JOT
JOURNAL OF
OBJECT TECHNOLOGY

Reflections on OCL 2

Edward D. Willink
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 (Object Management Group 1997b) suite of documents therefore includes a document defining the OCL 1.1 (Object Management Group 1997a) 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.

Unfortunately the prevailing state of the OCL specification is very unsatisfactory leading to incompatibility between implementations and difficulty in creating new ones. We will argue that we need to make the credible not only to support the full potential of OCL but also to support lightweight research activities.

In Section 2 we relate the history of the OCL 1 to OCL 2 transition before examining the problems with OCL 2. Then in Section 3 we examine these problems that variously emanate from the language, usability, the specification or typical tooling and identify solutions to the problems. Some evolution beyond the specification is described in Section 4 culminating in some suggestions in Section 5 as to how to specify a much simpler OCL 3. Finally in Section 6 we conclude.

JOT reference format:
Edward D. Willink. *Reflections on OCL 2.* Journal of Object Technology. Vol. 19, No. 3, 2020.
http://dx.doi.org/10.5381/jot.2020.19.3.a17

2. OCL 1 to OCL 2 History

The software community was not satisfied to leave UML 1 (and OCL 1) as a useful flexible semi-formal facility for communicating analysis considerations. Rather a near-formal semantics for UML 2 (and OCL 2) 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 (Richters 2002) was adapted to provide the formal Annex that accompanies all the OCL 2.x specifications. It 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 (Object Management Group 2003b), 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 (Object Management Group 2006) do not exist. However some do.

OCL 1 uses 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 are provided about how pseudo-operations and pseudo-attributes are modeled by UML's closed classes.

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.

OCL 1 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 specifies that all collections are always flattened.

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

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

OCL 1 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 (Object Management Group 2002),(Object Management Group 2003a) 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 (Object Management Group 2000b) and treated as a new self-standing specification with its own RFP (Object Management Group 2000a). 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 (Object Management Group 2003c).

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 (Object Management Group 2008) 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 (Object Management Group 2006). It is unclear how this could have happened since the draft clearly lacked the prototyping required by OMG, and contains many TBDs (To Be Decided) once UML 2.0 was finalized. UML 2.0 (Object Management Group 2003d) 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 (Object Management Group 2012) 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 (Object Management Group 2010) 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 the grammar to CS and the 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 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 based on UML's closed classes 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 (Desai et al. 2017)) by Martin Gogolla's team at Bremen[1] 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 criticizing 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 could 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 for implementers whose trust in the OMG specification was misplaced.

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 (Willink 2003) using the then planned QVTr language. This led to participation in the Eclipse support for QVTr and interaction with the Eclipse OCL project (Eclipse OCL Project 2020) to make it extensible for use by the Eclipse QVTd project (Eclipse QVT Declarative Project 2020). 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 (Object Management Group 2014) 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. Problems

The problems with OCL take many forms. In the following subsections we categorize them to distinguish those that directly affect users and those which only affect toolsmiths struggling to make sense of the specification.

3.1. 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.

3.1.1. 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 three 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 crashes and so the execution launcher will attempt to provide as much helpful diagnosis of the crash 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 a `RuntimeException` or an `Error` for catastrophic failures and by using `Exception` for recoverable failures. `Exceptions` form part of a Java function signature and so there can be some

[1] http://useocl.sourceforge.net/w/index.php/Main_Page

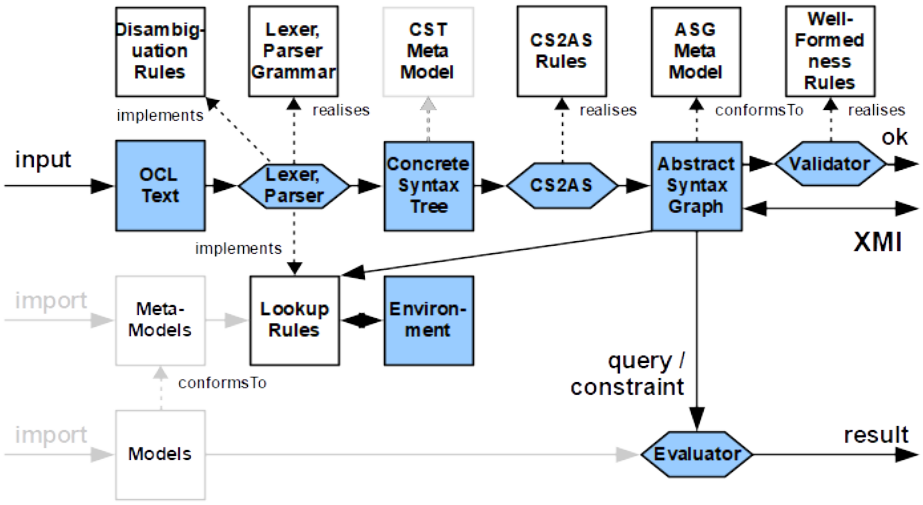

Figure 1 OCL 2 Architecture.

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 is 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 pointless restriction and provide a rich `invalid` that propagates diagnostics while preserving OCL semantics by ensuring that the diverse `invalids` behave as one.

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

Resolution Elevating a crash to a strict behavior is a straightforward change hampered only by a perceived resistance to expanding Booleans to 5-values: {`true`, `false`, `null`, `invalid`, `crash`}. It should be recognized that these five values only occur for malfunctioning programs. See Section 4.2.1 for work on proving that these malfunctions cannot occur and so proving that the programmed Boolean computation is indeed internally 2-valued.

3.1.2. Import, Extensibility, Modularization OCL is an unusual language in that by itself it is almost completely useless. It only becomes useful once embedded in an environment that provides models upon which OCL computations can operate.

The only partially specified practical usage of OCL is the Complete OCL extension that enables an OCL document to complement some pre-existing metamodel with

- constraints for UML classes
- pre/post-conditions/bodies for UML Operations
- initializers for UML Properties
- guards for UML Transitions

Clearly the Complete OCL document must be tightly coupled to some UML model, but unfortunately there is no 'import' declaration. This forces all tools to invent a proprietary solution.

Beyond the OCL specification, OCL is used as the basis for many model transformation languages such as QVT that provide the requisite models for the OCL queries and provide a disciplined mechanism by which the results of side effect free OCL queries provoke mutation of the models.

Unfortunately, the historic support for UML leaves OCL bloated with Message functionality that few tools support and State functionality that few users need. For pure OCL usage, this bloat can just be ignored, but for extended applications such as QVT this bloat is an embarrassment that should be removable. Conversely there are research areas such as Temporal specification for which researchers should be able to experiment with alternative temporal expressions.

Resolution The Pivot-based Eclipse OCL prototype adds an 'import' declaration for use by OCL or QVT or ... The statement can be used to declare not only the metamodel(s) which the OCL complements but an optionally extended Standard Library that supplies re-usable functionality.

The models provided by the overall OCL specification should be structured in cross-cutting modules so that a tool may select a subset of the standard modules and augment with its own mod-

ules, thereby excluding unwanted Message and Tuple support while adding experimental Time support.

3.1.3. Flattened Collections

OCL 1 collections provide support for UML's multi-valued properties and since UML has no support for nested multi-values, OCL 1 collections were specified to avoid nesting by flattening at every opportunity. This mistake was partially recognized when OCL 2 introduced the `collectNested` iteration and the `flatten` operation. However this leads to a confusion requiring special efforts to avoid type corruption.

Resolution All OCL collection operations should be type consistent so that a type conversion only occurs when the `flatten` operation is used explicitly.

3.1.4. Collection conforms to OclAny

The formalization in the OCL 2 Annex clearly states that the formalization applies to a type system in which Collections and Objects are distinct. It does not prohibit a unification, but observes that extra formalization effort is needed by a unification.

The OCL 2.2 specification changed to allow a `Collection` to conform to `OclAny` without putting in that extra effort or indeed even providing a use case to motivate the change.

Two possible use cases involve either defining a utility operation such as `printf` that can receive a polymorphic mix of collections and objects or to provide a stronger type than `Collection(OclAny)` for a heterogeneous nested collection. The latter challenge is perhaps where work in needed in the Annex.

Resolution The conformance of `Collection` to `OclAny` does not seem to offer significant benefits, so it may be best to revert this change until the formalization work is done to understand its consequences.

3.2. Perceived Problem

Not all problems are really problems at all.

3.2.1. 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.[2]

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. Exactly the same programming approach is possible with OCL provided the programmer ensures that the particular OCL tooling API 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 that returns all three outcomes as OCL values. This alternative is useful when the programmer really wants to exploit all three outcomes. An inappropriate choice of API is the probable source of unhappiness when only two outcomes are expected.

[2] The extra idempotent null output in OCL 2.4 is not relevant here.

3.3. Usability Problems

Some aspects of the OCL language, even when adequately specified, have been found to cause undue difficulties for OCL users.

3.3.1. 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 short-forms 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 short-form The introduction of the explicit `oclAsSet` library operation formalized the short-form

Resolution The EOL variant of OCL used by the Epsilon (Eclipse Epsilon Project 2020) transformation languages demonstrates that it is possible to make do with just a dot operator for both object and collection 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 short-form

```
myAggregate*.name          -- short-form explicit collect
```

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

3.3.2. 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.

```
aCol->isUnique(name)    -- aCol->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.

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

3.3.3. OCL Re-Use Cases

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?

We first examine a couple of use cases that OCL could respond to and then look at how OCL could make them much easier.

Novel Application re-using 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 (Wilke et al. 2010) 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 normalized metamodel is unaffected. It is just necessary to convert the user's new representation to the normalized representation. Dresden OCL (Dresden OCL Project 2020) 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 `Stereotypes` and `AssociationClasses`.

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

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.

Resolution The OCL specification must be demonstrably clear, re-usable and structured to encourage an exemplary architecture using a Pivot-based metamodel.

3.4. 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 oldest and best OCL implementations, USE and Eclipse OCL, are seriously incompatible and address very different use cases. USE has sensibly stuck with OCL 1 cherry-picking only a little OCL 2 functionality. The Classic Eclipse OCL developers adhered almost mindlessly to the letter of the OCL 2 specification. It is only with the Pivot-based Eclipse OCL that the inadequacies of OCL 2 start to be addressed.

3.4.1. Obsolete terminology, inadequate exposition

The truncated development process for the OCL 2 specification left it with many editorial problems, such as the continuing OCL 1 usage of 'property' to refer generically to UML 1's `AssociationEnd`, `Attribute`, `Method` or `Operation`. Unfortunately when UML 2 unified `AssociationEnd` and `Attribute` as `Property`, the failure of OCL 2 to track makes for an ambiguity; does a 'property' refer to just a `Property` or to an `Operation` too? Following UML, 'feature' is the correct term for a `Property` or `Operation`.

The lack of an OCL 2 prototype to demonstrate the many new specified capabilities means that many of them are not realizable as specified and the grammars and models that underpin them do not exist.

Resolution The inadequate specification can be remedied once a prototype has been evaluated. The missing grammars and models should be provided by that prototype.

The Pivot-based variant of the Eclipse OCL project has spent the last ten years attempting to prototype to satisfy this goal, This paper is in part a report on the successes and failures of that prototype.

The OCL specification makes extensive use of class names as part of its exposition, but each is an independently typed best endeavor. Unfortunately the `VariableDeclaration` class, which abstracts the commonality of `Parameter` and `Variable`, is missing. Auto-generation of large parts of the specification from the models, as has been done for UML 2.5, should avoid such oversights and also avoid numerous cases of failure to track refactorings.

3.4.2. UML alignment / EMOF support

The liberation of OCL 2 from the shackles of UML 1 should have provoked some consideration of what the 'aligned with UML' statement in the OCL specification actually means. For UML 1, it seems obvious that an OCL 1 tool must re-use classes such UML::Class and UML::Association and UML::Constraint. But OCL 1 had no metamodels so this was never specified.

For UML 2, it is specified that OCL may also be used with EMOF even though there is no corresponding Association or Constraint class. Is an OCL 2 tool expected to introduce an OCL::Constraint for use with EMOF but use a UML::Constraint when working with UML? Does an OCL tool really have to struggle with the complexities of UML Associations and association/class-owned Properties when EMOF (and Ecore) only have the equivalent of class-owned Properties?

Resolution It is appropriate to step back and see what is appropriate for an executable specification, as defined by OCL, rather than an analysis specification, uncomfortably bloated to a design specification, as provided by UML.

It should be noted that since OCL code is really executed, an OCL tool encounters all the difficulties that this entails. In practice UML models require significant and often very stylized conversions to make their models executable and so the many deficiencies of UML modeling are remedied by this stylized tooling.

The enduring success of OCL despite its limitations is probably due to its fundamental simplicity. The state of a system can be represented as a graph comprising classifier-typed nodes and feature-typed edges.

The classifier-typed nodes may be DataType-typed-values whose value may be used directly or Class-typed objects that provide transitive access to values.

The feature-typed edges may be simple Properties accessing values, often known as attributes, references to other objects, operation calls, iteration calls, references to stereotypes, access to static properties, access to stereotype properties, ... Each modeling concern may introduce a new flavor of feature.

Evaluation of an OCL query starts at some privileged node identified as `self` and then traverses edges to gather whatever values are required by the query.

Figure 2 shows a system comprising some arbitrary instances `p`, `q`, ... with correspondingly arbitrary types `P`, `Q`, ... 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 node to node in a regular way even though the expression involves collection, operation and datatype complexities.

To keep OCL simple, we want to exploit this fundamental simplicity by providing a metamodel that suits OCL rather than contorting to accommodate the eccentricities of UML. The problem of OCL::Constraint or UML::Constraint is solved. There is always an OCL::Constraint, and an OCL::Class and an OCL::Feature that is the basis for an OCL::Property and OCL::Operation and OCL::Iteration ... OCL has its own metamodel that is designed to suit OCL free from the limitations of UML or EMOF or Ecore or XSD.

In order for OCL to co-exist with another technology such as UML, it is obviously necessary to perform a transformation of the user's UML metamodel into the equivalent OCL metamodel, for which the term Pivot was coined by the Dresden OCL team and endorsed by the Pivot-based Eclipse OCL prototype.

The need for this transformation may seem like a burden, but it proves to be a major simplification. Without such a transformation, support for OCL for N different technologies requires N variants of the OCL tooling each adjusted to the eccentrici-

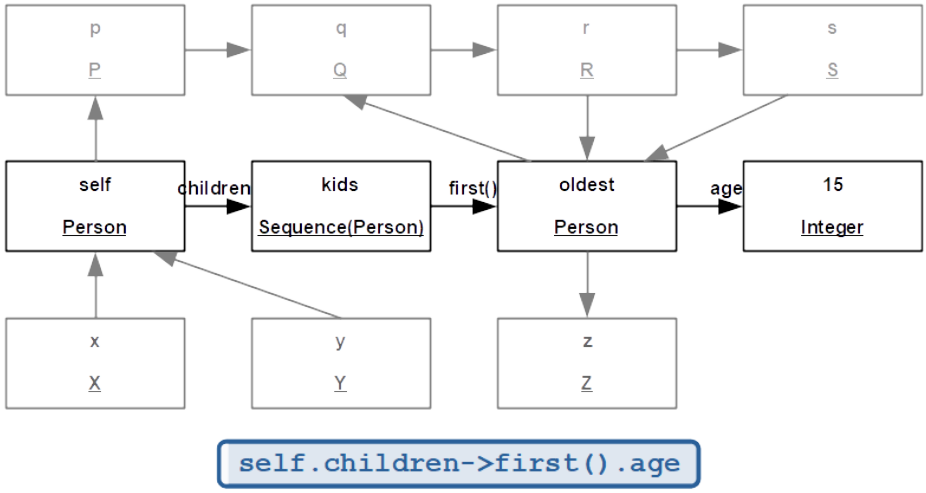

Figure 2 OCL Navigation.

ties of that technology. In contrast a single variant of the Pivot OCL tooling is possible supported by N transformations each of which normalizes the eccentricities of its technology to the Pivot OCL formulation. UML stereotypes no longer pollute the whole of OCL, rather a Stereotype is converted to a regular OCL classifier with regular OCL features.

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. 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 (Willink 2016). 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 `Classes` and a single `Property` instance to define the unidirectional link from a `Class` to a `DataType` value.

3.4.3. XMI The OCL 2 specification calls for model interchange between OCL tools using XMI but provides no clues as to how this is achieved. There appears to be a naive assumption that if there is an OCL metamodel, XMI will just work. Not so.

XMI elaborates XML's ability to serialize a tree-structure by introducing the `xmi:id` element and corresponding references to support the graph-structures formed by cross-references in a model. But a reference requires the `xmi:id` to exist. Unfortunately, when an OCL expression traverses an unnavigable opposite, the serialization of the `PropertyCallExp` needs to reference a property that does not exist in EMOF (or Ecore) and which exploits the unusual association-containment in UML.

The ability to extend classes with additional operations and properties poses a further challenge for XMI serialization, since the additional features are not available to be referenced as part of the original class.

More fundamentally, when a type, corresponding to for instance `Set(String)`, is referenced that reference must resolve to the same definition as a similar reference from elsewhere. No overall pool of shared definitions is specified. In UML, template instantiations are performed on demand so that there may be both `My::Set(String)` and `Your::Set(String)` without any clarity as to whether they are distinct types or not.

Resolution 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. Efficient tooling for OCL-based languages such as MOFM2T or QVT will however suffer if XMI support for OCL is not possible.

The problem with serialization of the non-existent unnavigable opposite may be resolved by introducing an `OppositePropertyCallExp` class to the metamodel whose reference is to the navigable property in the other direction. Internally the normalization to the Pivot OCL metamodel can ensure that all such opposites do exist. Similar opposite modeling extensions are needed in QVT.

The Pivot-based Eclipse OCL solves the problem of locating additional features in closed classes by introducing the concept of a Complete Class that is an overlay of same-hierarchically-named closed classes. This supports open classes by overlaying as many closed classes as required. Unfortunately it has the unpleasant corollary for the tooling that accesses to Classes must redirect to their Complete Classes to ensure that any additions from sibling Classes are not overlooked.

The Pivot-based Eclipse OCL solves the problem of unique definitions by 'clarifying' UML semantics to specify that the namespace hierarchy of collection, lambda, primitive, tuple types and template specializations are to be ignored. `My::Set(My::String)` and `Your::Set(Your::String)` are therefore the same type. Saving a model to XMI relocates all

the shared definitions by adding an `$orphanage$` package to host e.g. `$orphanage$::Set(String)`. Loading from XMI relocates the contents of the many incoming `$orphanage$` packages to a single global internal `$orphanage$` package. Each distinct type is modeled by a distinct singleton in the Pivot representation.

3.5. Tooling Problems

Some aspects of the OCL language / specification cause difficulties for tooling that far exceed any convenience they may afford to users.

3.5.1. Coherent Grammar

Just about the first thing any toolsmith will look for when tooling a language is its grammar.

OCL 1.1 Grammar In the early days of language tooling, it was understandable that languages could have bad properties such as the notorious dangling-else ambiguity which all C/-Java programmers learn about the hard way. However that all changed with the advent of LALR support tools such as yacc (Stephen Johnson 1975) which diagnose parsing problems as well as generating an efficient parser.

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 shift/reduce conflict for the expression/declaration ambiguity that we consider in Section 3.5.2.

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. It is no longer possible to determine how the ambiguity was resolved.

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 it was biased in favor of the left recursion typical of LALR tools and against LL tools such as Xtext (Eclipse Xtext Project 2020).

Resolution Obviously there should be a grammar that accurately specifies the language and which is demonstrably compatible with standard tools.

3.5.2. Expression/Declaration syntax ambiguity

The OCL syntax is primarily an expression language that elaborates external declarations with constraints or bodies; the declarations come from outside.

However OCL may provide a local declaration using the distinctive `let...in...` syntax or embedded with an iterator call: `isUnique(n | n)`.

The `let...in...` syntax has an irritating dangling-in difficulty for an LALR grammar parsing `let...let...in...in...`. This can be solved by duplicating the expression grammar to ensure that the recursion takes precedence.

The iteration syntax poses a potentially infinite look-ahead challenge to distinguish the first and second `n`'s in `isUnique(n | n)`. The first (declaration) `n` may be elaborated as multiple iterators with non-trivial type declarations. The second (expression) `n` may be elaborated with arithmetic operations and recursive iterations.

The look-ahead must traverse the entire declaration to detect the `|` separator to distinguish the explicit iterator `isUnique(n | n)` from the implicit iterator short-form `isUnique(n)`.

Resolution The bad grammar should have been detected by a yacc-like tool. The ambiguous declaration syntax should have been redesigned before it was released to the world. Now we have to introduce a probably breaking change to cure it.

The easiest solution is to use a distinctive 'here-comes-a-declaration' keyword. To some extent `let` already does this.

One possibility is therefore to require the long form `isUnique(n | n)` to be rewritten as `isUnique(let n in n)` so that a `let` without an initializer defines an iterator and a `let` with an initializer is the conventional local let variable.

Another possibility is to introduce a new `var` keyword supporting a prefix declaration as in `isUnique(var n; n)`. This ability to prefix a declaration to an OCL expression could render the let-expression redundant by rewriting

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

as

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

The above are two suggestions for eliminating the declaration/expression ambiguity. There are no doubt others.

Any change should support the pattern matching evolution discussed at Aachen (Brucker et al. 2013) and in Section 4.2.3.

3.5.3. Short-circuit operators

In many C-based programming languages short-circuit Boolean and-or operators support a guard idiom to ensure 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 or a multi-processor implementation may therefore evaluate the two commutative terms in arbitrary order and so be unable to avoid the crash. so the implementation 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 (Cook et al. 1999) 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 '$\perp$' 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 '$\perp$' 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 '$\perp$' for an Input value but as an idempotent 'ϵ' or '$\perp$' 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.

Resolution 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 mathematical elegance of commutative and-or operations provide any practical benefits; they certainly provide considerable implementation difficulties. Specify that evaluation of the first term must guard the second.

4. Evolution

This section summarizes successful and planned work that goes beyond the narrow limitations of the specification before drawing on experience of what has failed or is at least unduly complex to make some radical suggestions as to how an OCL 3 might be specified more usefully.

4.1. Prototyped Evolution

We first identify functionality that has been successfully prototyped by Eclipse OCL.

4.1.1. Map library type

The Map type is familiar to OO programmers but sadly missing from OCL.

Resolution Implemented (Willink 2019). Provision of Map literals and joint key/value iterators is useful.

4.1.2. Safe navigation

The side effect-free characteristics of OCL make an OCL program much easier to analyze than many other languages. However many constraints suffer from problems with null navigation.

Resolution The prototype exploits the `[1]` and `[?]` multiplicities of UML to distinguish not-null from maybe-null and so identify statically which expressions have a null-hazard. A `?.` safe navigation operator is introduced. The first implementation of this was almost useless for collections and so some elaborations that include specifying distinct collection and element multiplicities as in `Set(String)[*|1]` were required (Willink 2015b).

4.1.3. Type construction / shadow types

Users find it inconvenient that an OCL expression cannot create an instance of a type. The OCL specification also finds this inconvenient and introduces `make` operations. At first sight this is an insoluble problem because creation of a new instance creates a side effect in the memory system and OCL is side effect free.

Resolution The prototype introduces the concept of shadow objects (Willink 2018) in which the 'created' object is a shared immutable re-use from a notionally infinite pre-existing pool of all possible instances. There is no side effect even though a practical implementation no doubt populates the infinite pool lazily. Once extended to a model transformation environment supporting mutations, assignment of a container to a shadow object must clone to preserve the immutability.

4.1.4. Templates, Lambdas and Reflection

The OCL specification uses many facilities without properly specifying them.

The collections are specified using a magic `T` that has more akin to a textual macro than a type argument. Clearly the `T` in `Set(T)` is a template parameter. If it isn't, then how does it relate to the missing support for UML template parameters?

The body of an OCL iteration call is an expression to be evaluated for each iteration. This is clearly an anonymous function definition well known to the computer science community as a lambda expression. The contorted specification of the `closure` iteration using the language of textual macro substitution is clear evidence that lambda expressions are in use.

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 flatten() constraints.

```
post: result =
  if self.oclType().elementType.oclIsKindOf(CollectionType)
  then ...
```

Resolution An implementation that fully supports templates, lambdas and reflection avoids the need for clumsy workarounds to provide the half-baked functionality that fails to satisfy user expectations. Once lambda expressions exist, lambda types and so lambda variables follow, allowing function bodies to be passed arbitrarily rather than just as the special case of an iterator body.

4.2. Pending Evolution

Some problems have been tackled but not completed.

4.2.1. Fail-safe execution

Safe navigation, as described in 4.1.2, prevents the most common OCL execution failures but not all. Divide-by-zero is very rare in OCL, but index-out-of-bounds with ordered collections is far from rare, particularly for users who accidentally use the 0-based indexing of implementation languages rather than the 1-based specification policy. With these problems detected, an OCL expression can be guaranteed not to require the complexities of `null` or `invalid`; a Boolean expression is provably two-valued.

Resolution This is work in progress to support code generation for QVTr where run-time handling failures is very undesirable; a failure during a transformation is a failure and obscure when it happens. A compile-time check that proves that a failure is impossible is more useful for the user and allows for simpler and faster generated code.

A little integer value flow analysis is needed to prove that e.g. `x->at(x->size()-1)` is safe only if `x` is provably not empty.

A form of assertion will be needed for expressions that are too complicated for practical symbolic analysis.

4.2.2. 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.

Resolution Executable UML (Mellor & Balcer 2002) provided a plausible mechanism to model precision, but it was not adopted by UML 2. It did however influence the OCLforUML profile (Willink 2015a) that users apply to their UML models.

The precision aspects of the profile are mostly a documentation convenience. The profile was developed primarily to identify null-free Collections in support of the safe navigation work. See Section 4.1.2.

4.2.3. Patterns

At the Aachen (Brucker et al. 2013) workshop, an OCL enhancement to support pattern matching was discussed. At its most trivial, this would allow the verbose idiomatic test-and-cast usage

```
if x.oclIsKindOf(CastX)
then x.oclAsType(CastX).doSomething()
else null
endif
```

which requires two library calls and two specifications of the `CastX` type to be simplified as a pattern-match-guarded action:

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

The success, or failure of the match of the value of `x` expression against the `castX : CastX` pattern determines the direction taken by the `if`. Within the `then` branch, the successfully bound `castX` variable can be used with its matched `CastX` type.

Resolution Some work has been done on this, but it foundered on the Expression/Declaration syntax ambiguity discussed in Section 3.5.2. Elaborating the declaration syntax was too hard. Introduction of the `var` keyword could unblock this impediment.

The work also identified that the QVTr template syntax was probably suitable for re-use as a pattern syntax and that conversely introducing a different syntax for OCL would be very damaging to the QVTr grammar as a whole.

4.2.4. 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.

Resolution The OCL and QVTr Standard Libraries are implemented by models for the Pivot-based Eclipse OCL and QVTd, however an attempt to provide a tutorial demonstrating a maths library proved much harder than expected. Allowing a user to write `atan2(x, y)` challenges the support for static operations whereas `Maths::atan2(x, y)` is familiar to Java users but hardly worthy of a specification language.

5. An OCL 3 tooling Proposal

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 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.

5.1. The big problems

A solution to many of the problems has been suggested above and successfully prototyped as part of the Pivot-based Eclipse OCL. However there are two areas where the prototype solution is unpleasant; the prototype has tried too hard to maximize conformance to a perhaps over-enthusiastic interpretation of the letter of the specification.

5.1.1. Open classes

The ability to add attributes and operations to open classes is difficult to support when their AS representation is to be just like similar attributes and operations defined in a model based on closed classes.

Many modeling capabilities such as Acceleo (Eclipse Acceleo Project 2020), ATL (Eclipse ATL Project 2020) and QVTo (Eclipse QVT Operational Mappings Project 2020) 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 (Object Management Group 2015) 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. Once the need for fully open classes goes, the inconvenience of the prototype's Complete Class overlay of closed classes in the prototype can be resolved; externally the user's metamodel may use a closed class semantics. Internally the normalized Pivot model may open the class to inject the additional features.

5.1.2. Two metamodels

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 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.

The two metamodels are mostly isomorphic with AS classes such as `IfExp` with condition/then/else children replicated by a `IfExpCS` CS class with three equivalent children. Many of the differences arise from the need for the CS to accommodate ambiguities until there is sufficient context to select the unambiguous AS variant. Other difficulties arise from the way in which references to elements such as types are modeled.

Exactly the same problems occur in the QVT specifications and so I sponsored Adolfo Sanchez-Barbudo Herrera's EngD (Sanchez-Barbudo Herrera 2017) 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.

We can now challenge the presumption that two metamodels are necessary. Once we follow the resolution of Section 3.5.2 to revise the grammar and eliminate ambiguities we are left with just the element referencing issue, which we can resolve by trimming our two metamodels not quite to one metamodel, but to one and a bit.

5.2. Modeling a Type Reference

The ability to make one and a bit metamodels work relies on a seemingly trivial aspect of the metamodeling.

In the concrete syntax of many languages the letter-sequence for a built-in type such as `String` may occur many many times. This does not mean that there are many copies of the built-in type, rather that there are many descriptions of the built-in type. It is the responsibility of the tooling to ensure that only one such type does exist so that each reflective usage of the type shares a single definition.

Similarly the letter-sequence for a synthesized type such as `Set<String>` may occur many times. Once again, it is the responsibility of the tooling to ensure that only one underlying type definition exists.

When UML models a type, it uses `TypedElement` as the abstraction of all model elements that have types, `Type` as the

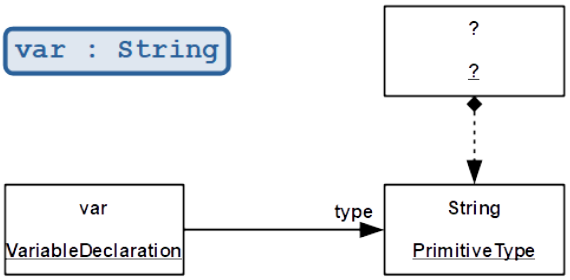

Figure 3 OCL 2 Type Reference.

abstraction of all types and the `TypedElement::type` property as the mechanism by which the `TypedElement` references its `Type`. Fig 3 shows how the OCL metamodel uses this property to associate `String` as the primitive type of a `VariableDeclaration`. Since the `String` primitive type may be used by many definitions, it cannot be contained by the `VariableDeclaration` rather it must be contained elsewhere. For a built-in type we can ignore the problem as a bit of implementation magic.

However once we model use of a `Set<String>` this approach does not work since not all possible template specializations can be built-in, rather a shared instance of `Set<String>` must be modeled. The UML metamodel provides two solutions to this problem. For types, the user is required to create an explicit specialization somewhere so that it can be referenced as many times as necessary. This approach is fine for an informal specification but fails when my subsystem has one definition of `Set<String>` and your subsystem has another. For non-types, where creation of an instance elsewhere is too inconvenient, UML has another idiom. There may be two alternative references such as the non-containment `TemplateParameterSubstitution::actual` and the sibling containment `TemplateParameterSubstitution::ownedActual` giving the modeler the freedom to use whichever is convenient and the user the inconvenience of a derived property to select the appropriate property.

The UML approach is 'right' in so far as it attempts to force the singleton existence of type definitions, but 'wrong' in that the enforcement is incomplete and inappropriate for a specification language. A unique definition is only a necessity for an implementation. A specification language may refer many times to the unique type.

In contrast if we examine the modeling for the same problem in Ecore, we find that `ETypedElement::eType` is very similar to UML's `TypedElement::type`. However when Ecore evolved to support Java generics, an `EGenericType` rag-bag was added. Each `EGenericType` is a descriptor for a required type contained by the new `ETypedElement::eGenericType` property. Ecore, although practically an implementation language, evolved to support type descriptions. The OCL metamodel, emulating UML, is cursed with UML's inappropriate modeling of type singletons.

Recognizing the different ways in which a type reference can be modeled, we can see that the 'CS' metamodel must model the multiple descriptions of a type whereas the 'AS' must model the single definition of that type. We do not however need two metamodels if our one and a bit metamodels provides a pair of properties; a primary type description property for use by consumers of the 'CS' metamodel such as parsers, and a secondary derived type definition property for the 'AS' consumers such as code generators and validators.

We are left with a design decision to make in regard to interchange using XMI. If the interchange exposes the 'AS' content, we can interchange fully resolved definitions but have difficulties ensuring that the two parties to the interchange are able to share globally unique definitions. If instead the interchange is limited to the 'CS' content only, type descriptions are shared and each party is free to use its own policies to resolve the descriptions to its own locally unique definitions.

Since the 'CS' content is exchanged with other tools we will refer to this as the 'external' perspective of our one and a bit metamodels, and to the private 'AS' content as 'internal'

Fig 4 shows the 'external' and 'internal' modeling for our `Set<String>` example. On the bottom row we show the external type description in which the `VariableDeclaration` owns a description comprising a `TemplatedTypeDesc` named `Set` which in turn owns a `SimpleTypeDesc` named `String`. On the middle row we show the internal type definitions comprising the singleton `PrimitiveType` named `String` that is referenced by the singleton `SetType` named `SetOfString`. The derived `resolvedType` properties of the many external perspective descriptions reference the singletons of the internal perspective.

The package and model ownership of the singletons is omitted in the interests of clarity. Transitively a single `MetamodelManager` is the container of all singleton definitions.

5.3. Overall OCL 3 Architecture

Having solved the problem of two metamodels, we can now see how the various components of the Overall architecture shown in Fig 5 collaborate to provide a satisfactory solution to many of the problems in Fig 1.

MetamodelManager At the heart of the architecture is the MetamodelManager that is responsible for the normalized singleton Pivot representations within the Pivot Metamodels.

When an 'import' is required, the MetamodelManager locates the referenced user metamodel, and activates the appropriate Loader to convert it to the normalized Pivot form ensuring that any shared declarations use the appropriate singleton definitions. As part of the loading process, any esoteric user elements such as UML's Stereotype or AssocationClass are normalized to the simple classifier-feature internal representation.

When a 'lookup' is required by a Derivation Rule to resolve an element description to its unique element definition, the lookup is performed by querying the Metamodel Manager.

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

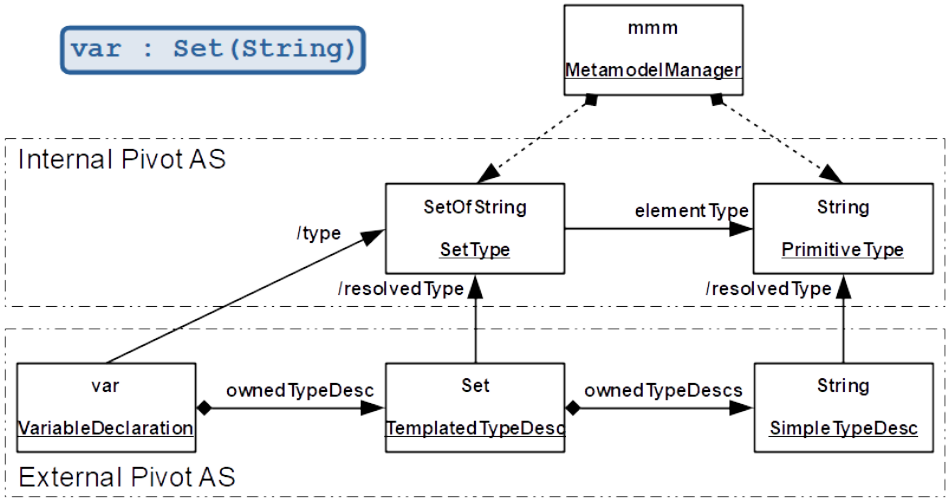

Figure 4 Possible OCL 3 Type Descriptor.

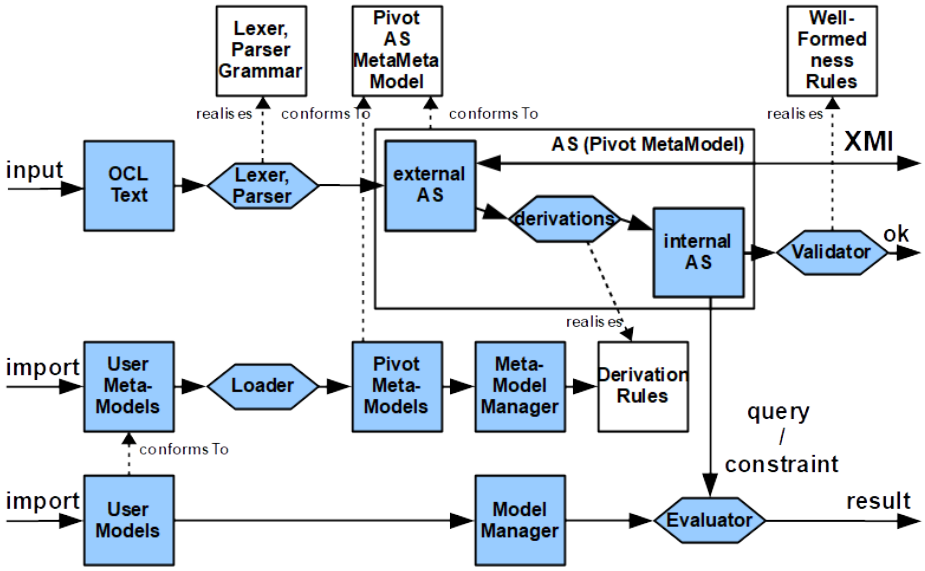

Figure 5 Possible simplified OCL 3 Architecture.

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. This is very inefficient and involves many mutations and so side effects.

Between imports, the Metamodel Manager is logically immutable lazily exposing only what is required from a nationally infinite pool of all possible definitions. A lookup is therefore much more easily resolved by an immutable search up the AS for what is required rather than a churning push down of everything that could be required.

In addition to the imported user metamodels, the Metamodel Manager supervises the loading of the standard Pivot metamodel, if the user requires reflective functionality, and the OCL Standard Library. Each of these loads may be overridden to support use of a customized Pivot or Library.

Specification The transparent boxes show the functionality to be provided by grammars and models in the specification. In contrast to Fig 1 we see that the Disambiguation Rules have vanished since the grammar ambiguities are removed. This allows standard parser tooling to be used. The lookup Rules have also vanished, or rather migrated to the Derivation Rules that perform a simple query to convert the element description parsed from the source text to the element definition required by the internal AS.

There is just one (and a bit) metamodels.

The Well-formedness rules are substantially unchanged.

Basic Parsing The second row of Fig 5 shows the OCL text input model being parsed by standard lexer and parser tooling configured by the specified grammar. The resulting External perspective of the Pivot AS is suitable for interchange using XMI.

This basic parsing can potentially be performed without the aid of a MetmodelManager. This may be convenient for some novel OCL re-use cases, but in practice the services of the advanced tooling will be essential for any form of validation or completion assist in an editor.

Advanced Tooling Advanced tooling activates the MetamodelManager to covert all the references in the external AS to the unique definitions of the internal AS. This enables tools that exploit the OCL such as Validators or Evaluators to provide enhanced capabilities.

Model Manager Analogously to the use of the MetamodelManager to supervise all metamodels, the ModelManager supervises the models for use by the evaluator. In addition to the obvious task of loading required models, it also supports all the possible forms of feature access that may be used in those models. For the simplest form of property access this may just require that a value be obtained from a slot and converted to the appropriate Boolean/Integer/Real/String representation. In more complex cases the reified form of an applied stereotype may need to be navigated to obtain a slot value from the instance of the stereotype.

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 struggled to implement it as faithfully as possible.

It is a pity that it has taken so long to recognize 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.

This appears to leave the OCL community with a three-way choice:

- Do nothing. USE and Eclipse OCL and so OCL will fade away.
- Evolve existing OCL tool(s). Current teams are inadequate.
- Develop a new OCL. There is no team.

Any future progress surely depends on a credible specification. This requires significant work, which may be mitigated if some simplifications are made.

Eclipse OCL could evolve to prototype the simplifications, but only if that is what the community wants.

Acknowledgments

Many thanks to Horacio Hoyos Rodriguez and the anonymous reviewers for helpful comments on an inadequate initial submission.

References

Brucker, A., Chiorean, D., Clark, T., Demuth, B., Gogolla, M., Plotnikov, D., … Wolff, B. (2013). *Report on the Aachen OCL Meeting*. Retrieved from http://www4.informatik.tu-muenchen.de/publ/papers/CKR+99.pdf

Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J., & Wills, A. (1999). *The Amsterdam Manifesto on OCL*. Retrieved from http://www4.informatik.tu-muenchen.de/publ/papers/CKR+99.pdf

Desai, N., Gogolla, M., & Frank, H. (2017). Executing models by filmstripping: Enhancing validation by filmstrip templates and transformation alternatives. In *Workshop executable modeling, exe 2017*.

Dresden OCL Project. (2020). Retrieved from http://www.dresden-ocl.org/index.php/DresdenOCL

Eclipse Acceleo Project. (2020). Retrieved from https://projects.eclipse.org/projects/modeling.m2t.acceleo

Eclipse ATL Project. (2020). Retrieved from https://projects.eclipse.org/projects/modeling.mmt.atl

Eclipse Epsilon Project. (2020). Retrieved from https://projects.eclipse.org/projects/modeling.epsilon

Eclipse OCL Project. (2020). Retrieved from https://projects.eclipse.org/projects/modeling.mdt.ocl

Eclipse QVT Declarative Project. (2020). Retrieved from https://projects.eclipse.org/projects/\protect\penalty\z@modeling.mmt.qvtd

Eclipse QVT Operational Mappings Project. (2020). Retrieved from https://projects.eclipse.org/projects/modeling.mmt.qvto

Eclipse Xtext Project. (2020). Retrieved from https://projects.eclipse.org/projects/modeling.tmf.xtext

Mellor, S., & Balcer, M. (2002). *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley.

Object Management Group. (1997a, August). *Object Constraint Language Specification, Version 1.1*. Retrieved from https://www.omg.org/cgi-bin/doc?ad/97-08-08.pdf

Object Management Group. (1997b, August). *OMG Unified Modeling Language Specification, Version 1.1*. Retrieved from https://www.omg.org/members/cgi-bin/doc?ad/97-08-11.zip

Object Management Group. (2000a, September). *Request For Proposal UML 2.0 OCL RFP*. Retrieved from https://www.omg.org/members/cgi-bin/doc?ad/00-09-03.pdf

Object Management Group. (2000b, September). *Request For Proposal UML 2.0 Superstructure RFP*. Retrieved from https://www.omg.org/members/cgi-bin/doc?ad/00-09-02.pdf

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

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

Object Management Group. (2003b, March). *OMG Unified Modeling Language Specification, Version 1.5.* Retrieved from https://www.omg.org/spec/UML/1.5/PDF

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

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

Object Management Group. (2006, May). *Object Constraint Language Specification, Version 2.0.* Retrieved from https://www.omg.org/spec/OCL/2.0/PDF

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

Object Management Group. (2010, February). *Object Constraint Language Specification, Version 2.2.* Retrieved from https://www.omg.org/spec/OCL/2.2/PDF

Object Management Group. (2012, January). *Object Constraint Language Specification, Version 2.3.1.* Retrieved from https://www.omg.org/spec/OCL/2.3.1/PDF

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

Object Management Group. (2015, March). *OMG Unified Modeling Language Specification, Version 2.5.* Retrieved from https://www.omg.org/spec/UML/2.5

Richters, M. (2002). *A Precise Approach to Validating UML Models and OCL Constraints. PhD thesis. Universitaet Bremen. Logos Verlag, Berlin, BISS Monographs, No. 14.*

Sanchez-Barbudo Herrera, A. (2017). *Auto-tooling to Bridge the Concrete and Abstract Syntax of Complex Textual Modeling Languages. EngD thesis. University of York.*

Stephen Johnson. (1975). *Yacc: Yet another compiler-compiler.*

Wilke, C., Thiele, M., & Wende, C. (2010). Extending variability for OCL interpretation. In *Ocl 2010: Workshop on ocl and textual modelling.* Models 2010, Oslo.

Willink, E. (2003, June). UMLX : A Graphical Transformation Language for MDA. In *Model driven architecture: Foundations and applications, mdafa 2003.* Twente. Retrieved from http://eclipse.org/gmt/umlx/doc/MDAFA2003-4/MDAFA2003-4.pdf

Willink, E. (2015a, November). The OCLforUML Profile. In *Eclipsecon europe 2015.* Ludwigsburg.

Willink, E. (2015b, September). Safe Navigation in OCL. In *15th international workshop on ocl and textual modeling (ocl 2015).* Ottawa. Retrieved from http://www.eclipse.org/modeling/mdt/ocl/docs/publications/OCL2015SafeNavigation/SafeNavigation.pdf

Willink, E. (2016, October). The Importance of Opposites. In *16th international workshop on ocl and textual modeling (ocl 2016).* Saint-Malo. Retrieved from http://www.eclipse.org/modeling/mdt/ocl/docs/publications/OCL2016Opposites/Opposites.pdf

Willink, E. (2018, October). Shadow Objects. In *18th international workshop on ocl and textual modeling (ocl 2016).* Copenhagen.

Willink, E. (2019, September). An OCL Map Type. In *19th international workshop on ocl and textual modeling (ocl 2016).* Munich. Retrieved from http://www.eclipse.org/modeling/mdt/ocl/docs/publications/OCL2019MapType/OCLMapType.pdf

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.