# An extensible OCL Virtual Machine and Code Generator

E.D.Willink
Eclipse Modeling Project
ed@willink.me.uk

## ABSTRACT

The Object Constraint Language (OCL) is a specification language that is also executable and so a variety of OCL execution capabilities have evolved. Some are interpreted while others use a code generator for an implementation language such as Java. The mapping of much of OCL to Java is obvious and so many implementations pursue the obvious approach but then find that the approach can only support an OCL subset.

In this paper we revisit OCL evaluation. We first establish a simple uniform execution framework that applies to the whole of OCL. We call this an OCL Virtual Machine. We then identify how optimizations can bridge the gap between the uniform framework and how applicability predicates can determine when the optimization can be applied without needing to resort to a subset OCL. We finally identify how this uniform framework is extensible to OCL-based languages such as QVT.

## 1. INTRODUCTION

It is nearly 20 years since OCL first emerged at IBM as an evolution from Syntropy, and nearly 10 years since the OCL 2.0 specification was drafted[10]. OCL is critical to the specification of UML and since no clear alternative to UML or OCL has emerged, why is OCL uptake so limited?

Perhaps OCL enthusiasts need to be honest and look hard at why OCL doesn't work today. Then it may be possible to realize the tremendous promise for the future and to exploit the many demonstrations of the power of OCL formality at research institutions.

### 1.1 Quality

The main official examples of OCL come from the UML 2.4.1[12] and OCL 2.3.1[11] specifications. Analysis[16] of the UML Superstructure specification has identified that over 50% of the 235 Well-Formedness Rules (WFR) have basic syntactic or semantic errors that any competent OCL tooling should detect. The 'correctness' of the OCL in UML 2.4.1 is almost unchanged since UML 2.0.

If the WFRs are so semantically incorrect, what chance is there that they are functionally useful?

Can we expect the wider software community to take OCL seriously when its most obvious exposition is so poor?

The UML 2.5 specification is now 'complete' and undergoing review. The specification is a major simplification. It reorganizes the previous layers of merge-able concepts in the Superstructure and Infrastructure specifications into a single specification of the merged concepts; this is a tremendous improvement and is supported by the use of model-driven auto-generation of the specification from normative models. With high quality models in place, attention has turned to the OCL embedded within them. Eclipse OCL[1] tooling was used to remove 74 syntax errors and correct one or more semantic errors in 250 distinct text regions. Further errors were corrected as missing OCL constraints were added and Diagram Interchange introduced. For the most part, the use of Eclipse OCL in the form of IBM's RSA, allowed authors to correct their own errors directly.

The UML 2.5 embedded OCL is therefore potentially free of syntax and semantic errors, but is it functionally useful?

The constraints are often the primary specification so there is a little to check them against apart from common sense. Some errors may be detected by analysis tools that look for redundancy and contradiction. But in practice, we need to execute them and so discover constraints that are too strong and which reject useful models. Uncovering constraints that are missing or too weak may depend on alert reviewers, users and toolsmiths.

In this paper we discuss work in Eclipse OCL to make the OCL embedded within the UML and OCL specifications usefully executable. On the one hand this facilitates the empirical discovery of the meaningless or over-strong WFRs, and on the other avoids the need for a manual transcription of the WFRs to an implementation language. It also avoids the associated transcription errors. In Section 2 we describe the features of the OCL VM that supports this, and in Section 3 we discuss optimization to give dramatic performance improvements.

### 1.2 Speed

Attempting to use OCL in earnest encounters another credibility gap. OCL execution is often slow, despite the very favorable showing of Eclipse OCL in comparison with other technologies reported in Slide 11 of [6].

In Eclipse OCL, this is because execution has used an interpreter; directly on Ecore for many years, and recently on a UML-aligned pivot representation. Table 1 shows the


*OCL* '12, September 30 2012, Innsbruck, Austria

**Table 1: Eclipse OCL Evaluation Assessment**

| Approach | Release | Performance |
|---|---|---|
| Direct Ecore interpreter | ..., Juno | 72 ms |
| Pivot Value interpreter | Indigo, Juno | 88 ms |
| Pivot Value code gen | Juno | 22 ms |
| 'Optimized' code gen | manual | 0.13 ms |

relative performance using the simplest benchmark from [6] for which the minimal Java code is just

```
return this.feature <= 0
```

evaluated on a set of 12800 model elements. A preliminary code generator was introduced in the Juno release and this gives a 3-fold improvement. Manual execution of plausible optimizations of the auto-generated code suggests that the minimal code is obtainable and so in total there is perhaps a 500-fold improvement available.

Naive validation of large UML models is slow and any loss of performance approaching 500-fold is obviously unacceptable for everyday tooling. The evaluation performance of Dresden OCL was reported as four times slower than Eclipse OCL, so the problem is not unique to Eclipse OCL; better code generation is available in research tools, but accurate fast evaluation has not made it into the mainstream tools.

## 1.3 Basic Code Generation

OCL has many superficial similarities to Java and so it is intuitively attractive to just translate OCL into Java and many authors have done this. Unfortunately OCL is not the same as Java and the intuitive translation only works easily for a Java-like OCL subset.

Some of the problems that a full-functionality OCL evaluator must accommodate are:

- An Integer has no upper bound; Java's int and long have upper bounds.
- An UnlimitedNatural is-a Integer which is-a Real; Java numeric classes are unrelated.
- UnlimitedNatural values have an unlimited value
- Datatype equality is determined by value and so 4 is equal to 4.0; Java numeric objects are unrelated.
- Set(T) entry uniqueness is determined by datatype equality; Java object equality is different.
- All types conform to OclAny
- Collections can be nested and can contain null values
- Null values may participate in computations and the OclVoid type conforms to all types
- Invalid values may participate in computations but may not be used in collections
- Operations may be overloaded
- Additional features may be added by a Complete OCL document
- oclType() may be a gateway to reflective functionality
- allInstances() requires access to a global model view
- The specification lacks precision in some critical areas

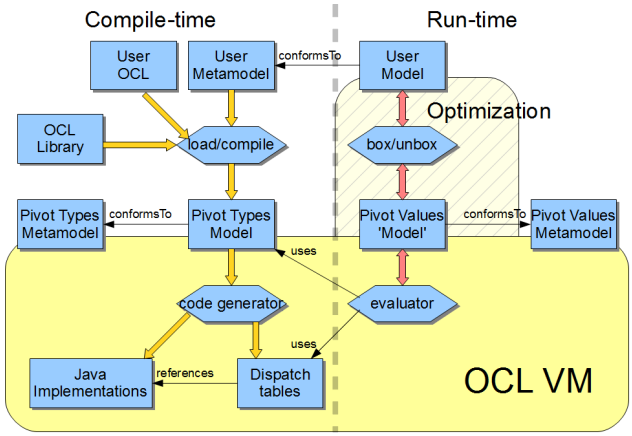


**Figure 1: The OCL VM.**

# 2. OCL VIRTUAL MACHINE

Figure 1 shows the context of the OCL VM. On the left hand side we have the compile-time activities in which user meta-models, user OCL and OCL libraries are loaded, normalized and compiled to give the Pivot Type representation of the structural models and the Abstract Syntax Tree (AST) of each OCL expression. On the right hand side we have the run-time activities in which user models are accessed by the evaluator to perform the required computations. In this section, we describe the various elements of the OCL VM that give a simple predictable full functionality framework. In the next section, we consider optimizations to avoid the overheads of boxing and unboxing model elements and speed-ups for the generated Java implementations.

The classes of the AST such as IfExp and OperationCallExp have distinct but relatively simple semantics, so that it is not difficult for a tree walking evaluator to start at an expression root node and use the appropriate node-class-specific semantics to visit each node in the tree to compute the value of the OCL expression.

We will examine this evaluation in a little more detail to see how the basic computation elements of values and types are used by an operation implementation associated with each AST node.

## 2.1 Pivot Value System

Evaluation requires values to compute with and computation is easier when the values are polymorphic and observe OCL semantics. Neither of these is a characteristic of an implementation language such as Java in which

- Boolean, BigInteger, String and Set share no useful inheritance
- Object rather than DataType equality applies to numbers
- Unbounded Numerics are costly

We therefore take the hint from Clause 10.2 of the OCL specification and use polymorphic values. A polymorphic Value system can have a family of IntegerValue implementations so that

- IntIntegerValueImpl wraps a 32 bit java.lang.int

- LongIntegerValueImpl wraps a 64 bit java.lang.long
- BigIntegerValueImpl wraps java.number.BigDecimal for unbounded support.

IntIntegerValueImpl is almost identical to java.lang.Integer, except that

- numeric overflow is detected to promote growth to a LongIntegerValueImpl
- equality is re-implemented so that same-valued Real and Integer values are equal.

For typical 32 bit usage, the performance penalty of using IntIntegerValueImpl rather than java.lang.Integer is small, and where necessary the unbounded OCL requirements are supported.

The Value types specified by the OCL specification and those used in the OCL VM are shown in the second and third columns of Table 2.1. The differences show the need for a certain amount of remedial action

- Missing Boolean/Real/Integer/Invalid/OrderedSet Values are defined
- Spelling of Bag/Sequence/Set/EnumerationLiteral Values are made consistent
- Type-valued expressions are are supported.

Use of types as values is implicit in OCL expressions such as `oclIsKindOf(SomeType)` but left rather vague in the specification. A practical system must recognize that types can be values and that their types can be templated. Thus a modeled library element declaration[19] might be:

```
operation oclAsType<TT>(type : Metaclass<TT>) : TT
```

## 2.2 Pivot Type System

Values of course have types, and with a polymorphic value system, the type is obtainable internally as `Value.getType()` which provides the underlying support for a fully reflective `oclType()`.

The Type types specified by the OCL specification and those used in the UML-aligned OCL VM[18] are shown in the fourth and fifth columns of Table 2.1.

For the basic data types, the only change is the removal of BooleanType which becomes redundant if a static (metaclass) method of Boolean rather than an instance method of a metaclass is used to model `Boolean.allInstances()`.

For object types, the specification is vague as to how UML-defined types conform to OclAny and what functionality they inherit as a result. The OclElement type that conforms to OclAny is introduced in the Pivot type system as the superclass of all UML-defined types that lack an explicit superclass. OclElement provides added functionality such as allInstances() and oclContainer(). OclType imposes further standard functionality on all types.

LambdaTypes are implicit in OCL as the type of an iterator body[19]. A corresponding LambdaValue unifying ExpressionInOCL may be appropriate for more flexible higher order support.

Finally ObjectValue provides the polymorphic pivot to a practical representation such as an EObject by a derived EObjectValueImpl.

## 2.3 Variables and Environment

Typed Variables (or Parameters) are introduced by the invocation context, let-expressions and iterator-expressions. The OCL specification defines a moderately complex Environment to maintain name value bindings. These are resolved at compile time so that AST nodes have properties such as `VariableExp.referredVariable` for an intermediate value or `PropertyCallExp.referredProperty` for a model access. There are two exceptions to this.

The OCL specification does not model iterations and so there is no `IteratorExp.referredIteration`. Iterations are resolved by a run-time name lookup. The OCL VM models an `Iteration` as an extension of `Operation` and so can exploit an `IteratorExp.referredIteration`[18].

Operations (and iterations) can be overloaded, even though the OCL specification fails to define a dynamic dispatch algorithm. The OCL VM dynamic dispatch support is described in 2.5.

## 2.4 Operations

With values and types, we just need some operations to perform computations. Do we need anything else?

### *2.4.1 If*

An if-expression evaluates only one of its `then` or `else` arguments, so an if-expression must be handled separately from operations.

### *2.4.2 Literal*

A literal expression could be regarded as a zero argument static operation of OclAny, but a literal is easily optimized so there is little point changing the distinct treatment defined by the OCL specification.

### *2.4.3 Model Access*

The `AssociationClassCallExp` and `PropertyCallExp` nodes provide the interaction between user models and the OCL evaluation. The various accesses can be modeled as operation on a host `ObjectValue`. The OCL VM therefore augments each property in the metamodel with an implementation appropriate to the composition, multiplicity or navigability characteristics of the property. This migrates some computation from run-time to compile-time.

### *2.4.4 Operation Bodies*

Once an operation has been determined, an executable implementation of the operation body must be located to evaluate the operation. The compiler therefore annotates each operation (and iteration) in a similar way to properties with an appropriate implementation of the operation.

## 2.5 Dispatch Tables

Dynamic dispatch requires selection of the operation implementation appropriate to the actual source type of an operation. If such a selection is made on a typical metamodel representation, it may be necessary to perform a 6-dimensional search

- Each of the immediate superclasses
- Each of the inherited superclasses
- Each of the operations to match the name
- Each of the parameters to match the name

**Table 2: OCL (Pivot) Values and Types**

| Type Name | OCL 2.3.1 Value | OCLVM Value | OCL 2.3.1 Type | OCLVM Type |
|---|---|---|---|---|
| OclAny | Value | | AnyType | |
| Boolean | ? | BooleanValue | BooleanType | PrimitiveType |
| String | StringValue | | PrimitiveType | |
| Real | ? | RealValue | PrimitiveType | |
| Integer<br>UnlimitedNatural | ? | IntegerValue<br>UnlimitedValue | PrimitiveType | |
| OclVoid | OclVoidValue | NullValue | VoidType | |
| OclInvalid | ? | InvalidValue | InvalidType | |
| Collection | CollectionValue | | CollectionType | |
| Bag | BagTypeValue | BagValue | BagType | |
| OrderedSet | ? | OrderedSetValue | OrderedSetType | |
| Sequence | SequenceTypeValue | SequenceValue | SequenceType | |
| Set | SetTypeValue | SetValue | SetType | |
| Tuple | TupleValue | | TupleType | |
| EnumerationLiteral | EnumValue | EnumerationLiteralValue | ? | Type |
| Object | ObjectValue | | ? | OclElement |
| Type | ? | TypeValue | ? | OclType |
| Lamda | | | ? | LambdaType |

- Each of the parameter type immediate superclasses for conformance
- Each of the parameter type inherited superclasses for conformance.

This can be very expensive in deep inheritance hierarchies and often redundant since there may be no overload to resolve.

The OCL VM therefore prepares dispatch tables for which the static referredOperation provides an integer operation signature index and inheritance depth reducing the problem to 1-dimensional

- Each of the same-depth superclasses

In practice even deep inheritance trees are relatively narrow so the search usually involves a simple indexed lookup in a single class. Only rarely is it necessary to examine two or three alternatives.

The dispatch tables also support improved speed for type conformance checks underlying `oclIsKindOf()` and reflective operations such as `oclType().ownedOperations()`.

# 3. CODE GENERATOR

The Juno release of the Eclipse OCL Examples and Editors includes an experimental release of the OCL VM support, in which the code generator is integrated with the EMF model generator. If the OCL preference page option to use the code generator is enabled, EMF generation then provides the OCL VM dispatch tables and implementations automatically.

The Juno release pursues the naive approach of serializing the OCL AST as Java code that sequences the dynamic dispatch of an operation for each operation-like AST node. The resulting code is simple and supports the full OCL semantics[1]. The three-fold speed up compared to the interpreted approach is useful but not spectacular. Additionally, the run-time compilation costs are eliminated since the OCL parsing now occurs at compile-time.

The Juno code generator uses Acceleo Model to Text templates to perform a direct AST model to Java text translation. A class comprising a single polymorphic function for each implementation. The function body contains two text regions.

The first text region provides a constant and variable declarations. Common Subexpression Elimination is achieved by the simple strategy of pruning duplicate text lines.

The second text region sequences the implementation calls from an AST tree walk. Since everything is an implementation call and there is no inlining yet, there are no operation-specific patterns as used by other authors.

## 3.1 Optimizations

With a full-functionality VM in place, it is now possible to look at optimizations that perform M2M rewrites of the OCL AST before performing the final M2T code generation to a preferred target language.

Initial manual experiments suggest that a further 100-fold speed improvement is available by:

- Directly accessing EMF properties as getXXX(), rather than eGet(XXX)
- Directly dispatching operations for which dynamic dispatch is redundant
- Inlining operations that are directly dispatched
- Unboxing Pivot Values to use the underlying Java values or EObjects directly
- Using an Unboxed operation call in the VM
- Directly accessing EMF object fields and bypassing getXXX().
- Performing a direct Collection mutation to re-use a no-longer-used Collection

[1] oclIsNew(), @pre, States and Messages are to-be-done

Each of the above optimizations has associated guard conditions, so for instance an `IntegerValue` can only be unboxed to an `int` if the dynamic range can be analyzed and found compatible with 32 bits. Unfortunately UML does not naturally support specification of value bounds, so opportunities for unboxing are limited to discovering that the source value is 32 bits and propagating until arithmetic is performed.

Similarly the absence of invalid and null values must be ascertained to allow the associated polymorphism to be discarded.

The progressive application of optimizations to a valid AST with full OCL functionality offers the prospect of retaining full-functional validity after optimizations. This is difficult to achieve when a direct single stage pattern pasting approach to code generation is taken.

## 3.2 Beyond OCL

Efficient evaluation of OCL is useful, but increasingly OCL is now used within extended contexts such as QVT[13] or MOFM2T[9]. These languages define an extended AST and so in principle are amenable to the same tree-walking evaluation as the basic OCL AST. In practice, and in particular for QVT Relations, substantial strategic planning will be needed prior to evaluation. Nonetheless the same basic interpreter can support naive evaluation and the code generator can be extended to the extended AST.

QVT Operational introduces an Imperative OCL extension that clearly violates the side effect free characteristics of OCL. This requires a two level approach with an overall structure that sequences side effect free OCL sections.

# 4. RELATED WORK

Many authors have provided a partial OCL code generator demonstrating good characteristics aligned with some research goal. Omission of OCL facilities such as allInstances(), oclIsNew(), @pre, States, Messages and even Tuples and opposites is common, but in many cases this just represents a pragmatic reduction of scope to facilitate research; these omissions can be rectified by a little more work. Failure to address unbounded numerics, numeric equality, null/invalid propagation, nested Collections and oclType() is a more fundamental challenge to some of the approaches.

Wilke[15] describes a reworked Java generator for Dresden OCL based on parameterized fragments. AspectJ is used to support model access in Java models. However many Java types are used directly and so functionality is limited to Java-like semantics for numeric range and equality.

Heidenreich[5] describes a Dresden OCL generator for SQL based on identifying typical patterns of database usage. It is not clear that this is able to handle arbitrary OCL or OCL that fails to exhibit SQL-like characteristics.

Egea[4] describes a MySQL generator to avoid the heavy overhead of loading a large model into an OCL tool. Stored procedures are used to realize the iterations that are common to many typical applications. The procedures are then executed within an SQL database. This is an interesting deployment option but does not help with full-functionality OCL code generation.

Shidqie[14] introduces Imperative Ecore as an intermediate model to separate OCL restructuring and Java formatting concerns, but ignores non-Java-like aspects such as unbounded numbers.

Moiseev[8] takes a more progressive transformational approach realized as rewrites in Maude. This is clearly beneficial when supporting multiple target languages, unfortunately it ignores the awkward aspects of OCL.

Mezei[7] relocates and caches OCL evaluations to reduce navigation costs.

Efficient code is one important aspect of OCL execution. Efficient scheduling is an equally important but largely orthogonal issue. Cabot[3] provides a review of the disappointing scheduling support in many tools that support Constraint evaluation. Eclipse OCL was overlooked in the review. Eclipse OCL inherits the default EMF capability to perform a total validation. Eclipse OCL has an Impact Analyzer that supports selective re-evaluation.

The need for variability in both the type and value system was recognized by Wilke[17]. The pivot Value and Type systems correspond to the variation points but normalize the external view to support a uniform OCL engine rather than adapting the engine to the external.

The very useful concept of a pivot[2] model was introduced by Dresden OCL.

# 5. CONCLUSIONS

We have identified the poor quality of official OCL expositions and the poor speed of accurate OCL evaluation as problems for OCL credibility in the wider software community. The imminent step change in quality with the UML 2.5 WFRs provides a further stimulus to tackle fast accurate evaluation.

We have noted the semantic differences between OCL and Java and the consequent difficulties of a direct translation for more than a Java-like OCL subset.

In contrast we have noted that the OCL AST is very amenable to a tree-walking evaluation, and described how a uniform polymorphic Pivot Value system and an associated Pivot Type system can support evaluation using an appropriate node-specific implementation.

The OCL VM's dispatch tables, which are automatically generated during EMF model generation have been outlined and shown to reduce a potentially six dimensional search for operation dynamic dispatch to barely one dimensional.

The experimental implementation of the above gives only a three-fold speed-up compared to interpretation. Manual experiments of model-to-model transformations that optimize the OCL AST have demonstrated that there is a 500-fold improvement available.

Extension of the OCL VM to support OCL-based languages should be possible.

# 6. REFERENCES


[1] Eclipse mdt/ocl project.

[2] M. Bräuer and B. Demuth. Model-level integration of the OCL standard library using a pivot model with generics support. In *Ocl4All: Modelling Systems with OCL*, Models 2007, Nashville.

[3] J. Cabot and E. Teniente. Constraint support in mda tools: a survey. In *European Conference on Model-Driven Architecture 2006, LNCS 4066, pp. 256-267.*

[4] M. Egea, C. Dania, and M. Clavel. Mysql4ocl: A stored procedure-based mysql code generator for OCL.

In *OCL 2010: Workshop on OCL and Textual Modelling*, Models 2010, Oslo.

[5] F. Heidenreich, C. Wende, and B. Demuth. A framework for generating query language code from OCL invariants. In *Ocl4All: Modelling Systems with OCL*, Models 2007, Nashville.

[6] B. Izsö, Z. Szatmári, and I. Ráth. High performance model queries and their novel applications. In *Third Biannual Workshop on Eclipse Open Source Software and OMG Open Specifications*, OMG quarterly meeting, Reston.

[7] G. Mezei, T. Levendovszky, and H. Charaf. Optimization algorithms for OCL constraint evaluation in visual models. In *Periodica Polytechnica 2007*.

[8] R. Moiseev, S. Hayashi, and M. Saeki. Generating assertion code from OCL: A transformational approach based on similarities of implementation languages. In *Proceedings of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, pages 650–664, Oct. 4–9 2009.

[9] Object Management Group. *MOF Model to Text Transformation Language*, v1.0, OMG Document Number: formal/2008-01-16 edition.

[10] Object Management Group. *Object Constraint Language*, version 2.0 draft, OMG Document Number: ptc/03-10-14 edition.

[11] Object Management Group. *Object Constraint Language*, version 2.3.1, OMG Document Number: formal/2012-01-01 edition.

[12] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure*, version 2.4.1, OMG Document Number: formal/2011-08-06 edition.

[13] Object Management Group. *Query/View/Transformation Specification*, version 1.1, OMG Document Number: formal/2011-01-01, 2011 edition.

[14] A. J. Shidqie. *Compilation of OCL into Java for the Eclipse OCL Implementation*. 2007.

[15] C. Wilke. *Java Code Generation for Dresden OCL2 for Eclipse*, technische universitat dresden, 2009 edition.

[16] C. Wilke and B. Demuth. OCL is still inconsistent! how to improve OCL constraints in the OCL 2.3 superstructure. In *OCL 2010: Workshop on OCL and Textual Modelling*, Models 2010, Oslo.

[17] C. Wilke, M. Thiele, and C. Wende. Extending variability for OCL interpretation. In *OCL 2010: Workshop on OCL and Textual Modelling*, Models 2010, Oslo.

[18] E. D. Willink. Aligning OCL with UML. In *OCL 2011, International Workshop on OCL and Textual Modelling*, TOOLS 2011, Zurich.

[19] E. D. Willink. Modeling the OCL Standard Library. In *OCL 2011, International Workshop on OCL and Textual Modelling*, TOOLS 2011, Zurich.