Challenges for Code Generated OCL Execution

Edward D. Willink
ed_at_willink.me.uk
Willink Transformations Ltd.
Reading, Berks, England

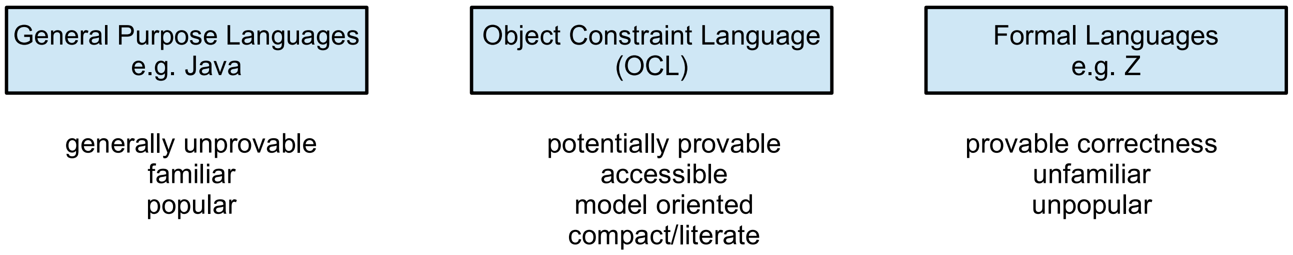

Figure 1: The OCL Compromise.

ABSTRACT

While OCL is primarily a specification language supporting the elaboration of often-graphical metamodels with textual constraints, it is also executable enabling the constraints to be used to validate models. The superficial textual similarity of OCL and Java has tempted some authors to attempt a textual transliteration to facilitate a faster Java execution. Unfortunately there are many aspects of OCL semantics that deviate from Java and so transliteration is close to impossible. We identify the semantic differences so that new transliteration attempts can review the almost inevitable limitations of an OCL-like transliteration when choosing to implement a Java-Friendly OCL rather than a full OCL code generator.

KEYWORDS

OCL, Code Generation, Optimization, Transliteration

ACM Reference Format:
Edward D. Willink. 2022. Challenges for Code Generated OCL Execution. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3550356.3561537

1 INTRODUCTION

OCL is primarily a specification language supporting the elaboration of often-graphical metamodels with textual constraints.

As shown in Figure 1, OCL[9] compromises the familiar popularity of typical general purpose languages, such as Java, with formal languages, such as Z. Formal languages are so unfamiliar as to be unacceptable to many programmers. OCL's use, in conjunction with UML[10], ensures that OCL has model oriented support. OCL's functional style facilitates the definition of a constraint as a compact expression.

The foregoing contrast is important but just ergonomic. More important for specification purposes is correctness, for which the generality of General Purpose Languages makes correctness proofs very hard if not impossible. In contrast, Formal Languages can be provably correct but the time and skills necessary to provide that proof are not available to many programmers or applications. OCL is positioned nicely between these two extremes. The relevant, accessible syntax means that 20 years on OCL is still the language of choice for writing model constraints. The potential for proof arises from the underlying side-effect-free functional specification principles. Sadly, the potential for proof has not been exploited in mainstream tools and so remains just a potential for most specifications.

As a specification language, OCL is just a model-oriented pseudo-code, for which tooling ensures some degree of syntactic consistency with the standard. It is only once OCL's executable capability is realized that the pseudo-code is elevated to functional accuracy.

Model validation is an important use case in which a model is loaded and all the applicable OCL constraints are executed to confirm that a model's Well Formedness Rules are respected. For large models, the efficiency of the execution is important and so it is attractive to replace a direct naive interpreted OCL execution of the WFRs by distinct compilation and execution phases. The compilation phase generates code in a language that can be executed faster.

In this paper, we first review the attraction of a transliterator in Section 2 and contrast it with the traditional tooling in Section 3. OCL and Java compatibility challenges are addressed in Section 4 for syntax and Section 5 for semantics. We then consider some code synthesis concerns in Section 6 before summarizing the simplifications that a Java-Friendly OCL might make in Section 7. Related work follows in Section 8 and we conclude in Section 9.

OCL	Similar Java
name	name
(a = b) and true	(a == b) && true
'xyzzy'.substring(a,b)	"xyzzy".substring(a,b)
if a then b else c endif	if (a) { b } else { c }
elements->forall(e : Element | body)	for (Element e : elements) { body }
Set{1, 1.0}->size()	Sets.newHashSet(1, 1.0).size()

Table 1: Alluring Transliterations

2 TRANSLITERATION

Table 1 shows examples of the kind of superficial textual and functional similarity between OCL and Java that have enticed some authors to attempt a textual transliteration.

The examples in the table clearly show that neither character-by-character nor word-by-word transliteration is feasible. The challenge for a transliteration approach is to do sentence-by-sentence transliteration rather than a total analysis of the whole document followed by total synthesis by a code generator.

Unfortunately there are many aspects of OCL semantics that deviate from Java and so transliteration is close to impossible; there are too many concepts that require total analysis.

We briefly identify some errors in the alluring transliterations.

`name` to `name` is only correct if `name` is the same concept in both languages. An unnavigable opposite would not be the same.

`"xyzzy".substring(a,b)` neglects to correct from 1-based to 0-based indexes.

OCL's `forall` is a multi-term `and` operation whereas Java's `for` does not relate its iterations. The transliteration should be more like:

```
boolean result = true;
for (Element e : elements) {
  if (!body) {
    result = false;
    break;
  }
}
... = result;
```

The `Set{1, 1.0}->size()` demonstrates the uniformity of OCL numbers; the OCL size is 1 since OCL counts the number of distinct numeric DataType-typed values. Java numbers are not uniform; the Java size is 2 since Java counts the number of distinct numeric Class-typed objects. A better transliteration could be

```
Sets.newHashSet(Double.valueOf(1), 1.0).size()
```

We revisit these and many other semantic differences so that a transliteration author can choose whether to resolve each difference by implementation heroism in the transliterator or by compromising to offer only a defective OCL subset that we refer to as Java-Friendly OCL (jfOCL).

3 TOOL STRUCTURE AND PERFORMANCE

To understand what the transliterator does not do, it is helpful to review what standard tooling provides.

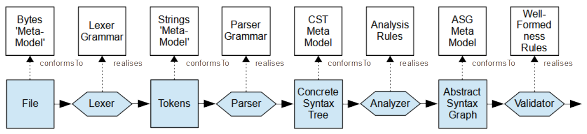

Figure 2: Traditional OCL Tool Architecture.

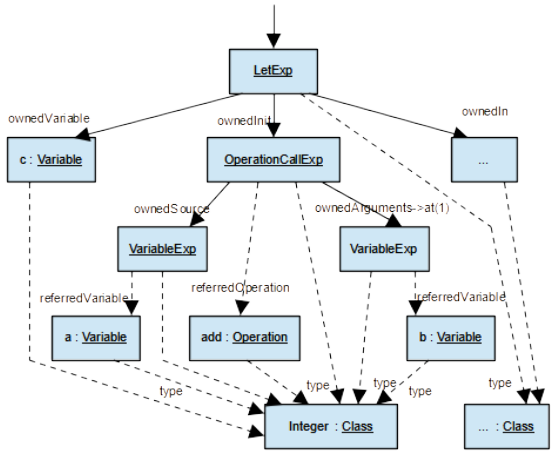

Figure 3: ASG of let c=a+b in ... example

3.1 Traditional Tooling Architecture

The OCL specification separates concerns by suggesting an architecture such as Figure 2 for an OCL tool.

Lexer and Parser realize appropriate grammars to convert source file text characters via multi-character tokens to the Concrete Syntax Tree nodes.

The Concrete Syntax provides a model that is close to the source text. Short forms are not elaborated. References are retained as unresolved names. The Concrete Syntax is therefore a Tree.

The Analyzer resolves the names to appropriate model elements, expands shortforms, synthesizes implicit iterators, elaborates implicit collect and infers types to annotate each node with accurate usage. A simple example for the OCL expression `let c = a + b in ...` is shown in Figure 3 using the style of a UML Object Diagram.

The composition tree has its root at the `LetExp` node. It has an `OperationCallExp` child and two further `VariableExp` grandchildren. Composition is shown using directed solid edges between the nodes whose type is underlined. Additional references are shown as dashed edges. The `referredOperation` identifies that `OperationCallExp` executes using the `Operation` named add. Similar `referredVariable` edges identify the variable to be read by each `VariableExp`. The ancestry of the `a` and `b` variables and the `Class` instances is not shown to avoid clutter. Nearly all the nodes in this simple example use the `Integer` node as their `type`. (The `...` node and class are not relevant to this example; just necessary to make the `LetExp` usage valid.)

The Abstract Syntax provides a rich model that is suitable for execution or further analysis. References are resolved unambiguously to model elements; the Abstract Syntax is therefore a Graph.

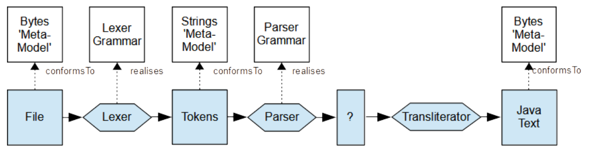

Figure 4: Transliterator Architecture.

The Validator is an example of a tool that executes the Abstract Syntax to determine whether a model confirms to its OCL-defined Well Formedness Rules. Execution typically uses an evaluator that interprets the ASG.

A Code Generator is another tool that can use the Abstract Syntax. It translates the ASG into a language that can execute more rapidly.

The existence of, and distance between, the CS and AS models is one of the major inconveniences for OCL tooling. The 'draft'-like precision [16] of the OCL specification is yet another major inconvenience requiring implementers to work hard to determine what was intended rather than specified.

3.2 Transliterator Architecture

An attraction of the transliterator is that it has a simpler tool architecture, as shown Figure 4. The complexities of an Abstract Syntax are ignored; rather the transliterator comprises a simple serializer direct from the parser output to the required Java text. Ignoring the AS discards all the useful analysis that is captured in the AS. A transliterator author has just two choices. Either to replicate the analysis that underpins the AS, or to simplify the OCL language to such a degree that no analysis is needed.

3.3 Evaluator Architecture

While the Abstract Syntax is technically a graph, each OCL constraint is defined by an expression that is represented by a composition tree in which there is a child for each value needed to compute the result. Evaluation of an OCL expression is simply performed by a depth-first traversal of the composition tree. Once the traversal has reached the leaves, execution proceeds as each node returns its result to its parent.

If we consider the execution necessary for our simple example, whose OCL ASG is shown in Figure 3, we can see the execution cost of traversing the tree will far exceed that of the minimal pseudo-assembler for c=a+b.

```
LD r0,a        -- load register r0 from memory a
LD r1,b        -- load register r1 from memory b
ADD r2,r0,r1   -- add r0 to r1 with result in r2
ST r2,c        -- store register r2 to memory c
```

We might rate the pseudo-assembler as 25% efficient since only one of the lines does necessary work; the other three control instructions shuffle values around memory and registers. In practice, we may achieve higher efficiency as the stores for a first expression may be rendered redundant with respect to the loads of a next expression. In the extreme case of a perfectly understood execution and unlimited registers, a near 100% efficiency is possible. Of course a perfectly understood execution will give a very long entirely linear execution flow that need not actually execute at all; it can be optimized to return just the perfectly understood result.

Realistically, some control overhead is necessary for loops that accommodate data dependencies and reduce program size. A practical assembler implementation might perhaps aspire to 50% efficiency.

If we now use a LISP-like assembler notation and consider the overheads of interpreting each opcode:

```
(ST (ADD (LD a) (LD b)) c)
```

There will be control overheads as each opcode such as LD is fetched and dispatched. Each nested term will result in a nested function call. It is easy to see that there are now many additional control overheads that are likely to reduce the efficiency well below 10%.

The 10% estimate above and further estimates below are based on experience. The numbers are a little more helpful than purely relative terms such as 'really inefficient'. The numbers cannot be precise since a much better performance can be achieved when the programmer is able to adopt a very direct flat bloated style. A more maintainable style that satisfies diverse use cases with many overrideable helper functions may give worse performance. For the particular case of `LD A`, a single assembler instruction may easily bloat to many hundreds of instructions once an interpreted access uses a hierarchy of scoped `Maps` with each access needing to compute a hash code in order to lookup an entry and then invoke an equality test to check on a hit. Contrasting the hundreds of instructions for a `Map` access with the typically better performance of a small `List` search is not the topic of this paper. An efficient code synthesis may require just a single instruction by locating `A` in a local variable.

A LISP-like selective rendering of the OCL ASG is even less direct

```
(LetExp #c
  (OperationCallExp "add"
    (VariableExp #a)
    (VariableExp #b)
  )
  ...
)
```

After finding the `OperationCallExp` opcode, the interpreter must fetch the `"add"` argument and select the appropriate actions. This involves yet more control overheads so that the efficiency of interpreted OCL execution may well be only 1%. This is clearly very disappointing and motivates the use of a code generator that can certainly do very much better than 1%. At least 10% should be possible using Java. Perhaps 25% could be achieved using C as a portable assembler language.

Perhaps the simplest approach to a speed up just provides a large library of helper functions for everything so that we may execute the Java serialization

```
doLetExp(env, "c",
  doOperationCallExp_add(env,
    doVariableExp(env, "a"),
    doVariableExp(env, "b"))
  )
);
```

The above has already reified the "add" into the operation-specific doOperationCallExp_add helper, but has not reified "a" as a variable; rather an env is passed around. doVariableExp may get from, and doLetExp may put to, the environment at a non-trivial cost. Clearly we can do better by exploiting Java variables directly, but we can start to see the challenges. Each change that we make to exploit a Java capability to realize an OCL capability must ensure that the Java execution observes OCL semantics. We will now examine the many ways in which OCL semantics differs from Java and why the following 'obvious' Java is too simplistic.

```
int c = a + b;
```

4 SYNTAX COMPATIBILITY

We will first look at aspects of the OCL syntax that require accurate processing by the CS to AS analysis to determine the meaning. The code generator must faithfully represent the meaning as valid Java code. It is not clear how a transliterator can resolve the incompatibilities without the AS resolutions to guide the transliteration.

4.1 Reserved Words

Languages have reserved words that the user may not use as regular names. Use of a reserved word as a regular identifier almost always results in a compilation error that will prevent the transliterator being used. In pathological cases use of a reserved word could create a differently 'valid' program.

In Essential OCL, names such as self, else and not are reserved. In Complete OCL, names such as context and package are also reserved. forall is not reserved. T and T1 are also reserved but only to support the evasion of a proper implementation of UML templates. There is an escape mechanism so that _'\n' or _'self' may be used; any Unicode character sequence is an acceptable OCL name.

In Java, names such as this and package are reserved. There is no escape mechanism.

A transliteration of OCL to Java must therefore provide a mechanism to ensure a distinct and valid Java spelling for every possible OCL name spelling. jfOCL may attempt to push this responsibility back onto the OCL programmer, but sometimes metamodels have awkward names such as class. Third party metamodels cannot be changed for the convenience of jfOCL.

4.2 Shortforms, Name capture, Implicit Sources

Object Oriented languages often allow a field of the current context object to be accessed without explicitly specifying the context object. Thus self.name in OCL, or this.name in Java can be shortened to just name. This is friendly but requires the transliterator to understand the OCL shorform before attempting to exploit a corresponding Java shorform.

4.2.1 Java Shortforms. In Java, name may refer to

- a function parameter
- a local variable
- a non-static field of this
- a static field of this.getClass()

When there is only one option, user and tooling may be happy. When there is an unresolvable ambiguity, tooling must report a compilation error. But sometimes there is a resolution priority; a function parameter hides a non-static field. This priority may not always suit the user, so tooling may offer a warning to encourage the user to rename the parameter and so avoid the hazard.

Further complexity arises when an outer definition may be accessed from within

- a nested class
- an anonymous function
- a lambda.

4.2.2 OCL Shortforms. OCL's name resolution policies are similar

- an operation parameter
- a let variable
- a non-static property of self
- a static property of self.oclType()

but include

- any unique iterator
- an advanced navigation
- a type
- a template parameter

Implicit Sources. The OCL implicit iterator source is dangerously flexible. In

```
books->forall(b
        | chapters->forall(c
                   | authors->notEmpty()))
```

authors can be a shortform for c.authors or b.authors or self.authors or a let-variable or a function parameter. The iterator shortform is a candidate even when the explicit b and c iterators are implicit. Resolution of the shortform requires a search of the metamodel to see whether Chapter::authors or Book::authors exist before considering the usual candidates. jfOCL might well prohibit the dangerous nested implicit source. Prohibiting the immediate loop iterator could be a step too far.

OCL Navigation. In addition to the traditional Object Oriented instance to slot interpretation of the dot navigation operator, OCL normalizes a variety of different forms of navigation behind the one syntax. The most complicated is probably the implicit opposite navigation for which the transliterator must generate a call to an appropriate helper that searches the model for the required opposite. See Section 5.2 and Section 5.8.2 for further details.

OCL Type Names. OCL lacks a distinctive type literal syntax analogous to Java's MyClass.class. Consequently the argument for oclIsKindOf(V) may refer to at least the type V or the property V or the let-variable V.

OCL is UML-aligned, consequently OCL supports UML templates even though the OCL specification lacks any grammar, concrete or abstract syntax support. Once templates are supported, template parameters are yet another option for a name reference.

4.2.3 Summary. The many differences in name resolution make it unsafe to attempt to transliterate shortforms directly from OCL to Java. The actual access must be understood, matched to an appropriate Java access, and then serialized in a way that ensures that the appropriate access is actually used.

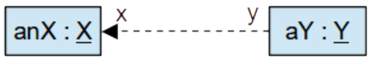

Figure 5: Navigation example

5 SEMANTIC COMPATIBILITY

Once the OCL semantics has been identified, it is possible to translate the OCL into Java that exhibits the identical OCL semantics. We must therefore identify how and when OCL semantics can be realized in Java. Generating Java without an appropriate understanding risks generating inaccurate code. We will therefore examine the compatibility of various constructs.

5.1 Program Structure

Java package, class and operation constructs are very similar and more powerful than OCL's so there is little difficulty in mapping OCL structural concepts to a subset of Java.

5.2 Properties

OCL, or rather UML or Ecore, has Classes with Properties in a similar fashion to Java's Classes with Fields, so it would seem that the `aY.x` OCL navigation using the unidirectional association from `Y` to `X` in Figure 5 can be transliterated to `aY.x` in Java.

But this is to ignore the power of OCL's model oriented support. OCL supports many different forms of Property navigation, many of which appear to be simple.

Even the simplest access may need care since OCL does not specify the reification of each class instance. Distinct mechanisms will probably be needed depending whether the source and/or target class instance is:

- Java Object
- Java Bean
- EMF EObject
- XML Element
- Object Data Base Object
- UML Instance
- UML Stereotype
- UML Association

The transliterator must invoke an appropriate helper, since direct use of Java's mechanism is not often correct. Distinct mechanisms will usually be needed depending whether the access is to a DataType or a Class object and whether the access is to a single object or a collection of objects.

OCL hides a variety of access mechanisms behind the simple dot-navigation syntax. The unnavigable opposite syntax is useful and powerful but hard to implement. With respect to Figure 5, OCL supports the `anX.y` navigation even though `X::y` is not navigable. Implementation of this is discussed in Section 5.8.2. Further lexical complexities arise when the `y` role is omitted from the metamodel. A default name is available by re-using the name of the target type allowing `anX.Y`. If the auto-generated name is ambiguous the more elaborate `anX.Y[Y::x]` syntax may be used.

Yet more complexity arises from

- static properties
- derived properties
- inherited properties
- Stereotype properties
- Association-owned properties
- UML subset/union/redefined properties

Access to a property is only part of the challenge. Many properties may have an OCL expression defining the initial and/or derived value that must be calculated in a timely fashion and which may need to be cached to avoid redundant calculation. But once cached, the external usage must be respected to invalidate the cache in an equally timely fashion.

It is not clear how this can be done without an overall analysis to determine all the forms of property access that occur, then to plan an implementation strategy, and finally to synthesize distinct construction code as well as the use of an appropriate helper for the actual access.

5.3 Control Semantics

OCL's operation call and return is broadly compatible with Java.

OCL's `if then else endif` is easily accommodated in Java.

An OCL let-variable is relatively easily realized by a Java variable declaration.

OCL's loops do not have direct Java counterparts, so helpers or templates are required. See Section 6.2.

There is a major incompatibility in regard to OCL's `invalid` which is philosophically the same as a Java exception but very different implementation-wise. `invalid` handles the very inconvenient use case where the program flow fails perhaps due to a program error, perhaps due to an external environment problem.

OCL's `invalid` is a value that is returned as an alternative to the intended value and must be propagated through every computation until it is handled or returned as the overall result.

In contrast, a Java exception is a `Throwable` instance that is thrown. It bypasses all intermediate computations until a catch handler is encountered.

These very different practical semantics are not amenable to naive transliteration.

An implementation may chose to use Java exception semantics throughout the transliteration. Every generation of an OCL `invalid` must then be replaced by a thrown `Throwable`. Conversely every OCL operation that may be passed an `invalid` value must have an operation call that carefully catches any exceptions to create the required `invalid`. This is inconvenient for ordinary code, it is distinctly awkward when the exception can occur while initializing a let variable. The awkwardness arises because an exception while evaluating the 'init' variable child of a `LetExp` must be caught so that it may be correctly propagated or suppressed according to the logic of the 'in' child.

Alternatively, an implementation may use OCL `invalid` semantics throughout the transliteration. This then requires every helper to be accurately coded to guarantee that exceptions are returned as `invalid`, a guarantee that is likely to be overlooked by maintainers. Every call to a helper may therefore need to wrapped in an exception conversion. All helpers must be coded to propagate `invalid`.

The latter approach could be suitable if `invalid` was a prolific use case, but hopefully `invalid` is rare, so it is more efficient to work with, rather than against, Java's natural capabilities. It is appropriate to analyze the potentially invalid flows in order to plan the necessary catch handlers. An accurate analysis can ensure that only those very rare let-variables that can propagate an invalid value incur the overheads.

Implementation of `invalid` is a significant pain, so tools may follow USE's[12] example and omit support for `invalid` altogether. However this is not OCL; it provides support only for OCL programs that never fail. There can be no divide-by-zero, no null-navigation, no index-out-of-bounds on ordered collections.

None of the above approaches is attractive. Careful study of the OCL specification reveals that it is well-intentioned but irrational [17]. A much more practical and sensible design can be compatible with Java exception semantics. Unavoidable exceptions such as a network-failure are classified as desirable exceptions that should always be thrown. Programming errors such as collection-index-out-of-bounds are classified as undesirable exceptions that should be eliminated by analysis at compile time. The challenging need for commutative rather than short-circuit `and`/`or` operators that uncrash can be eliminated by commuting the inputs at compile time to guarantee that only the second input can crash.

The OCL validity analysis [17] is a non-trivial whole program analysis that cannot possibly be done sentence-by-sentence. However, while the motivation of the analysis is to prove that the OCL is free of undesirable crashes, jfOCL can follow the principles and define all crashes as desirable avoiding the need to catch any exceptions anywhere. It is only necessary to convert `invalid` to a thrown exception. This is easily done by providing no reification of `invalid`; exceptions can be thrown from the outset. With no catching of exceptions, a rational implementation of `and`/`or` that computes first argument first will exhibit the familiar short-circuit behaviour.

5.4 Type / Value

When we use helper functions to implement each OCL semantic, the values we pass to the functions and which are returned by those functions should observe OCL semantics and conform to OCL types.

Before we examine typical types, we should note one important difference between OCL/UML and Java. UML defines two kinds of Object, Class-typed and DataType-typed. A Class is instantiated as an instance which typically has a unique address and so two distinct class instances are not equal. A DataType defines a value that is often embedded as the attribute of an instance. A DataType does not have a unique address. DataType values are equal when a deep comparison of their information fields finds all are equal.

Java has user-defined Class instances but only built-in primitive DataType values. There is a flexibility to overload `equals()` so that, for instance, a pair of `String` instances can compare equal even when their addresses differ. This is important to avoid the need for `Map` keys to be used without the overhead of interning. However, this can lead to some surprises whenever `int`, `double`, `Integer` and `Double` are intermixed. The surprise is demonstrated by the difficulty in transliterating `Set{1, 1.0}->size()` to ensure that the two representations of the one value populate a single-entry Set.

The overt example mixing real and integer literals is perhaps unrealistic so jfOCL could prohibit it. More complex examples might not be apparent to the user, so jfOCL should diagnose them, however the effort required to diagnose them is comparable to the effort required to enforce the common type determined by the regular OCL analysis. jfOCL should really support numeric value uniformity.

5.4.1 Boolean, String. The semantics of Boolean and String values are compatible so Java's type and values can be used. Only a little care is needed where the spelling of operations such as `toUpper` varies.

5.4.2 UnlimitedNatural. Java has no counterpart to UnlimitedNatural, so a helper type is necessary. However UnlimitedNatural is not a numeric type, rather it is an unbounded enumeration used solely to support the upperbound of UML's Multiplicity. jfOCL might omit UnlimitedNatural.

5.4.3 Integer, Numeric Precision. OCL specifies two ideal numeric types `Real` and `Integer`, whereas Java specifies many practical types such as `int`, `Long` and `BigInteger` with inconsistently imperfect functionality. A 32 bit integer is sufficient for almost all use cases, but just occasionally position or time calculations need more.

Suppose a salary is represented by a 32 bit integer. After many generous pay rises, a salary may grow to the 31 bit threshold. At this point, the salary wraps around to a negative number; the employee must now pay to work.

This is OK in Java; it is what the specification says should happen. It is absolutely not OK in OCL for which unbounded integers grow and grow.

An OCL2Java conversion must choose a Java representation for OCL's `Integer`

- always 32 bit Integer
- always 64 bit Long
- always BigInteger
- use a variety of types

32 bits is simple and efficient but significantly defective.

64 bits is simple and similarly efficient on a modern 64 bit machine and very rarely defective.

BigInteger is costly for the vast majority of use cases and defective only in pathologically huge use cases.

The approach taken by the pivot-based Eclipse OCL is to use a polymorphic abstract `IntegerValue`, It has `LongIntegerValue`, `IntIntegerValue` and `BigIntegerValue` concrete derived classes. The implementation of addition returns a larger type when required.

`IntIntegerValue` like Java's `Integer` wraps an underlying `int` content so there is very little performance penalty for the 'correct' solution.

5.4.4 Real. Floating point arithmetic is little used in OCL, so there is little benefit in struggling to use `Float` or `Double` rather than `BigDecimal` everywhere.

5.4.5 Object. References to Model Elements can be compatible. Conversely, if they are not, the overheads of converting or wrapping Objects incur an undesirable cost.

5.4.6 Collection. Java's Collection types were originally designed as non-template types. These were upgraded to templated classes when Java 5 added generics, but backward compatibility necessitated a concept of type erasure. This means that the actual templating type is not known at run-time preventing expressions such as `new T()`.

OCL's collection types have a magic underspecified `T` that behaves in many ways like a template parameter. Once the requirement for UML alignment is respected, the magic `T` can be respecified as a regular template parameter and the rules for collection type conformance revisited to avoid the anomaly that `Set{}->including(4)` is erroneous. The empty `Set{}` is a `Set(OclVoid)`. When the source argument is used to specialize the `Set(T)::including(T)` operation to `Set(OclVoid)::including(OclVoid)` the error arises because 4 does not conform to `OclVoid`. More declaratively, there is an alternative specialization `Set(Integer)::including(Integer)` for which both source and argument are type conformant.

Eclipse OCL applies this declarative philosophy to ensure that the Abstract Syntax is annotated with consistent specializations. The `Operation::type` declaration has the unspecialized type. The invoking `OperationCallExp::type` has the specialized type. These types are propagated and checked throughout the AST. They provide the accuracy necessary to declare variables in Java and to inject the necessary casts from what the OCL analysis guarantees to what the Java declarations require.

The Java `Set<T>` and OCL `Set(T)` classes are templated so that the programmer can enforce their intent. A `SetValue` class that provides the underlying implementation of `Set(T)` for OCL does not need to be a templated class since the underlying implementation can access the template parameter more flexibly as a regular field rather than as a template parameter.

Unless jfOCL replicates the collection type analysis, the transliterator may have to sprinkle the code with casts of dubious validity.

Java provides `ArrayList` and `HashSet` that can be used relatively easily to implement OCL's `Sequence` and `Set` collections. `LinkedHashSet` can be used for `OrderedSet` once a bug-fixing `equals` overload is provided to ensure that differently-ordered contents are not equal. There is no Java counterpart for `Bag`. jfOCL can exclude `Bag` or provide a custom helper type.

5.4.7 Compatible Value Representation. One of the many challenges in implementing an OCL tool is the lack of a specification of the surrounding environment. OCL is an add-on to an existing system with which OCL must co-exist. There may therefore be three distinct representations of the same value. So considering a Set of Integer and EMF as an example surrounding environment, the usage may need to be

- SetValue for use in OCL support helpers
- Set<BigInteger> when interacting with a Java routine
- EList<EInt> when interacting with EMF

The transliterator must track the prevailing representation of each passed value and invoke conversions whenever necessary.

5.4.8 Conformance, Inheritance, OclAny, OclVoid. OCL supports type conformance whereas Java supports single-class, multiple-interface inheritance.

For most types conformance and inheritance are the same, but OCL adds a conformance from all classes to `OclAny` and from `OclVoid` to all classes.

UML[8] (and Ecore) metamodels may use multiple-class inheritance. The EMF[2] support uses distinct interface and implementation classes so that the interfaces accurately model the multiple inheritance. The implementation classes can only inherit one implementation and so the auto-generation re-implements all additional inheritances to create the illusion of true multiple-class inheritance.

Helper functions for OCL conformance cannot rely on Java's reflection capabilities so custom metadata may be required.

jfOCL may choose to prohibit multiple inheritance and not support OclAny or OclVoid.

5.5 Null

OCL's `null` is very similar to `null` in Java.

Difficulties can arise if the implementation of a type such as an enumeration fails to provide for a null value.

5.6 Operation

A distinct Java function can be provided for every OCL operation. This makes it more likely that OCL semantics are used throughout, but incurs a significant coding effort. It is therefore desirable to use pre-existing Java implementations wherever possible with the occasional wrapper to ensure appropriate semantics.

There are however some notable function/operation inconsistencies.

5.6.1 Boolean Short-Circuiting. Many languages provide a short-circuit capability for Boolean operations whereby in, for instance `a && b`, no attempt to evaluate `b` occurs if `a` is false. If evaluation of `a` crashes, the overall evaluation crashes.

OCL specifies commutative Boolean operations, which allows the OCL evaluation of `a and b` to evaluate its inputs in any order or concurrently. If the earliest input evaluation crashes and is `invalid`, but then the later input evaluation is `false`, the crash of the earlier term is 'uncrashed'.

This very different semantics prohibits the simple transliteration of `and` to `&&`. A helper operation can support the OCL-and semantics in Java. Since the helper requires `invalid` input values, the caller must ensure that any Java exceptions are caught.

The OCL validity analysis [17] enables the inconvenient commutated short circuit use case to be detected and rewritten so that regular Java exception semantics can be used.

5.6.2 Indexes. Java follows the tradition of implementation languages with arrays and lists using `0`-based indexes. Conversely OCL follows the specification tradition of `1`-based indexes. Java operations that take an index argument have an easily resolved semantic incompatibility. The transliterator must convert the index base at some point taking care to convert exactly once.

5.6.3 Implicit Operations. OCL uses dot navigation for object navigation and arrow navigation for collections. Additionally, OCL defines an arrow navigation for objects that is equivalent to an invocation of `oclAsSet` and a dot navigation for collections that is equivalent to invocation of a `collect()` iteration. These implicit invocations are expanded by the OCL analyzer as it creates the

AS. A transliterator that ignores the AS must replicate the analysis that in some cases must accurately resolve types to distinguish whether for instance `myAggregate.name` is an implicit collect of a user collection type or a simple navigation of a regular class.

Java does not (yet) have null-safe navigation and so an implementation of OCL that provides null-safe operators [15] must again replicate the synthesis that the analyzer performs when synthesizing the ASG.

5.7 OCL-specific

Some OCL concepts have no direct Java counterpart.

5.7.1 Tuple. In Java a primary type such as `MyClass<T>` is declared once. Its specialization such as `MyClass<String>` may be 'declared' many times with each re-'declaration' sharing the same specialized type.

OCL does not declare types at all; types are declared elsewhere in some metamodel. Usage of the collection types such as `Bag(String)` behave in a similar way to a Java specialization, perhaps confirming that the magic behind the underspecified `T` that parameterizes them should be modeled as a UML TemplateParameter.

An OCL Tuple type has no unique declaration, rather each repetition of the same set of {name,type} pairs is the same type. Java has no corresponding dynamically constructed type, so a code generator must pre-analyze the OCL to aggregate all Tuple declarations and create a conventional Java class for each distinct Tuple type.

Alternatively jfOCL may prohibit Tuples.

5.7.2 oclIsKindOf(), oclType(), reflection. As noted in Section 5.4.8 OCL uses conformance rather than inheritance and while the concepts are similar there are subtle differences that `oclIsKindOf()` and `oclIsTypeOf()` must respect.

OCL's `oclIsKindOf()` is similar to `instanceof` in Java, and `oclType()` is similar to Java's `getClass()` however they operate on different type systems. The OCL calls use modeled types where Java uses Java classes. Sometimes these are the same, but more often they differ. In Ecore there is typically a distinct interface and implementation Java class for each modeled class. Any multiple inheritance in the model is respected by the interfaces but only emulated by the classes.

Helper functions for `oclIsKindOf()` and `oclType()` must respect the OCL type conformance relationships. An implementation can provide tables to facilitate efficient access. These tables may support full reflective capabilities rather than the very partial form specified for `oclType()`.

5.7.3 oclIsNew() and @pre. OCL's `oclIsNew()` and OCL's `@pre` may only be used in post-conditions in order to contrast the system state on entry and exit states of an operation. This has no counterpart in Java and implementation of a helper is hard.

This is a difficult corner of the OCL specification that jfOCL may reasonably choose to skip.

Eclipse OCL[3] with an emphasis on execution has no execution capability for `oclIsNew()` or `@pre`. No user has reported this bug.

USE[12] has an emphasis on simulation and a custom model framework with built-in `oclIsNew()` and `@pre`. This is able to support a 'filmstrip' [1] of system states.

In principle, the support for two system states, may require the operation entry code to have a complete copy of the system state. This can be difficult and expensive, but is only necessary when the post-condition's `@pre` access involves a cascade of collection operations that are too complex to analyze. In practice, the `@pre` accesses are often trivial making an on-entry copy simple.

5.8 Model / UML-specific

OCL provides some unique capabilities associated with its support for models.

5.8.1 allInstances(). OCL's `allInstances()` has no counterpart in Java. The deficiency can be remedied by a helper function that traverses the transitive closure of the containment and reference relationships starting from a single seed model element. This assumes that the Java objects have sufficient context to reflect on the metamodels that define the Java classes. EMF provides this capability.

A naive helper implementation may perform a total model search for every `allInstances()` call. A better implementation may start with a metamodel analysis of the OCL to determine the typically small number of calls that can occur followed by a model analysis to determine all required `allInstances()` in one model traversal. This may be performed as part of the initial model loading or lazily when the first call occurs.

5.8.2 Unnavigable Opposites. OCL's navigation to unnavigable and sometimes implicitly named opposites again has no counterpart in Java and like `allInstances()` can be remedied by a helper function that traverses the whole model searching for the solutions. Again, like `allInstances()`, a naive implementation may be very costly, but a better implementation may use a metamodel analysis to discover all the `X::y` unnavigable opposites actually in use so that a single traversal of the model can populate a cache of `{X -> X::y}` for constant-time cost during execution. These analyses can be performed at the same time as the `allInstances()` search.

5.8.3 Stereotype. Normal OCL usage involves developing a typically M1 metamodel that constrains the corresponding M0 instances. OCL constraints are written against the metamodel and evaluated on the model.

UML Stereotypes provide a confusing ability to merge extensions into the M1 metamodel classes. The extensions are defined in the M2 metametamodel. OCL constraints are written against the metametamodel and evaluated on the M1 metamodel to verify consistent extension usage.

When a Stereotype provides an additional property `P::p`, it is accessed by an M2 constraint as `aY.extension_P.p`. However at M0, the Stereotype is logically folded into the extended instance and so the access may be `aY.p`. How this folding actually occurs is dependent on the model support. For EMF-based UML2Ecore, the properties are folded in as regular properties subject to spelling adjustments to avoid clashes. For UML, the run-time is a little vague, but probably the property is not folded, rather the missing StereotypeApplication metaclass that lies behind the XMI magic, needs to be reified to support accessing the stereotype application as a sub-object. Whatever, the transliterator must choose and parameterize an appropriate helper to perform the property access.

6 CODE GENERATION SYNTHESIS

We have identified the near-impossibility of a sentence-by-sentence transliteration and so we now describe a few synthesis considerations for a code generator. The code generator uses the Abstract Syntax Graph as input and so has an accurate representation of what needs to be done; all intermediate expression types are known, synthetic operations such as safe navigation and implicit collect have been elaborated, implicit iterator variables have been created and names resolved appropriately.

6.1 Name Synthesis

The initial implementations of new tools are rarely bug free, consequently users as well as developers may need to study the generated code to understand what has gone wrong, and perhaps to find a workaround for a bug. This study is eased if the code is pleasantly formatted and if the names are easily related to names that are already familiar to the user.

Mindless re-use of names from metamodels and OCL can fail through use of reserved words, multiple use of the same name or differential name occlusion between Java and OCL.

These failures can be resolved by accurately tracking the names visible in each synthesis scope and generating names that avoid clashes. The user's names can provide hints to which prefixes or suffixes are applied to clarify roles and avoid clashes.

6.2 Loop Synthesis

It was noted in Table 2 that while OCL's `forall` looks rather like Java's `for`, a more complex text template is needed. The following text template can be used for iterations other than `closure()` that merits a dedicated handler.

```
declare-result;
{
    declare-and-initialize-iterator(s);
    declare-and-initialize-accumulator;
    while (true) {
        if (iteration-done) {
          assign-result-from-final-accumulator-value;
          break;
        }
        advance-iterator(s);
        execute-iteration-body-and-assign-to-accumulator;
    }
}
```

For the common single iterator iterations, it is well worth generating inline Java to reduce control overheads. For less common iterations such as `sortedBy` or multi-dimensional iterations, it may be appropriate to realize the text template by virtual function calls to a polymorphic loop manager and to pass the body as an anonymous function.

6.3 Code Generator Model and Optimization

The Abstract Syntax Graph is a good model of the OCL semantics, but somewhat distant from the Java semantics. A Code Generator may therefore choose to transform the OCL ASG into a more Object-Oriented CGM (Code Generator Model) from which generation of Java or C or some other language is much easier. In the ASG, every expression term can return a value. In the CGM, every not-inlined expression term may return a Static Single Assignment variable facilitating a synthesis that can always refer to a variable. The expression terms include many extra terms to cast to known types, guard against nulls and convert between representations. The substantial support code to provide conformance tables, or implicit opposite caches can be reified in the CGM reducing the complexity of the eventual CGM to text model-to-text transformation. For Eclipse OCL a further CGM to JavaModel to JavaText intermediate is planned.

Once a sensible CGM is in use, support for optimizations such as Common Subexpression Elimination, Constant Folding and Loop Hoisting follows naturally. The Validity Analysis [17] is a significant extension to Constant Folding that once integrated should identify many more candidates for Dead Code Elimination.

Concept	Eclipse OCL	jfOCL
iterator shortforms	supported	prohibited
Java reserved words	supported	prohibited
invalid	supported	crash always
Bag	supported	prohibited
Integer/Real	polymorphic	distinct
OclAny	supported	prohibited
OclVoid	supported	prohibited
UnlimitedNatural	supported	prohibited
oclIsKindOf()	supported	prohibited
oclType()	supported	prohibited
and/or	commutative	short-circuit
allInstances()	supported	prohibited
unnavigable opposites	supported	prohibited
qualified navigation	supported	prohibited
qualified associations	supported	prohibited
qualified navigation	supported	prohibited
Tuples	supported	prohibited
stereotypes	supported	not supported
@pre, oclIsNew()	not supported	prohibited

Table 2: Simplification Options

7 JAVA-FRIENDLY OCL

While describing the various challenges, we have identified deviations from specified OCL that a hypothetical jfOCL might adopt in order to avoid the complexity of a full OCL analysis and subsequent synthesis.

A potential OCL tool vendor may reasonably ask why it is so hard to develop OCL tools. We have provided some of the answers

- poor quality of the specification
- lack of standard surrounding model access
- subtly distinct semantics from Java

Each of the simplifications summarized in Table 2 enables a jfOCL to be more like Java making the jfOCL2java transliteration easier, but if too many are adopted there must be a question as to whether too much of OCL has been discarded. Many of the original advantages of OCL such as compact collection cascades and iterations have been diminished by the advent of Java streams and lambdas. Perhaps it would be better to devote the development effort to providing a better library of helpers for regular Java use and possibly some annotation magic to facilitate model-oriented use.

Revisiting Figure 1, the outstanding advantage of OCL is its potential for provable correctness. Discarding too much of OCL may not only reduce the richness but also undermine the correctness. Once this happens, OCL becomes pointless.

8 RELATED WORK

This paper revisits Willink [14] in which "An extensible OCL Virtual Machine and Code Generator" was presented. It probably remains as the only extensible code generator for 'full' OCL (and QVTc and QVTr). Ten years on, the emphasis on a VM is clearly misleading hype. A VM suggests a JVM-like byte code interpreter, which is possible, but something of a premature optimization. The XMI serialization of the OCL ASG is more appropriate. Compacting it to byte code while saving on file size would needlessly lose readability when debugging. In this paper we draw on 10 years evolution to explain why a 'full' code generator is hard so that other authors can fully appreciate the challenges they face.

Many authors have provided a partial OCL code generator demonstrating good characteristics aligned with some research goal. Omission of OCL facilities such as oclIsNew(), @pre, allInstances(), 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[13] 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[11] 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[7] 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.

More recently Lano[6] has used some powerful rules to create a remarkably small transliterator from OCL to Swift. It is not clear how this approach avoids the need for an overall analysis, how extensive the library of helpers is, or how the reference and ambiguity resolution of a traditional OCL analyzer is replicated.

9 CONCLUSION

We have identified how the 'friendliness' of OCL's shortforms requires significant semantic analysis to enable an appropriate exposition in the semantics of another language.

Since the results of this analysis are available in the OCL Abstract Syntax, we conclude that synthesis of code from the semantically resolved AS can be accurate. In contrast, full transliteration is only possible if the standard CS to AS conversion is duplicated within the transliterator.

We have enumerated many of the OCL facilities such as opposite-navigation, unbounded-integers and invalid that are inconvenient to code generate so that OCL-like implementers can determine which inconveniences to omit in their 'Java-Friendly OCL'.

REFERENCES

[1] Nisha Desai, Martin Gogolla, and Hilken Frank. 2017. Executing Models by Filmstripping: Enhancing Validation by Filmstrip Templates and Transformation Alternatives. In *Workshop Executable Modeling, EXE 2017.*

[2] Eclipse EMF Project. [n. d.]. https://projects.eclipse.org/projects/modeling.emf.emf.

[3] Eclipse OCL Project. [n. d.]. https://projects.eclipse.org/projects/modeling.mdt.ocl.

[4] Marina Egea, Carolina Dania, and Manuel Clavel. 2010. MySQL4OCL: A Stored Procedure-Based MySQL Code Generator for OCL. In *OCL 2010: Workshop on OCL and Textual Modelling.* Models 2010, Oslo.

[5] Florian Heidenreich, Christian Wende, and Birgit Demuth. 2007. A Framework for Generating Query Language Code from OCL Invariants. In *Ocl4All: Modelling Systems with OCL.* Models 2007, Nashville.

[6] Kevin Lano. 2021. In *OCL 2021: Workshop on OCL and Textual Modeling.* https://oclworkshop.github.io/2021/papers/ocl2021_paper_3.pdf

[7] Rodion Moiseev, Shinpei Hayashi, and Motoshi Saeki. 2009. 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).* 650–664.

[8] Object Management Group 2005. *Unified Modeling Language, Infrastructure* (version 2.03, OMG Document Number: formal/2005-07-05 ed.). Object Management Group. http://www.omg.org/spec/UML/2.0/

[9] Object Management Group. 2014. Object Constraint Language, Version 2.4. https://www.omg.org/spec/OCL/2.4/PDF.

[10] Object Management Group 2017. *Unified Modeling Language, Infrastructure* (version 2.5.1, OMG Document Number: formal/17-12-05 ed.). Object Management Group. http://www.omg.org/spec/UML/2.5.1/

[11] Ayatullah Jibran Shidqie. 2007. *Compilation of OCL into Java for the Eclipse OCL Implementation.*

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

[13] Claas Wilke. 2010. *Java Code Generation for Dresden OCL2 for Eclipse* (technische universitat dresden ed.). http://dresden-ocl.sourceforge.net/downloads/pdfs/gb_claas_wilke.pdf

[14] Edward D. Willink. 2012. An extensible OCL virtual machine and code generator. In *OCL 2012: Workshop on OCL and Textual Modeling.* https://doi.org/10.1145/2428516.2428519

[15] Edward D. Willink. 2015. Safe Navigation in OCL. In *OCL 2016: Workshop on OCL and Textual Modeling.* Ottawa. http://www.eclipse.org/modeling/mdt/ocl/docs/publications/OCL2015SafeNavigation/SafeNavigation.pdf.

[16] Edward D. Willink. 2020. Reflections on OCL 2. *The Journal of Object Technology* 19 (01 2020), 3:1. https://doi.org/10.5381/jot.2020.19.3.a17

[17] Edward D. Willink. 2021. A Validity Analysis to Reify 2-valued Boolean Constraints. In *OCL 2021: Workshop on OCL and Textual Modeling.* http://ceur-ws.org/Vol-2999/oclpaper1.pdf