Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Reflections on OCL 2.

Edward D. Willink[a]

a. Willink Transformations Ltd., Reading, UK,

Abstract

Twenty years after the OCL 2.0 Request For Proposals, it is perhaps long overdue for a review as to how well the resulting semi-formal OCL 2 specification makes the informal OCL 1 specification more precise. We briefly relate the history that allowed an imprecise draft to be adopted by the OMG as the OCL 2.0 specification resulting in a language that was fatally flawed from the outset. We draw on personal experience to explain why recognition of the fatality of the flaws has taken so long. However despite these flaws, OCL remains the language of choice for specifying model constraints. Therefore armed with an understanding of the flaws, we make practical suggestions for how an OCL 3.0 might resolve them.

Keywords OCL, Object Constraint Language, Precise specification, Side effect free

1 Introduction

The UML specification arose to resolve the 'method wars' that left users confused as to whether a class should be drawn as a cloud or rectangle. In this respect UML has been a total success, but obviously there is more to modeling than classes, and as soon as models become non-trivial, additional constraints are required that cannot be sensibly expressed graphically. The UML 1.1 [Obj97b] suite of documents therefore includes a document defining the OCL 1.1 [Obj97a] textual language that can elaborate UML diagrams.

The OCL language evolved from work on the Syntropy method at IBM, and in so far as many thousands of academic papers have successfully used OCL to express constraints, OCL too has been a total success.

In Section 2 we relate the history of the OCL 1 to OCL 2 transition before examining the problems with OCL 2. We firstly examine the language problems directly affecting users in Section 3 and then the specification problems that directly affect toolsmiths and indirectly affect users in Section 4. Then in Section 5 we consider what a useful OCL Building Block might look like and what an OCL 3 might do to support it. Finally in Section 6 we conclude.

Edward D. Willink. *Reflections on OCL 2.*. Licensed under Attribution-NoDerivatives 4.0 International (CC BY-ND 4.0). In Journal of Object Technology, vol. V, no. N, 2011, pages M:1–21.
doi:10.5381/jot.201Y.VV.N.aN

2 OCL 1 to OCL 2 History

The software community was not satisfied to leave UML (and OCL) as a useful flexible semi-formal facility for communicating analysis considerations. Rather a near-formal semantics for UML (and OCL) was required to specify behavior precisely and so facilitate synthesis of functional code direct from UML diagrams.

The formality of OCL was addressed by Martin Gogolla's student Mark Richters whose PhD thesis [Ric02] was adapted to provide the formal Annex that accompanies all the OCL 2.x specifications. Ir provides a useful reference to help resolve issues in the main specification, however since the Annex has not tracked all the OCL 2 evolutions, sometimes the Annex just contributes to a contradiction for implementers and users to reconcile.

2.1 OCL 1.5

Reviewing the final version of the OCL 1 specification embedded within the UML 1.5 specification [Obj03b], we find a nice simple 50 page informal exposition of:

- the language from a user perspective
- the library operations from a user perspective
- an EBNF grammar

This could be called a black box specification. It reveals what the user sees. It imposes no limitations on how an implementation satisfies the specification. With so little detail, most of the problems of the 210 page OCL 2.0 [Obj06] do not exist. However some do.

OCL 1 supports open classes in so far as let-operations and let-attributes define new pseudo-operations and pseudo-attributes for the exclusive use of the OCL. No clues about how pseudo-operations and pseudo-attributes are modeled in UML is given.

OCL 1.5 claims that 'A parser generated from this grammar has correctly parsed all the constraints in the UML Semantics section, a process which improved the correctness of the specifications for OCL and UML.' This is patently untrue since skipping over obvious formatting typos such as the `Guard` invariant, many of the constraints have failed to track the language evolution that mandates empty parentheses on parameterless operations and replaces the `#` syntax for enumerations by qualified names.

UML 1.x has `Attribute`s, `AssociationEnd`s, `Operation`s and `Method`s for which OCL 1 uses the generic term 'property'. Unfortunately when UML 2.0 unified the `Attribute` and `AssociationEnd` classes as the `Property` class, the OCL text failed to track uniformly; the resulting OCL 2 text can be rather confusing.

OCL 1.x uses the term 'stereotype' to refer to the `inv`/`pre`/`post` discriminant. This is confusing to any reader who may be familiar with the UML `Stereotype` class and its capabilities.

OCL 1.x specifies that all collections are always flattened.

OCL 1.x recognises that non-collection values may be 'null', but does not provide any clues as to what this may mean semantically.

OCL 1.x does not specify what happens if the index-is-in-range precondition for `Sequence::at` is not satisfied. OCL 1.x does not specify a no-divide-by-zero precondition on `Real::/`.

OCL 1.x specifies that the Standard Library is a modeled `Package` and specifies how it may be extended by another `Package` using an ≪`OCL_Types`≫ dependency. No model for the Standard Library is provided and so the specified extension is not practical.

2.2 UML 2.0 submissions

The two competing UML 2.0 submissions [Obj02],[Obj03a] both recognized the utility of OCL and also recognized that OCL had utility beyond UML. OCL was therefore excluded from the responses to the UML 2.0 RFP [Obj00b] and treated as a new self-standing specification with its own RFP [Obj00a]. This separation was very convenient for the UML teams and in many respects good for OCL too, but unfortunately the separation from UML and the drive to better formality required many extra problems to be addressed. When the UML teams ran out of enthusiasm / resources, the result was a work in progress draft [Obj03c].

The draft OCL sat on the shelf at OMG for three years until seven out of eight of the competing QVT specification submissions agreed that the QVT specification should exploit OCL. The QVT specification [Obj08] could not be adopted until the OCL 2.0 specification had been adopted and so the work in progress draft was dusted off, polished slightly and adopted leading to the official OCL 2.0 [Obj06]. It is unclear how this could have happened since the draft clearly lacked the required prototyping required by OMG, and contains many TBDs to be resolved once UML 2.0 was finalized. UML 2.0 [Obj03d] was of course adopted three years prior to this OCL 2.0 adoption. The TBDs persist to this day and are even present in OCL 2.3.1 [Obj12] which was adopted as an ISO standard; perhaps the only ISO standard with explicit TBDs and prolific known inconsistencies.

2.3 OCL 2.0 aspirations

Whereas OCL 1.x was a black box specification, OCL 2.x is a white box specification. It specifies the Abstract Syntax (AS) model for OCL with the excellent intention that this should facilitate interchange using XMI between alternative OCL tools. Unfortunately this model was not provided until OCL 2.2 [Obj10] and even then it is not quite right since no prototype had been built.

As an evolution of OCL 1.x, it was natural for the OCL AS model to re-use UML metaclasses. However once the UML-OCL connection was severed, the use of such a bloated and in some respects inadequate foundation should have been reconsidered. The re-use of UML became untenable once OCL 2.0 Section 13 added the claim that EssentialOCL could work with EMOF.

Specification of the AS model required that the conversion between the grammar and the AS be specified as well. This is achieved by specifying a non-normative Concrete Syntax (CS) model that closely resembles the grammar, and a variety of rules mapping grammar to CS and CS to AS. Since the grammar is ambiguous, a further category of disambiguating rules is required. The exposition of this conversion burden is arranged around the non-normative CS classes, for which no model has been provided. This avoids revealing how far from correct the CS classes are. The coherent OCL 1.5 grammar is replaced by CS-relevant snippets scattered throughout the chapter. As a minimum this imposes a major cut and paste burden on any developer attempting to use them. More practically, it obfuscates to such an extent

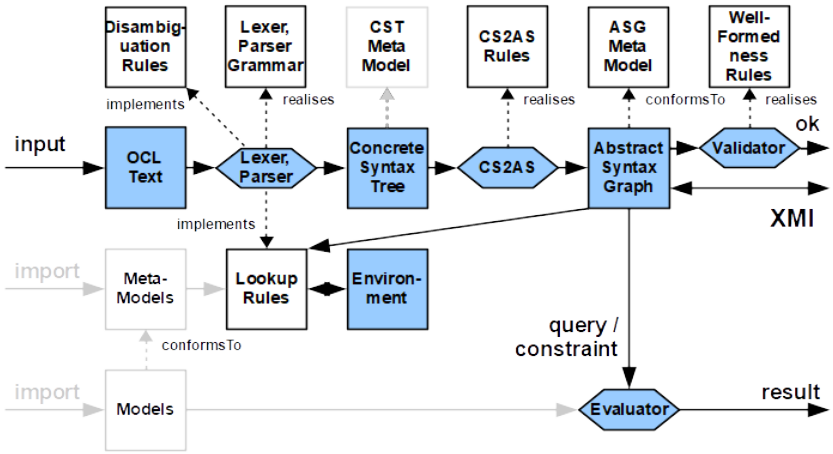

Figure 1 – OCL 2 Architecture

that a casual reader is impressed by the apparent detailed formality and unaware of the total absence of any prototyping to substantiate the unusable content.

Standardization of the AS requires that the internal awkwardness of OCL 1.x's pseudo-attributes and pseudo-operations be properly modeled. They are not and to make matters worse the pseudo-attributes and pseudo-operations are elevated to attributes and operations that can be used just as if they were part of the original model. This is tantamount to providing open classes and presents an unresolved challenge when the AS model is serialized as XMI.

The availability of an AS model provided an opportunity to specify the execution semantics. The exposition is semi-formal and uses a rather obvious and incomplete `ValuesPackage` and `EvaluationPackage`.

The `EvaluationPackage` makes a first attempt at temporal modeling using a `LocalSnapshot` class to maintain a history of object states. I'm not aware of any implementation that uses this aspect of the specification. The relatively recent work on sequences of system snapshots (filmstrips [DGF17]) by Martin Gogolla's team at Bremen [USE] seems much more promising.

2.4 OCL 2 Architecture

Fig 1 shows the components of the architecture implied by the specification. Along the top row, transparent boxes show different aspects of specified behavior that configure the shaded concrete boxes realized by the tooling on the second row. Input text is lexed and parsed to give a Concrete Syntax Tree that is converted to an Abstract Syntax Graph and then validated.

The troublesome ability to interchange the ASG using XMI is indicated.

A non-standard parser is required since it must implement the disambiguation rules and lookup rules. The lookup rules cannot be expressed in pure OCL since they create and modify new `Environments` for each construct such as a `LetExp` that introduces a nested scope. The mechanism by which the user-metamodels are imported and the construction of a root `Environment` is greyed out since it is unspecified.

The bottom row shows how the parsed ASG provides a query or constraint that an evaluator may use to provide a result, using models whose import mechanism is

again unspecified.

Before criticising OCL 2 too harshly, it must be remembered that there were no metamodels for OCL 1 and so the metamodels for OCL 2 were a significant novelty. We should perhaps praise the OCL 2 metamodels for being perhaps 95% correct rather than dwelling on the 5% wrong. However it is the 5% wrong and the lack of a comprehensive prototype to reveal the wrongness that has caused so much trouble.

2.5 Author's Background

Since parts of this paper rely on the personal observations of one of the leading participants at OMG and Eclipse, it is appropriate to provide a selective biography to distinguish my direct and indirect knowledge.

My involvement started in around 2003 from providing Eclipse support for the UMLX model transformation language [Wil03] using the then planned QVTr language. This led to participation in the Eclipse support for QVTr and interaction with the Eclipse OCL project [Ecld] to make it extensible for use by the Eclipse QVTd project [Ecle]. I had no involvement with OCL 2.0. I contributed a few review comments to QVT 1.0.

Involvement with OCL and QVT at Eclipse led to the my appointment as the Thales representative for the OMG Revision Task Forces for OCL and QVT. I therefore contributed some revisions for OCL 2.2 and consistent models for QVT 1.1.

As personnel at OMG and Eclipse moved on, I found himself as chair of the OMG OCL and QVT RTFs and as project lead of Eclipse OCL and QVTd projects. Lack of active personnel meant that I was often the sole active participant.

I 'retired' from Thales in 2012. Since then I am grateful, firstly to Tricia Balfe at Nomos Software, and then to Cory Casanave at Model Driven Solutions, for appointing me as their OCL and QVT RTF representatives.

At OMG, I resolved the 'easy' problems in OCL 2.3 and OCL 2.4. This led to increasing awareness of the 'hard' problems and the issuing of a Request For Proposal to address these via an 'OCL 2.5' rewrite. The RFP [Obj14] can be read as a catalog of the serious OCL 2 problems.

At Eclipse, I inherited the Classic Eclipse OCL for Ecore/UML whose stable APIs made significant development almost impossible. A new fully-modeled Eclipse OCL exploited Xtext to provide much enhanced UX and a Pivot model to unify the competing Ecore and UML needs. Enhanced APIs and Java code generation support extension for QVT. The Pivot-based Eclipse OCL prototyped many solutions for OCL specification problems. Many of the solutions have been presented to the annual OCL Workshop. Unfortunately the need for API stability has become a hindrance to further development.

3 Language Problems

Problems with the OCL language have a very direct impact on the user and may require users to program in an unnatural style to circumvent the limitations.

3.1 Program Failure

An inevitable characteristic of any manually developed program is that it may malfunction and consequently the program language and execution support must accommodate

failures. Failures typically take one of two forms.

catastrophic failure A catastrophic failure is often called a crash. It may occur from a hardware, software, network, or I/O system failure. Programs cannot normally recover from these failures and so the execution launcher will normally attempt to provide as much helpful diagnosis of the failure as possible before terminating execution abruptly.

recoverable failure A recoverable failure may occur when the programmer finds it convenient to reuse some failure detection code and then to compensate for the failure, typically by catching a thrown exception.

not-a-failure Conversely a programmer may correctly anticipate a failure and provide a guard to direct the program control to bypass and so avoid the failure.

In Java the two actual failure cases are separated by using `Error` or `RuntimeException` for catastrophic failures and by using `Exception` for recoverable failures. `Exception`s form part of a Java function signature and so there can be some static diagnosis of code that neglects to handle the recoverable failures.

OCL 1 has preconditions but provides no indication of how an evaluation should behave when a precondition is not satisfied. OCL 2 is similarly vague and so OCL 2 tools often treat the corresponding preconditions (and postconditions) as just syntax-checked comments to document a hazard for a human reader.

OCL 2 pursues a functional approach and, in the event of a malfunction, returns a regular `invalid` value rather than imposing an alternative 'return' mechanism for an `Exception`. The `invalid` value can be 'caught' by using the `oclIsInvalid()` library function. This is different to many languages but more regular and so perhaps better.

Unfortunately the OCL 2 specification is not really concerned with failures for which it mandates that `invalid` is a singleton; all failures are the same and free from any helpful diagnostic detail. Some OCL tools ignore this stupid restriction and provide a rich `invalid` that propagates diagnostics while preserving OCL semantics by ensuring that the diverse `invalid`s behave as one.

The lack of consideration for crashes forces an OCL 2 implementer to use the `invalid` return for all crashes.

The OCL specification provides no ability for the use of `invalid` to be declared as part of a function signature, consequently OCL, whose strong side effect free formality supports strong analysis, has a gaping hole in regard to guaranteeing that a program execution will not fail.

3.2 2-valued Booleans and invalid

Almost every gathering of the OCL community provokes discussion of why OCL Booleans are not 2-valued {`true`, `false`}. Clearly many OCL users are unhappy with the prevailing 3-valued {`true`, `false`, `invalid`} specification [1].

But this is all a misunderstanding. In OCL, as in other languages, a non-trivial Boolean-valued calculation has three possible outcomes; success/`true`, failure/`false` and crash/`invalid`. When the crash is realized as a thrown exception, the programmer can ignore the crash outcome and code as if there were only two possible outcomes.

[1] The idempotent addition of a null as a fourth output in OCL 2.4 is a further but irrelevant confusion.

Exactly the same programming approach is possible with OCL provided the programmer ensures that the particular OCL tooling API such as `check()` that is used for the evaluation is 'strict'; i.e. it converts the `invalid` or `null` value returned by OCL to a crash by throwing an exception. The OCL tooling may offer an alternative API such as `evaluate()` that returns all three outcomes as OCL values. This alternative is useful when the programmer really wants to program all three outcomes, but is the probable source of unhappiness when only two outcomes were expected.

3.3 Short-circuit operators

In many C-based programming languages short-circuit and/or operators support a guard idiom to ensure that ensures that the evaluation of a first term converts the crash that would result from evaluating the second term to not-a-failure.

```
(x != null) && x.doSomething()
(x == null) || x.doSomething()
```

OCL appears to be much the same:

```
(x <> null) and x.doSomething()
(x = null) or x.doSomething()
```

but since the operators are commutative the following must return the same results

```
x.doSomething() and (x <> null)
x.doSomething() or (x = null)
```

An optimizing implementation or a multi-processor implementation may evaluate the two terms in arbitrary order. An implementation may be unable to avoid the crash, so it must instead catch the crash from the 'wrong' term and suppress it once the guard from the 'right' term is determined. The not-a-failure is not necessarily avoided.

This is not what was intended when the Amsterdam Manifesto [CKM+99] adopted the Kleene logic to support short-circuit rather than 'strict' Boolean operators. The extended Truth Table for the `and` operation was and is.

Use Case	Input 1	Input 2	Output
2-valued	false	false	false
	false	true	false
	true	false	false
	true	true	true
Normal Short-Circuit	false	X	false
Commutated Short-Circuit	X	false	false
Residue	true	X	X
	X	true	X
	X	X	X

The table has been redrawn here to distinguish the four 2-valued Boolean cases, the two short-circuit cases and three residual cases.

In the Amsterdam Manifesto, 'X' was spelled as 'undefined' and was clearly described as a virtual value meaning not-yet-computed in order to explain how the

two short-circuit cases yield a useful result without needing to compute a redundant and quite possibly uncomputable term.

In OCL 2.0 and 2.2, 'X' is spelled as '⊥' to represent a `null` or `invalid` value. Whether the output is `null` or `invalid` was unclear.

In OCL 2.3, 'X' is spelled as 'ϵ' or '⊥' respectively for a `null` or `invalid` Input value and explicitly just `invalid` as an Output.

In OCL 2.4, 'X' is again spelled as 'ϵ' or '⊥' for an Input value but as an idempotent 'ϵ' or '⊥' as an Output.

The 2-valued cases are uncontroversial.

The Short-Circuit cases solve the problem of choosing a Truth Table row when one of the input values cannot be computed since the short-circuit rows are available for use by not-yet-computed inputs. There is no need to attempt to compute what cannot be computed.

In so far as the Residual Use Cases describe the propagation of the virtual not-yet-computed value there is nothing wrong with them. However program execution does not normally reify the not-yet-computed result; rather we twiddle our thumbs waiting for the computation to complete or fail.

The virtual not-yet-computed meaning of 'undefined' in the Amsterdam Manifesto evolved to the actual values of `invalid` and `null` in OCL 2 so that the Residual Use Cases no longer describe not-computations but failure propagation.

Similarly the commutative and short-circuit characteristics of the Boolean operators, when implemented, conflict with the not-a-failure intent of the Amsterdam Manifesto. This causes surprise to the user of a debugging or tracing tool who may observe the chaos of a failing redundant computation, and a further surprise to a user who finds that a crashing first term is rescued by a second term. This potential of the commutated short-circuit for surprise, inefficiency and implementation difficulty may justify a change to a traditional non-commutative short-circuit. It is not clear that the mathematically elegance of commutative and/or provide any practical benefits; they certainly provide considerable implementation difficulties.

3.4 Dot and Arrow Navigation operators

Newcomers to OCL are confused by the difference between dot and arrow navigation operators. Prior to OCL 2.4, the specification was unhelpful and so newcomers fail to discover the simple rule that dot is for objects and arrow for collections. The availability of the implicit-collect and implicit-as-set shortforms give the dot and arrow operators a utility for the 'wrong' sources. This can confuse even experienced OCL programmers.

```
myCollection->collect(name)      -- explicit collect
myAggregate.name                 -- ?? implicit collect ??
```

The utility of implicit-collect is mixed. Some users like the compact exposition of some constraints. Other users dislike the ease with which a typo acquires an unexpected meaning. In the second example above, the use of a singular word such as `myAggregate` makes it impossible to tell locally whether `name` is a property access of the `myAggregate` instance of a `MyAggregate` class, or an implicit collect of the elements of a `myAggregate` collection.

Prior to OCL 2.4, the implicit-as-set was ill-specified and not a shortform. The introduction of the explicit `oclAsSet` library operation formalized the shortform.

The EOL variant of OCL used by the Epsilon [Eclc] transformation languages demonstrates that it is possible to make do with just a dot operator for both objects and navigations. The user confusions are eliminated.

A clearer OCL could similarly use just a dot operator. The brevity of implicit-collect could be rescued by defining a `*.` navigation operator which reads naturally as many-dot for an implicit-collect shortform.

```
myAggregate*.name                    -- shortform explicit collect
```

A similar `.*` shortform which reads as dot-to-many could rescue the implicit-as-set, but this usage is probably too rare to merit the shortform.

3.5 Implicit Source

implicit-self OCL, like many Object Oriented languages, allows the `self` start point of a navigation to be omitted. Since the `self` context is so important, this is very reasonable and can improve readability.

```
name                  -- self.name
```

implicit-source OCL, unlike other languages, has powerful collection iteration capabilities and allows the start point of a navigation from an iterator to be omitted.

```
aCollection->isUnique(name)   -- aCollection->isUnique(e | e.name)
```

This again may aid readability by shortening the exposition. Unfortunately it also adds confusion since the tool and a reader must decide which of many possible implicit-sources in a nest of multi-iterator iterations or an implicit-self has been omitted. Typos and misunderstandings are too easy.

Within iterator bodies, only the first iterator of the most nested iteration should be available as an implicit source, self should be explicit.

3.6 Reflection and oclType()

Reflection is little used in OCL, perhaps because the OCL 2 specification has repeatedly changed the semantics of `oclType()` seemingly in an effort to find a valid way of providing access to the name of a type without imposing the baggage of a fully reflective type system.

The reflective OCL 1 seems much clearer and supports the usage within some of the OCL 2 constraints.

3.7 Precision

As a specification language, OCL specifies unbounded precision for its ideal Integer and Real calculations. This is clearly unrealistic and inefficient for many practical applications where 16 bits is often enough for counters and sequence indexes.

In practice, the type declarations of the model slots from which values are read provides a strong clue as to what precision is appropriate, but it is only a clue. To avoid implementation guesswork, there should be a mechanism for OCL evaluations to specify precision, overflow and underflow behavior.

3.8 Libraries

One of the most impressive characteristics of Java is how its Object polymorphism enabled it to launch with powerful Collection libraries that have grown and grown. C++ lagged horribly. OCL still lacks support for standard or user libraries the most obvious of which would be a maths library.

4 Specification Problems

Problems with the specification are mostly a concern for toolsmiths, since they must work hard to find workarounds for the difficulties. The problems are only apparent to users when the workarounds lead to disappointing or confusing functionality or incompatibility between alternative tools. Sadly the two best OCL implementations, USE and Eclipse OCL, are seriously incompatible and address very different use cases.

4.1 OCL 1.1 Grammar

Prior to the availability of LALR support tools such as yacc [Ste75], it was understandable that language specifications might be imperfect; thus C suffers from the notorious dangling-else ambiguity which all C/Java programmers learn about the hard way.

With the availability of yacc, it is inexcusable for any language to fail to provide a yacc grammar. The OCL 1.1 grammar dates from 1997 (22 years post-yacc). It is an EBNF grammar that can be converted to LALR form without too much trouble. Converting to LALR form reveals a multiplicity bug, a name conflict and lexer comments that are easily resolved. It also reveals a very serious shift/reduce conflict when parsing a property call such as `x.y(z1, z2, z3)`. A potentially large lookahead is needed to search for the `|` that distinguishes the iteration call `y(z1, z2 | z3)` from the operation call `y(z1, z2, z3)`.

The OCL 1.1 grammar is bad; it is incompatible with standard tooling. If LALR tooling had been used, the OCL syntax would have been adjusted.

The free parser advertised by the specification is no longer available from the IBM website making it impossible to determine how the ambiguity was resolved.

4.2 OCL 2.1 Grammar

At least the OCL 1.1 grammar exists as a nearly coherent whole; it can be cleaned up from a cut and paste from the specification PDF. For OCL 2.0, which is only a draft, the grammar was split up and interleaved with inherited and synthesized attribute rules. Distinct rules 'clarify' each different form of navigation. This introduces many ambiguities necessitating some disambiguation rules. These difficulties are aggravated by partial name refactorings corresponding to work in progress tracking UML 1 to 2 evolution. Further difficulties arise from incomplete evolution to accommodate qualified names and static operations.

Eventually I have come to accept that the OCL 2.x grammar and CS rule specifications are not fit for purpose. Each implementer is obliged to empathize with the spirit of the specification and code accordingly. It is not surprising that few tools fully support the complexities of unnavigable opposite navigation or association classes.

I developed a yaccable version of the OCL 2 grammar for inclusion in the OCL 2.3 revision, but retracted it at the last moment when it became clear that the left recursion typical of LALR tools was troublesome for the LL tools such as Xtext [Eclg].

4.3 Other Problems

There are too many other problems to address here. Some are listed in the 'OCL 2.5' RFP [Obj00a] and others as part of the revisions in Section 5.2.

5 OCL Building Block

The liberation of OCL from UML was intended to make OCL more generally useful. It is therefore particularly irritating when a user asks 'how can I re-use OCL in my application'? This is irritating because the honest answer is that you can't unless you devote considerable skilled programming effort. Why is it so hard?

5.1 OCL Re-Use Cases

We first examine a couple of use cases that OCL could respond to and then look at how OCL could make them much easier. This leads us on to some ideas for OCL 3.

5.1.1 Novel OCL

If the user has a novel application such as using OCL as a replacement for XPath in the XML/XSD technology space, there are two obvious choices.

Re-implement A custom implementation can obviously satisfy all the user's requirements, but it requires the user to become familiar with all the complexities of OCL and to rediscover solutions to the many inadequacies of the OCL specification.

Re-Use Re-use of existing functionality is often preferable, particularly if a re-usable implementation is available. Unfortunately the lack of a clear architecture in the specification encourages the proprietary struggles for solutions to pervade the implementation. It is not re-usable.

Wilke [WTW10] highlighted the lack of architecture nicely by identifying that an OCL implementer had two significant design choices to accommodate the user's preferred metamodel representation (UML, Ecore, XSD, Java, ...) and a further two design choices for the user's preferred model representation (Ecore, XML, Java, ...).

Denormalized metamodels If the OCL functionality is to specify expressions for a particular metamodel representation, the OCL tooling can be coded specifically for that representation. In practice this means substantial re-tooling for each new metamodel representation. When the Classic Eclipse OCL support for Ecore was enhanced to support UML as well, an attempt was made to mitigate the costs of this re-tooling by introducing long (ten) template parameter lists and a reflective class to polymorphize the non-polymorphic Ecore/UML classes. This led to unpleasant code for all representations and probably made the prospect of supporting a third representation even more daunting.

Normalized metamodels Alternatively, the OCL functionality can be defined for a normalized metamodel representation. There is then no need to re-tool for another metamodel representation since the OCL tooling using the normalised metamodel is unaffected. It is just necessary to convert the user's new representation to the normalized representation. Dresden OCL [Dre] coined the term Pivot model and realized it by a family of adapter classes. The Pivot-based Eclipse OCL performs

a full model transformation from Ecore or UML to Pivot taking advantage of the transformation stage to normalize bloated irregular UML concepts such as `Stereotype`s and `AssociationClass`es.

The cost of providing a new normalization for a new metamodel representation is much less than the cost of re-tooling to denormalize OCL. Since there are comparatively few metamodel objects in an application, the extra memory cost of dual metamodel objects is acceptable.

Denormalized models When evaluating OCL expressions, it is necessary to access the user models which naturally exist in a denormalized form. This could require re-tooling the evaluator to use the denormalized representation.

Normalized models Alternatively each user object could be translated to a normalized form for use by a normalized evaluator.

For the potentially very large numbers of user objects, creating a normalized version of each is unattractive since it is liable to double memory consumption. Conversely re-tooling to denormalize all the OCL library routines that support Boolean, Integer, Real and String calculations is also unattractive. A halfway house is much more practical; use the normalized representation for all the built-in values and the denormalized representation for the user objects. It is then only necessary to perform normalizing conversions as part of the property call evaluation facility that fetches a value from the slot of a user object.

From these considerations we can see that a user with a novel metamodel and model representation could hope to get away with coding

- a custom metamodel to normalized pivot metamodel transformation
- custom model property access conversions

5.1.2 Bigger OCL

Alternatively a user may be interested in using OCL as part of a bigger system such as a model transformation language. This is the use case that caused QVT to rescue the OCL 2 draft from oblivion.

It is desirable that the bigger system can re-use as much of the basic OCL as possible and one would certainly hope that the basic evaluation functionality would be reusable; only minor extension should be needed for additional library routines. Extension is self-evidently easier if the specification provides neutral extensible machine readable expositions such as grammars, metamodels and rules rather than pseudo-code or code. Tool quality is also much improved since code that is auto-generated from grammars, metamodels and rules shares the debugging efforts of other auto-generators. Residual auto-generation bugs tend to have really obvious catastrophic effects.

5.1.3 Summary

No matter whether extension occurs for source or compiled functionality, it is very desirable that OCL maximises its ability to be-re-used by providing genuinely re-usable grammars, models and rules. This is the approach that has been pursued by the Pivot-based Eclipse OCL with extension for QVTc or QVTr. However the poor quality of the OCL and QVT specifications have made the extension rather hard.

If OCL is to be easy to extend, it must have a clear specification with a clear architecture. Once these are clear there are opportunities for an implementation to

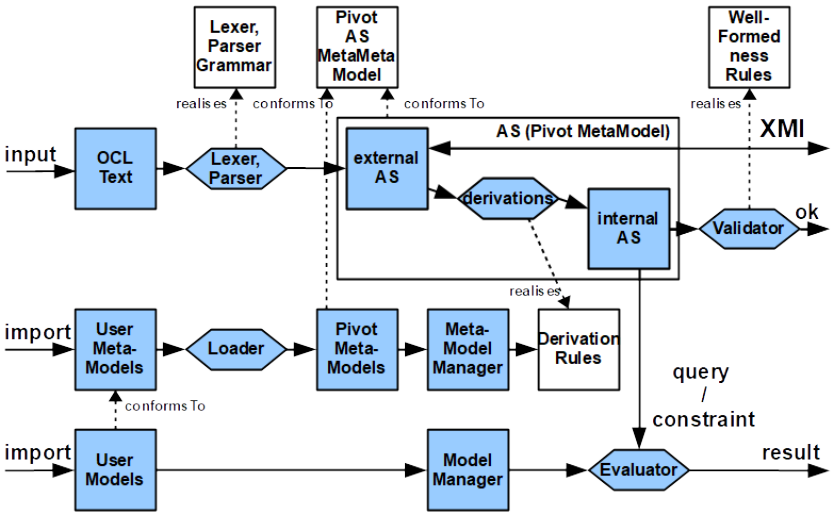

Figure 2 – Possible simplified OCL 3 Architecture

exploit this clarity to produce a correspondingly clear tooling implementation. The clear implementation facilitates selective 'borrowing' by a user who really wants to rewrite, and selective overriding by a user who is interested in re-use.

Unfortunately the significant omissions from the OCL specification result in the rather vague architecture shown in Fig 1 that practical implementations may ignore completely or revise in proprietary fashion. The net result is poor quality incompatible tools that discourage re-use.

5.2 Specification Revisions

Once we accept the need to make OCL much simpler and much more re-usable, what should be changed? The main complexity comes from the two distinct CST and ASG metamodels aggravated by the poorly designed grammar that mandates non-standard tooling to support the disambiguation with untimely semantic insights. The grammar can be improved to solve the aggravations, but two metamodels seem unavoidable since the CST is similar to the grammar to ease parsing. The ASG is a compact and sensible information model to facilitate efficient use for execution and analysis.

Exactly the same problems occur in the QVT specifications and so I sponsored Adolfo Sanchez-Barbudo Herrera's EngD [Her17] to provide automated tooling to assist in the awkward CS2AS conversion. This work started in 2013 and so at that time we still lacked the confidence or insight to call out the OCL 2 specification approach as fundamentally unsound.

Let's challenge the presumption that two metamodels are necessary. Figure 2 shows a simpler and more regular architecture that might be adopted for OCL 3. We will contrast it with Figure 1 to indicate how problems vanish.

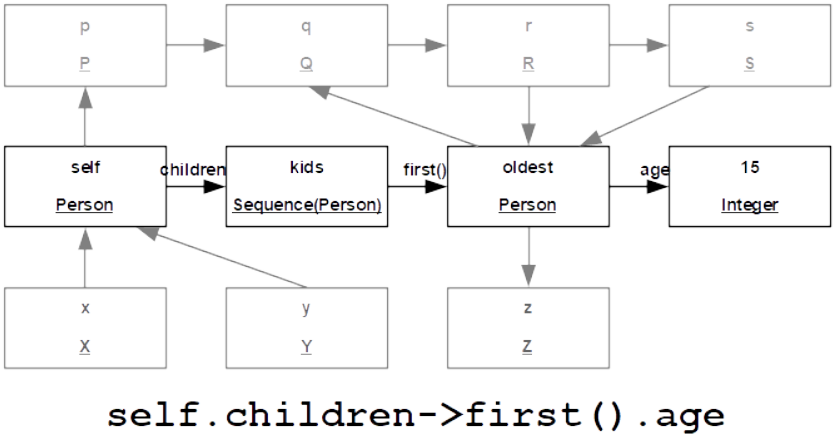

Figure 3 – OCL Navigation

5.3 Potential OCL 3 Architecture

The most glaringly missing, greyed out, part of the OCL 2 architecture is the support for user and built-in metamodels (and models). The missing import mechanism, root `Environment` and treatment of specialized constructs such as `Stereotype` must be resolved.

5.3.1 Metamodel Representation

The enduring success of OCL despite is limitations is probably due to its fundamental simplicity. The state of a system is defined by its many constituent state objects some of whose slots have simple datatype values, while other slots interconnect the objects to form a mesh. OCL evaluation just involves starting at some object, typically known as `self`, and then taking navigation steps from object to object.

Figure 3 shows a system comprising some arbitrary instances `p` with correspondingly arbitrary types `P`. In the center row, four objects of interest to an example expression are given more meaningful names and types to show how the example expression steps from object to object in a regular way even though the expression involves collection, operation and datatype complexities.

The figure is clearly a graph in which the nodes can be modeled as instances of classes or as values of specialized classes often called datatypes. These concepts are adequately supported by a variety of metamodel representations.

Edges are more troublesome, since an edge is meaningless without two nodes to define its ends. A minimal modeling of a bidirectional edge that links two nodes identifies the pair of ends. EMOF and Ecore do exactly this with their `Property` and `EReference` classes. Unfortunately UML, on which OCL 2 is based, is much more complicated. The edge is modeled by an `Association` with the ends modeled by `AssociationEnd`s in UML 1 or `Property`s in UML 2. UML adds the even more powerful confusion of an `AssociationClass` to treat the edge as a node. The UML complexity is clearly excessive for a simple regular OCL treatment.

A further complexity arises with unidirectional edges for which, given a known starting instance, it is only necessary to identify the far edge. EMOF and Ecore may therefore have a property contained by the source instance and referencing the remote

instance. Remote navigation is almost impossible. UML also supports unidirectional edges, but changes the containment of the unnavigable end to the association.

As a specification language, it is important for OCL expressions to be able to navigate in both directions [Wil16]. But this was not possible using EMOF (or Ecore) until I caused a `Tag` (or `EAnnotation`) solution to be adopted.

The diverse UML, EMOF or Ecore modeling of edges are all clumsy and a poor basis for a friendly OCL representation. The irregularities are easily normalized away during a transformation to the normalized Pivot metamodel which always has a pair of `Property` instances to define a bidirectional link between `Class`es and a single `Property` instance to define the unidirectional link from a `Class` to a `DataType` value.

Another source of UML complexity arises from `Stereotype`s. These are completely ignored by the OCL specification. The UML 2 specification provides only a minute example from which to extrapolate the, perhaps only possible, coherent design. `Stereotype`s are readily normalized to `Class`es and `Property`s to facilitate regular OCL navigation.

5.3.2 Metamodel Management

Returning to the proposed architecture in Figure 2, we see the metamodel management on the third row with a defined import mechanism for the user metamodels followed by a loader transformation that converts the user's preferred metamodel representation to the normalized Pivot representation. The loading and storage is supervised by the `MetamodelManager` which also provides the requisite abilities to query the metamodels to locate a required metamodel element from its description.

The `Loader` transformation accommodates alternative user metamodel representations and normalization of their eccentricites.

OCL 2 neglects to specify how metamodels are imported, but it does specify an `Environment` class to perform queries at a particular scope. Unfortunately a new `Environment` instance is created and then modified for each nested scope created by for instance a let-expression. This is clearly not OCL. The `Environment` instances are passed down the CST so that each node has its own instance with all possible definitions.

Once loading has completed the `MetamodelManager` instance and its children are logically immutable; the infinite pool of all possible synthesized types already exists. It is just necessary to return a reference to the required one. Obviously a practical implementation will create them lazily on demand.

The very inefficient and non-OCL push-down of `Environment` instances can be replaced by an immutable search up of the AS.

5.3.3 Unified Metamodel

The second row of Figure 2 is similar to Figure 1 in so far as OCL Text is lexed and parsed and eventually validated. However the need for a non-standard parser is eliminated by revising the grammar to avoid the need for disambiguation rules. The distinct CST and ASG are replaced by the external and internal perspectives of a unified Pivot metamodel. The descriptive properties of the external perspective, like the CST, are populated by the parser. The definitive properties of the internal perspective are somewhat like the ASG, however since they share the same host classes, no transformation is needed. Rather a derivation rule is lazily invoked to derive the internal transient property by resolving the description from the external perspective to its definition for the internal perspective.

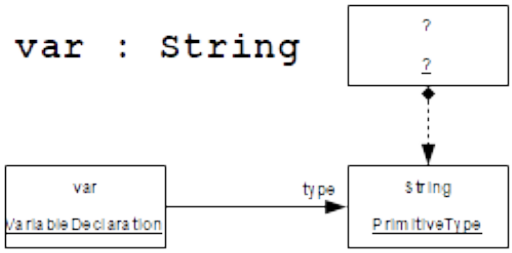

Figure 4 – OCL 2 Type Reference

The utility of the element descriptions may be demonstrated by considering the way in which a type reference is handled by the OCL 2 and OCL 3 architectures.

```
var : String
```

The partial AS for OCL 2 is shown in the Figure 4 Instance Diagram. The `VariableDeclaration` named `var` has a `type` reference to the `PrimitiveType` named `String`. The containment of the `PrimitiveType` is unclear but no doubt magically available from somewhere. The reference-to-type is simple and obvious but a containment problem appears once we consider a declaration with a synthesized type; one that must be constructed rather than just magically available.

```
var : Set(String)
```

The `VariableDeclaration::type` must now point to some type that is constructed or re-used on behalf of the reference. The OCL 2 specification provides no clues, although in response to the issue that I raised, `ExpressionInOCL::generatedType` was added in OCL 2.2 without any indication of how it solves the problem or ensures uniqueness.

The `MetamodelManager` brings order to this anarchy by supervising the model elements that may be built-in to the tooling, form part of a library, form part of a user metamodel or which may be constructed on demand to satisfy the need for a unique definition of a synthesized type. Figure 5 shows a `MetamodelManager` with transitive containment via omitted packages and models to the built-in library `PrimitiveType` instance and the synthesized `SetType` instance.

In order to isolate the parser from the complexities of locating unique definitions, the `VariableDeclaration::type` property is changed to a derived property that is part of the internal perspective. The `VariableDeclaration::ownedTypeDesc` external perspective property is populated by the parser with a description of the referenced type. This description is locally contained by the referencing `VariableDeclaration` and so may be as complicated as necessary. In the example, the `Set` is parsed as a `TemplatedTypeDesc` parameterized by a `Set` name and `String` template parameter. The template parameter is parsed as a `SimpleTypeDesc` parameterized by a `String` name. Each type descriptor has a lazily resolved `TypeDesc::resolvedType` derived property as its internal perspective.

5.3.4 XMI

The OCL 2 specification calls for model interchange between tools using XMI but leaves many awkward challenges such as references to unnavigable opposites and references to additional operations and containment of synthesized types unresolved.

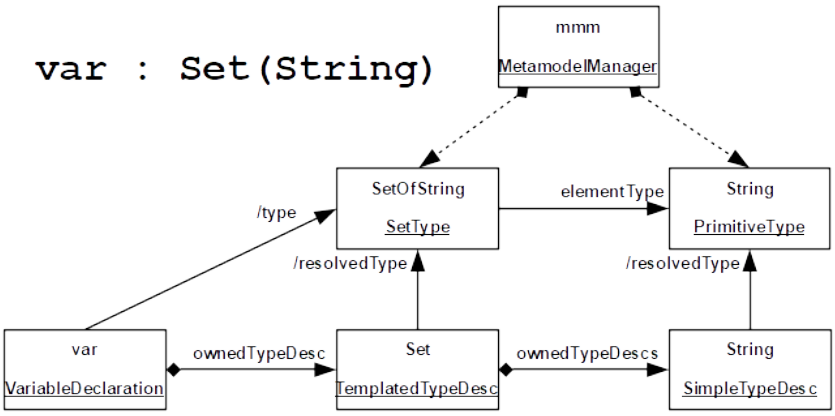

Figure 5 – Possible OCL 3 Type Descriptor

Since OCL 2 has survived without XMI for 20 years, we could just eliminate this specification point, but given a sensibly designed metamodel XMI should be easy.

The problem with XMI for OCL 2 is the need to serialize references to awkward definitions. With the two perspective AS suggested above, the external perspective comprises descriptors for the definitions avoiding the difficulties. After an XMI load, the descriptors can be resolved in a very similar fashion to the lazy resolution after parsing.

5.3.5 Open classes

The ability to add attributes and operations to classes is difficult to support when their AS representation is to be just like similar attributes and operations defined in the model. This problem led Eclipse OCL to prototype a solution whereby a complete-Class was an overlay of same-named partial-Classes enabling the additional definitions to be overlaid on the primary definition and serialized in a similar fashion. This gives a powerful but non-trivial transitive Package merge capability for same-URI packages. Is it really necessary?

Many modeling capabilities such as Acceleo [Ecla], ATL [Eclb] and QVTo [Eclf] support additional features as helper operations and helper attributes. They are clearly useful and an important aid to modularization of non-trivial OCL constraints. The deliberate avoidance of such helpers by the UML 2.5 [Obj15] specification leads to some long unreadable repetitive OCL expressions. This clearly demonstrates their utility.

However we only need helper features and so a reversion to the OCL 1.x pseudo-features would suffice. With an AS serialization that serializes references as descriptors rather than resolved references, the problem of where the referenced definition exists is no longer a problem.

5.3.6 Grammar Rationalization

Where the grammar is bad there are only two choices.

- Require skilled implementers to provide trickery to workaround the bad grammar

- Change the grammar to something sensible

The former approach is incompatible with our goal of a simple re-usable grammar, so a textual syntax change is unavoidable.

The major syntax problem arises through the introduction of prefix declarations for a long form iteration call such as

```
aCollection->forAll(e1, e2 | ...)
```

This requires undue lookahead through the iterators to find the `|` that distinguishes the iterator call syntax from the similar operation call syntax

```
aCollection->myForAll(e1, e2)
```

Most of the OCL syntax is freeform which is friendly but can be challenging. Where the freeform is too hard, additional keywords such as `let` or `endif` ensure that the parser knows exactly what is happening.

If we introduce the `var` keyword for variable declarations, we can rewrite the iteration call as

```
aCollection->forAll(var e1; var e2; ...)
```

in which each `var ... ;` clause is a prefix to the subsequent OCL expression. In this case the variable type can be inferred.

This syntax can be re-used to eliminate the let syntax by rewriting:

```
let x = ... in
let y = ... in
...
```

as

```
var x := ...;
var y := ...;
...
```

This change also solves the dangling-in parsing challenge that arises when let expressions are nested carelessly. This change should also unblock my failed prototype of an OCL extension to support QVTr-like patterns that in one trivial example avoids the clumsy need for an `oclIsKindOf(CastX)` immediately followed by a matching `oclAsType(CastX)`.

```
if (var castX : CastX := x)     -- x.oclIsKindOf(CastX)
then castX.doSomething()        -- x.oclAsType(CastX).doSomething()
else null
endif
```

6 Conclusion

We have shown how the well-intentioned upgrade of the simple but useful OCL 1 specification went astray as part of the UML 2 activities. We have identified that draft work-in-progress was accidentally adopted as the OMG OCL 2 specification.

We have drawn on personal experiences to explain why OCL tool implementers treated the OCL 2 specification with unwarranted reverence and so struggled to implement it as faithfully as possible.

It is a pity that it has taken so long to recognise the OCL 2 specification for the disaster that it is.

We make proposals for an OCL 3 that can almost be seen as going back to OCL 1 and then moving forwards again to avoid the mistakes of OCL 2.

References

[CKM+99] Steve Cook, Anneke Kleppe, Richard Mitchell, Bernhard Rumpe, Jos Warmer, and Alan Wills. *The Amsterdam Manifesto on OCL*. December 1999. http://www4.informatik.tu-muenchen.de/publ/papers/CKR+99.pdf.

[DGF17] Nisha Desai, Martin Gogolla, and Hilken Frank. Executing models by film-stripping: Enhancing validation by filmstrip templates and transformation alternatives. In *Workshop Executable Modeling, EXE 2017*, 2017.

[Dre] Dresden OCL Project. http://www.dresden-ocl.org/index.php/DresdenOCL.

[Ecla] Eclipse Acceleo Project. https://projects.eclipse.org/projects/modeling.m2t.acceleo.

[Eclb] Eclipse ATL Project. https://projects.eclipse.org/projects/modeling.mmt.atl.

[Eclc] Eclipse Epsilon Project. https://projects.eclipse.org/projects/modeling.epsilon.

[Ecld] Eclipse OCL Project. https://projects.eclipse.org/projects/modeling.mdt.ocl.

[Ecle] Eclipse QVT Declarative Project. https://projects.eclipse.org/projects/modeling.mmt.qvtd.

[Eclf] Eclipse QVT Operational Mappings Project. https://projects.eclipse.org/projects/modeling.mmt.qvto.

[Eclg] Eclipse Xtext Project. https://projects.eclipse.org/projects/modeling.tmf.xtext.

[Her17] Adolfo Sanchez-Barbudo Herrera. Auto-tooling to Bridge the Concrete and Abstract Syntax of Complex Textual Modeling Languages. EngD thesis. University of York, 2017.

[Obj97a] Object Management Group. Object Constraint Language Specification, Version 1.1, August 1997. https://www.omg.org/cgi-bin/doc?ad/97-08-08.pdf.

[Obj97b] Object Management Group. OMG Unified Modeling Language Specification, Version 1.1, August 1997. https://www.omg.org/members/cgi-bin/doc?ad/97-08-11.zip.

[Obj00a] Object Management Group. Request For Proposal UML 2.0 OCL RFP, September 2000. https://www.omg.org/members/cgi-bin/doc?ad/00-09-03.pdf.

[Obj00b] Object Management Group. Request For Proposal UML 2.0 Superstructure RFP, September 2000. https://www.omg.org/members/cgi-bin/doc?ad/00-09-02.pdf.

[Obj02] Object Management Group. Unambiguous UML (2U) 2nd Revised Submission to UML 2 Superstructure RFP, December 2002. https://www.omg.org/members/cgi-bin/doc?ad/02-12-23.pdf.

[Obj03a] Object Management Group. 2nd revised submission to OMG RFP ad/00-09-02: Unified Modeling Language: Superstructure version 2.0, January 2003. https://www.omg.org/members/cgi-bin/doc?ad/03-01-02.pdf.

[Obj03b] Object Management Group. OMG Unified Modeling Language Specification, Version 1.5, March 2003. https://www.omg.org/spec/UML/1.5/PDF.

[Obj03c] Object Management Group. Response to the UML 2.0 OCL RfP (ad/2000-09-03), January 2003. https://www.omg.org/cgi-bin/doc?ad/03-01-07.pdf.

[Obj03d] Object Management Group. Unified Modeling Language (UML) Specification: Infrastructure, Version 2.0, September 2003. https://www.omg.org/members/cgi-bin/doc?ptc/03-09-15.

[Obj06] Object Management Group. Object Constraint Language Specification, Version 2.0, May 2006. https://www.omg.org/spec/OCL/2.0/PDF.

[Obj08] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0, April 2008. https://www.omg.org/spec/QVT/1.0/PDF.

[Obj10] Object Management Group. Object Constraint Language Specification, Version 2.2, February 2010. https://www.omg.org/spec/OCL/2.2/PDF.

[Obj12] Object Management Group. Object Constraint Language Specification, Version 2.3.1, January 2012. https://www.omg.org/spec/OCL/2.3.1/PDF.

[Obj14] Object Management Group. Object Constraint Language (OCL) 2.5 Request For Proposal, February 2014. https://www.omg.org/cgi-bin/doc?ad/14-03-05.pdf.

[Obj15] Object Management Group. OMG Unified Modeling Language Specification, Version 2.5, March 2015. https://www.omg.org/spec/UML/2.5.

[Ric02] Mark Richters. A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis. Universitaet Bremen. Logos Verlag, Berlin, BISS Monographs, No. 14, 2002.

[Ste75] Stephen Johnson. Yacc: Yet another compiler-compiler, 1975.

[USE] USE, The UML-based Specification Environment. http://useocl.sourceforge.net/w/index.php/Main_Page.

[Wil03] Edward Willink. UMLX : A Graphical Transformation Language for MDA. In *Model Driven Architecture: Foundations and Applications, MDAFA 2003*, Twente, June 2003. http://eclipse.org/gmt/umlx/doc/MDAFA2003-4/MDAFA2003-4.pdf.

[Wil16] Edward Willink. The Importance of Opposites. In *16th International Workshop on OCL and Textual Modeling (OCL 2016)*, Saint-Malo, October 2016. http://www.eclipse.org/modeling/mdt/ocl/docs/publications/OCL2016Opposites/Opposites.pdf.

[WTW10] Claas Wilke, Michael Thiele, and Christian Wende. Extending variability for OCL interpretation. In *OCL 2010: Workshop on OCL and Textual Modelling*, Models 2010, Oslo, 2010.

About the author

Edward D. Willink is the chair of QVT and OCL specification Revision Task Forces at the Object Management Group and project leader for QVTd and OCL at the Eclipse Foundation. Contact him at `ed_at_willink.me.uk`.