, and Birgit Pröll, editors, *EC-Web*, volume 3182 of *Lecture Notes in Computer Science*, pages 174–185. Springer, 2004.
9. Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *Proc. 3rd International Conference on the Unified Modeling Language (UML)*, volume 1939 of *LNCS*, pages 278–293. Springer-Verlag, 2000.
10. Petra Malik and Mark Utting. CZT: A framework for Z tools. In *ZB*, pages 65–84, 2005.
11. Object Management Group (OMG). *Unified Modeling Language Specification, Version 1.4*, September 2001. `http://cgi.omg.org/docs/formal/01-09-67.pdf`.
12. Bernhard Rumpe. <<java>>OCL based on new presentation of the OCL-syntax. In Tony Clark and Jos Warmer, editors, *Object Modeling with the OCL*, volume 2263 of *Lecture Notes in Computer Science*, pages 189–212. Springer, 2002.
13. Bruno Tatibouet. The jBTools b-suite for JEdit, 12 2003.
14. Rune Mathisen Vidar Bronken Gundersen. ISO character entities and their LATEX equivalents, 12 2001.