Support for OCL Libraries and Static Features

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

ABSTRACT

Libraries provide a powerful re-use capability allowing developers of one application to exploit the developments of others. Sadly, OCL has no first class library capability and attempts to use available capabilities have not led to any re-usable libraries. Problems include lack of support for imports, foreign language calls, object creation, maps and inadequate specification of static features. We combine previous and new resolutions prototyped in Eclipse OCL to make libraries feasible.

KEYWORDS

OCL, Library, Native Call, Code Generation, Static Feature

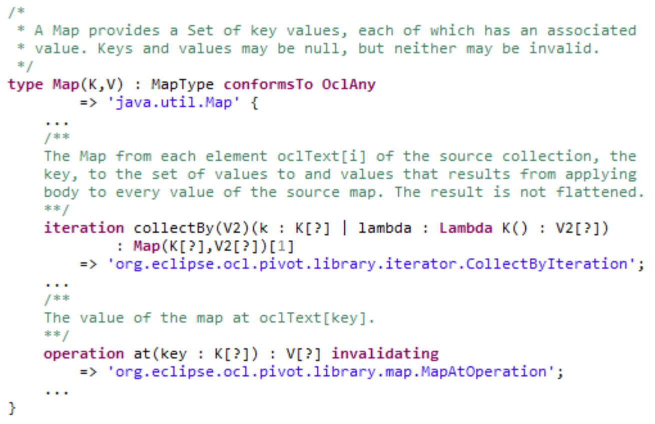
ACM Reference Format:
Edward D. Willink. 2022. Support for OCL Libraries and Static Features. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3550356.3561540

1 INTRODUCTION

Libraries provide a powerful mechanism for providing re-use. The extreme polymorphism of Java's Object and Collection classes and the consequent ease of use demonstrates how libraries can create a virtuous adoption cycle. In contrast, C++ was overtaken as its Standard Template Library arrived over ten years late and was compromised by a need to support bare arrays as collections. In further contrast, OCL has only a built-in library and few users.

The obvious utility of libraries has prompted authors [1],[3] to query where the OCL math library is. There is no answer. In this paper, we review the inadequate support for libraries and then provide the technology to support them. We will find that the non-Object Oriented style of typical mathematical operations requires the semantics of static features in OCL to be resolved.

OCL 2.0 [8] attempts to formalize the intuitive obviousness of OCL 1.x with the aid of models. Unfortunately the formalization never progressed beyond the draft stage and so whenever we look too closely at OCL 2.0 we find incompleteness and inconsistency [18]. The Abstract Syntax model omits critical classes such as `VariableDeclaration`. There is no model of the library, just many well-intentioned partial specifications.



```
/*
 * A Map provides a Set of key values, each of which has an associated
 * value. Keys and values may be null, but neither may be invalid.
 */
type Map(K,V) : MapType conformsTo OclAny
        => 'java.util.Map' {
    ...
    /**
    The Map from each element oclText[i] of the source collection, the
    key, to the set of values to and values that results from applying
    body to every value of the source map. The result is not flattened.
    **/
    iteration collectBy(V2)(k : K[?] | lambda : Lambda K() : V2[?])
            : Map(K[?],V2[?])[1]
        => 'org.eclipse.ocl.pivot.library.iterator.CollectByIteration';
    ...
    /**
    The value of the map at oclText[key].
    **/
    operation at(key : K[?]) : V[?] invalidating
        => 'org.eclipse.ocl.pivot.library.map.MapAtOperation';
    ...
}
```

Figure 1: OCLstdlib model example.

In [14], Eclipse OCL [5] introduced a model and grammar for the OCL Standard Library supported by an Xtext editor. The library model declares the library types and their features including template parameters.

Figure 1 shows part of the declaration of the `MapType` with `K` and `V` template parameters. An iteration with a `V2` template parameter and an operation are also shown. The operation demonstrates the fairly strong modeling identifying that the templated argument and return may be null. Additionally the `invalidating` keyword indicates that the result may be `invalid`. The iteration declares its potentially null parameter followed by its body typed as a lambda-expression[1]. The return is a non-null map of potentially null keys and values.

The figure also demonstrates using `java.util.Map` for the native implementation of the type, and using a custom helper class for each operation and iteration. The helper classes use polymorphism to dispatch `OperationCallExp::referredOperation` rapidly.

Once the OCL Standard Library was modeled, it was possible to separate the OCL Language from the OCL Standard Library allowing a user to provide an alternate or extended library implementation. It was also possible to define additional libraries. A few JUnit tests demonstrated the principle within Eclipse OCL, however an attempt to write a tutorial to show users how to exploit the potential of an additional library demonstrated that the implementation was not ready for prime time[2].

We first review the ergonomic issues to be solved in Section 2 then in Section 3 we examine a few examples that motivate exploiting static features whose semantics we examine in Section 4. In Section 5 we describe the status of this work and in Section 6 we

[1]The bodies of iterations are lambda expressions in all but name.
[2]https://bugs.eclipse.org/bugs/show_bug.cgi?id=415146

take static properties further. Then in Section 7, we look at other work, before concluding in Section 8.

2 LIBRARY ERGONOMICS

In order to make a library useful we need to solve four problems for the user.

- import the library of declarations
- specify the library declarations
- implement each library declaration
- invoke a specific library declaration

2.1 Import

The OCL specification [11] defines a language that can usefully constrain models, but does not specify where the models come from. This provides considerable flexibility but has prevented vendors from providing a general solution. Each tool has its own proprietary solution to solving the relationship between the OCL programming and the models with their metamodels.

Complete OCL comes closer to being complete in so far as the specification defines a self-standing document with a grammar that can complement a model. However a practical implementation must either add some form of import statement to allow the Complete OCL document to reference its complemented models, or operate in a pre-loaded context with ready to-go models.

A similar import statement or pre-loading will be needed to exploit a library of declarations. This should be just a variation of a proprietary facility that we do not need to address further here. Eclipse OCL supports an import such as:

```
import 'https://git.eclipse.org/r/plugins/gitiles
  /ocl/org.eclipse.ocl/+/refs/heads/master/plugins
  /org.eclipse.ocl.pivot/model/Types.ocl'
```

2.2 Declare

A Complete OCL document, a UML model or Eclipse OCL's Standard Library model provide adequate mechanisms for declaring many, but perhaps not all, declarations. A minimal `maths.ocl` supporting just the `tan` operation could be declared as[3]:

```
  package maths
  context Real
  def: tan() : Real = ...
  endpackage
```

2.3 Implement

OCL is probably Turing complete, see Section 7.1, so it is in principle possible to implement any library functionality in OCL and embed that implementation as the body of a Complete OCL declaration.

However few users will be keen to transliterate existing implementations from their favorite language in order to use them within OCL. We therefore have a vicious circle whereby the lack of any OCL libraries imposes a significant barrier to development of any OCL libraries.

Even if a good algorithm for `tan` is available, there may be Intellectual Property issues that prevent its re-use. The transliteration must be tested for interesting corner cases. Transliteration alone is not sufficient.

OCL needs a mechanism to enable re-use of an implementation already available in a foreign language.

2.4 Invoke

If the imported library declarations integrate well with other OCL declarations, the existing call capabilities should be re-usable.

```
 2.5e1.tan()
```

However when we come to implement the library declaration we find that we have just moved the problem sideways; we still need a mechanism to invoke a foreign language implementation. (And once we provide a foreign language call, we don't need a library at all.)

A generalization of an existing solution can solve this call problem too. The existing problem arises when the surrounding OCL context fails to provide an 'import' for MyPackage as in:

```
    ...oclIsKindOf(MyPackage::MyClass)
```

The existing solution exploits the `_'...'` escape mechanism for identifiers with awkward characters. A URI-qualified name can be specified as:

```
    ...oclIsKindOf(
    _'http://org.my.spec/2022#MyPackage'::MyClass)
```

We can generalize this to exploit a Java String function that the OCL String type does not support as:

```
thisString.
  _'java:java.lang.String.compareToIgnoreCase'
    (thatString)
```

The escaped name specifies a `java` language scheme followed by a Java-specific `java.lang.String.compareToIgnoreCase` fully qualified name.

The names are unpleasantly long, requiring multiple lines for this paper, but at least they can work for object methods with compatible types.

Once we support the invocation of arbitrary Java operations, we undermine the guarantee that OCL is side-effect free. The author of a foreign operation call is ultimately responsible for guaranteeing that the called operation is side effect free. Section 6.3 discusses enforcing the guarantee.

3 EXAMPLES

We have so far considered `tan` as an example. We will now change to looking at the syntactically more challenging pow before moving onto look at Complex numbers and then `Employee::uniqueId()`.

3.1 Math.pow

Most languages support a functional call of `tan` as `tan(2.5e1)` or `tan 2.5e1` rather than the Object-Oriented `2.5e1.tan()` which can be supported by declaring a regular `Real::tan()` operation. This OO style is strange if we want `pow(2, 4)` rather than `2.pow(4)`.

An OO style is not actually necessary since OCL 2.2 [9] added support for a `static` keyword when declaring operations and properties.

We may therefore provide a minimal `maths.ocl` as

[3]... denotes functionality that has been omitted in order to focus on a relevant issue.

```
package maths
context Real
static def: pow(mant : Real, expt : Real) : Real
  = _'java:java.lang.Math.pow'(mant, expt)
endpackage
```

This can support the call from regular OCL as

```
... maths::Real::pow(2, 4) ...
```

If the proprietary import mechanism exposes the contents of the maths package, the maths:: qualification and perhaps even the Real:: qualifications can be omitted giving a more optimal exposition for the call.

While OCL 2.2 added the static keyword to the grammar, it is hard to characterize the change as more than a me-too catch-up to align with UML static features. Very little run-time semantics was provided for OCL, and there is little to infer from UML [13]. Section 7.5.10 of the OCL specification provides an example that we shall see in Section 3.3 is not fully implementable in OCL[4]. Whether and how static is supported is down to the enthusiasm of the OCL tooling implementer.

It may be noted that Ecore [4] has no support for static features, so that if a UML model with static feature is converted using UML2Ecore, the static keyword is ignored.

Using an OO style:

```
  ... 2.pow(4) ...
```

may workaround the absence of support for static in the grammar, but the underlying Java function is static so internal support for foreign static calls is unavoidable.

For the semantics of static operations, intuitive common sense seems adequate. For static properties we need to clarify the semantics in respect of when a 'new' static instance is created and initialized. See Section 4.

Once we have a foreign call syntax, the user has a choice. The foreign calls can be used directly avoiding any need for a library at the expense of some clumsiness in the call expressions. Alternatively the foreign calls may be hidden within a library that provides friendly declarations that delegate to the foreign call.

In practice development/prototyping may use the direct call and evolve to use the more readable library delegation.

3.2 Complex

The foreign call delegation for pow can be appropriate when the types are simple and when the implementation is too complicated to re-implement.

But consider a Complex class, which strangely Java does not provide. Even if there was a Java implementation, we might choose to re-implement since Complex is rather simple. Except that we cannot, since, in general, addition of two complex numbers creates a new complex number. OCL provides no new capability since construction of a new object creates a side effect in the memory system. This can be seen by considering how a new capability might work:

```
new Complex{real=2,imag=3} = new Complex{real=2,imag=3}
```

[4]Diligent readers of the OCL specification may find a StaticValue class that is nothing to do with the static declaration of a feature. Rather it is a confusing throwback to when UML 1.x referred to the embedded DataType-typed attribute slots of an instance as its static structure.

Obviously two same-valued complex numbers should be equal; this is the UML DataType semantics. But if Complex is a Class, each call to new creates a new instance; they should therefore be different in accordance with UML Class semantics. (Java provides alternative = and equals facilities that allows a free and confusing choice of DataType or Class semantics.)

The problem with new was discussed at the Aachen workshop [2] where the helpful term 'shadow' object was coined for the solution [16].

In principle, let all possible instances of all possible classes have a permanent shadow existence, so that whenever the Tuple syntax is used to 'create' an object, the appropriate shadow instance is returned avoiding the side effect. Rewriting our example:

```
Complex{real=2, imag=3} = Complex{real=2, imag=3}
```

the two creations each return the same immutable shadow object and so the equality is satisfied.

In practice, creation of all possible instances of all possible classes will run out of time and memory long before any useful computation can be performed. Each required shadow instance must therefore be created lazily.

This shadow object 'creation' is exactly what is needed to enable a library to support the operations of a Complex type.

```
context Complex
def: real : Real
def: imag : Real
static def: new(real : Real, imag : Real) : Complex
  = Complex{ real = real, imag = imag }
static def: add(x1 : Complex, x2 : Complex) : Complex
  = Complex{
      real = x1.real + x2.real,
      imag = x1.imag + x2.imag
    }
```

3.3 Unique id

Section 7.5.10 of the OCL specification provides half an example demonstrating how the static Employee::uniqueID operation can be used to provide a unique ID for each Employee.

```
context Employee::id : String
  init: Employee::uniqueID()
```

Lazy (Java) Solution.

```
private static int counter = 0;
public static String uniqueID() {
    return Integer.toString(counter++);
}
```

As shown above, Employee::uniqueID can easily be realized in Java by defining a static counter for the IDs and incrementing the counter for each invocation.

An alternative eager approach in which all IDs are allocated together is awkward since it requires access to all instances that may need IDs. This includes those instances that are yet to be created.

Eager (OCL) Solution.

```
static context Employee::insts : Sequence(Employee)
  init: Employee::allInstances()->asSequence()
```

```
context Employee::uniqueID() : String
  init: insts->indexOf(self).toString()
context Employee::id : String
  init: Employee::uniqueID()
```

The lazy approach cannot be written in OCL since the updates to the static counter constitute a side effect that is illegal in OCL. The eager approach above is possible since `allInstances()` can find the instances and since the lack of side effects guarantees that no later creation of instances can occur.

The static `Employee::insts` property is initialized once with a sequence of all `Employee` instances. Thereafter each access of the `Employee::uniqueID()` operation is able to compute a distinctive `self`-dependent value by searching for the index of `self` in `Employee::insts`. The derived `Employee::id()` property delegates to `Employee::uniqueID()` for the first access. Subsequent accesses should benefit from a cache.

It seems unreasonable that the simple efficient lazy approach cannot be written in OCL. The semantics of static OCL features is unspecified, so there are opportunities for a resolution. Perhaps the initializer for a static property should be re-evaluated for each access. Alternatively the semantics could be refined so that the `init` initializes on the first access and a `der` updates on subsequent accesses. This would allow the count to evolve, but an unstable static property seems deeply suspect.

It also seems unreasonable that further instances cannot be created after some unique IDs are allocated. But for pure OCL, this is not a problem since OCL has no ability to assign or mutate; the counter with its incrementing value cannot exist. Once OCL is embedded, perhaps in QVTo [12], the imperative capabilities allow successive assigns and so the lazy approach can work. Alternatively when embedded in a strongly declarative context such as QVTr, all instances exist before the 'atomic' transformation completes and so it is just a scheduling challenge for the tooling to assign the IDs to the output in a timely fashion that satisfies the write/read dependencies. For pragmatic OCL enriching a model, it is the modeling environment's responsibility to reset all caches before each execution of some OCL-defined functionality; `allInstances()` must be 'recomputed' each time the modeling environment makes a significant change to the system state.

Unfortunately the `indexOf()` in our solution contributes to a quadratic implementation cost. We can linearize the cost by using a Map [17].

```
static context Employee::inst2id : Map(Employee,String)
  init: let insts = Employee::allInstances()
                  ->asSequence()
        in insts->collectBy(e with i | i.toString())
context Employee::uniqueID : String
  init: inst2id->at(self)
context Employee::id : String
  init: Employee::uniqueID()
```

At start up, or first access, the `Employee::inst2id` initialization computes the sequence of `allInstances()` to the `insts` let-variable. The `collectBy()` collection-to-map iteration then populates the map result from an iteration over all the instances. The iteration declares, but does not directly use, the primary element iterator `e`. It also declares the co-iterator `i`, which provides the index of the primary iterator in the ordered source collection. The co-iterator provides the same value as `insts->indexOf(e)` without the need to re-compute what the iteration evaluation already 'knows'. Each map entry is keyed by the iteration element `e` and maps to the iteration body `i.toString()`.

We can see that provision of globally coherent static facilities is much easier when each per-element property can be cached as an entry in a static global Map.

4 FEATURE SEMANTICS

The OCL specification does not define the semantics of static features, perhaps because the semantics is too obvious. But is it? We can answer this by reviewing the semantics of non-static features for which the specification is rather thin.

OCL supports two kinds of feature:

- Regular features declared in a UML / Ecore / ... metamodel
- Additional features declared in a Complete OCL document

4.1 Regular Features

Regular features are provided and maintained by the modeling environment to which the OCL contributes. When objects are created/deleted or properties initialized/updated is nothing to do with OCL. The modeling environment may implement and must refresh caches as appropriate.

The modeling environment may invoke an OCL expression when initializing a property. How an OCL expression evaluation accesses a slot is unspecified. But obviously it must route the access through the modeling environment to ensure that any access protocols are observed.

A modeling environment may similarly invoke an OCL expression when evaluating an operation. How an OCL expression may invoke an operation is also unspecified. Logically it should do so via the modeling environment, which would allow the modeling environment to implement the virtual dispatch policy that is not specified by OCL[5]. However operations are modeled using an `ExpressionInOCL` to provide bindable parameters so there is a clear intent that OCL should be able to invoke at least OCL operations directly. Eclipse OCL does this and applies a Java-like virtual dispatch.

4.2 Static Regular Features

For regular features, the `static` keyword in OCL may just echo the corresponding declaration in UML to ensure that a Complete OCL complement complements the correct feature. (Ecore does not support static features.)

With the modeling environment responsible for providing and maintaining the static features, the modeling environment is also responsible for their usage.

For OCL purposes, use of `staticFeature` can be regarded as just syntax sugar for `dummyObject.staticFeature` saving the user the need to provide a suitable `dummyObject` that will be ignored by the execution. Use of `self` within a static feature initializer or body should be a Well-Formedness Rule violation.

[5]For UML, the lack of a virtual dispatch specification is worked around by defining all the overrides as redefinitions.

4.3 Additional Features

A Complete OCL document may complement pre-existing operations and properties by defining missing bodies and initializers. It may also extend pre-existing classes with additional operations and properties. These additions are specified to behave as regular features. This is necessary to avoid new forms of `PropertyCallExp` and `OperationCallExp` in the Abstract Syntax, but with the additional features not actually part of a regular model, it is unclear where the target of a `PropertyCallExp.referredProperty` may be found.

It is not clear if the modeling environment is aware of the additional features, so it is unclear whether additional properties are lazily or eagerly initialized. There is no doubt that the additional operations are only invokable from OCL expressions within the transitive closure of Complete OCL documents that import the additional feature definition. Whether caches should be used is neither specified nor configurable.

4.4 Static Additional Features

OCL provides no mechanism to update a property value, so an additional static property is almost useless. It can only be a constant since nothing is able to change it. It cannot contribute useful system state. However the good practice of localizing magic constant values behind a symbolic name can be supported:

```
context Real
static def PI : Real = 3.14159265359
```

A static additional operation is also rather boring. It differs from a non-static additional operation by requiring no access to a `self` context, and no need to perform a virtual dispatch of overloads. Although boring, we have seen in Section 3 that static operations avoid the need for OO-style.

4.5 Summary

Static features have very limited utility. As with non-static features, the structural functionality is provided by the modeling environment. OCL is just a client of what is available.

OCL provides facilities to initialize properties and variables, but no ability to assign or re-initialize. In Section 6.2 we consider an opportunity that arises if the 'static' keyword is permitted for a Tuple part.

5 STATUS

The facilities described so far in this paper are all working as part of JUnit tests for the development version of Eclipse OCL [5]. Addition of support for interpreted execution was fairly straightforward. However enhancing the OCL2Java code generator justified a code refactoring. Each different style of operation and property access was handled by an appropriate branch of a substantial if-tree. The additional styles for static support and foreign access combined with a lack of Ecore support pushed this style of coding too far. The refactoring therefore reifies each different style via an appropriate `CallingConvention` class. This refactoring is still in progress.

Once this refactoring is tested and released, the library tutorial can finally be written and a maths library provided as a demonstration of the principles.

6 FURTHER WORK

The inability in Section 3.3 to provide the obvious implementation of `Employee::uniqueID` is disappointing, but perhaps there are solutions that do not stretch OCL semantics too far.

6.1 Lazy Maps

The solution in Section 3.3 uses a global map to provide overall coherence. A similar functionality could be provided if there was an additional operation for `Map(K, V)`.

```
operation lazyAt(key: K, init: Lambda K() : V[?]) : V[?]
```

If the map already contains an entry for `key`, the corresponding value is returned just like `Map::at(key)`. If there is no entry, one is created using the `init` lambda expression to define the value for the new key. The lazy OCL solution could then be:

```
static context Employee::inst2id : Map(Employee,String)
  init: Map{}
context Employee::id : String
  init: inst2id->lazyAt(self,
                      inst2id->size().toString())
```

The `Employee::inst2id` is initially empty, and each access to `Employee::id` uses `lazyAt` to provide a distinct value.

`Employee::inst2id->size()` can be accessed from anywhere, with unstable results depending upon `Employee::id` access to occurrence. There is therefore an observable side effect. This is not OCL.

6.2 Mutable Statics

Implementing the incrementing counter requires an ability to assign diverse values to a variable. This is possible in OCL, but only for the specific case of the result/accumulator of an `iterate` iteration:

```
myList->iterate(e; acc = '' | acc + ' ' + e.toString())
```

The multiple assignments to `acc` do not create a side-effect since the `acc` is private to the `iterate` and only accessible on a per-iterator basis in the body. The prevailing state of `acc` is not arbitrarily visible so the changes are not visible. `acc` is not part of the observable side-effect-free system state.

The example above gives a consistent result, for an ordered source, but could give diverse results if the source is a set. This is not a side effect; it is a lack of determinism for which [15] provides solutions.

If we could somehow generalize `iterate` to make each step independent, we could re-assign to our unique counter rather than `acc`, discount the unique counter from the observable system state and similarly discount inconsistent step sequencing as a lack of determinism.

There are other mechanisms of assignment in OCL

- let-variable initializer
- property / part initializer
- iterator initializer

but each is a single assignment. The assignment that occurs, when for instance a new let-variable is created within a loop, is not a re-assignment. Can any of these mechanisms be the basis of a re-assignment?

It was noted in Section 4 that the semantics of statics are poorly specified, so we could be more imaginative and allow private static parts or properties to be changed by constructors as in the following lazy OCL solution:

```
static context Employee::initTuple : Tuple(id : String)
  init: Tuple{                   -- Tuple constructor
   static count : Integer = 0,   -- private static part
    id : String = null          -- regular public part
  }
context Employee::id : String
  init: Tuple{                   -- Tuple constructor
    static count : Integer = count+1, -- private static
    id : String = count.toString()    -- regular public
  }.id
```

At start-up, `static Employee::initTuple` creates an initial `Tuple(id : String)` with a regular `id : String` part and a novel private static `count : Integer` part initialized to zero.

Subsequently, each first invocation of `Employee::id` initializes the employee-specific cache of the `Employee::id` property with a new `Tuple(id : String)`. The new Tuple re-initializes the novel private static `count : Integer` part with an incremented value. The regular `id : String` part is given a unique string value based on the previous `count`. The new tuple ceases to exist once its `id` part is accessed to initialize the `Employee::id` cache.

We call the static part a private part since it can only be accessed by an expression within the constructor, and its value can only be exported from the constructor via a regular part whose initializer accesses the static part. Re-assignment of the static part can only occur in the constructor, and can only be observed if the constructor uses the private part to initialize a regular part. However this does not guarantee no side-effects. We have to ensure that a re-execution can never produce a different result, which is impossible when the whole purpose is to have changes. We must therefore guarantee that execution only ever happens once. This is already guaranteed when the underlying implementation of an operation or property uses a cache to avoid re-execution. This is a very good practice, although not mandatory, for non-trivial operation bodies or property initializers. For static parts we need execution to a cache to be mandatory.

Construction of a shadow object re-uses the Tuple construction syntax and so could reasonably apply the same policy to maintain/exploit the statics of a regular class.

6.3 Side-effect Free

A foreign feature call was introduced in Section 2.4 enabling OCL expressions to exploit functionality already available in foreign languages. In practice re-use of Java operations is very desirable. However evaluation of an OCL expression must be side-effect free and so any invocation of a Java operation such as `List.add()` has a side-effect that may invalidate the use of OCL.

Introduction of a foreign language call is therefore a dangerous compromise between utility and validity. Can the danger be mitigated?

Compile-time analysis of the invoked functionality will not generally be possible since Java has too many side-effecting mechanisms and too many deep overload alternatives.

At run-time, comparison of the total system state before and after a suspect foreign call is technically possible, but could be unrealistically expensive. Less expensive in memory, but not time, could be a comparison of a hash of the total system state.

A less ambitious comparison of shallow rather than transitive state could be feasible to catch obviously unsuitable operations such as `List.add()`. A database of known bad operations could be gradually built-up, but progress would be slow since new entries only appear after a programming violation.

Realistically, observing the no-side-effect rule must be the programmer's responsibility. Configuring the `Operation::isQuery` property in a UML model is of course already a programmer responsibility.

7 RELATED WORK

The inadequacies of the OCL specification make it hard for authors to do more than express enthusiasm for the concept of an OCL library.

Baar [1] argues for a standard mechanism to support OCL re-use but offers no solution.

Cabot and Gogolla [3] observe that the specified OCL Standard Library is too large for new users, but inadequate for more experienced users. They identify the provision of OCL libraries as a Research Agenda item.

Lano [6] enumerates at least eight different OCL libraries many of which require an imperative extension to OCL.

Willink [14] introduces a model, grammar and editor for the OCL Standard Library, but as noted in Section 1 this has not matured sufficiently to support custom libraries.

At the 2016 OCL workshop, a lightning talk by Cabot and Gogolla called for a repository of OCL benchmarks, which could no doubt also accommodate OCL libraries.

7.1 Turing Complete

Mandel [7] provided an early assessment of the expressive power of OCL and cast doubt on its Turing completeness because an infinite WHILE loop was neither feasible nor breakable. However the authors overlooked the utility of Tuples.

An unbreakable iteration that computes some function `f(e, acc)` at each step:

```
...iterate(e; acc : Acc = ... | f(e, acc))
```

may be rewritten with a Tuple to be breakable:

```
...iterate(e; acc2 = Tuple{
                         break : Boolean = false,
                         acc : Acc = ...
                     }
    | if break then acc2
      else Tuple{
           break = ... ,
           acc = f(e, acc)
         }
    endif
  )
```

The data flow associated with the `break` part is very simple; initially false, unchanging once true, never reset to false. Tooling can easily recognize this to terminate promptly. Perhaps OCL should have a break value that may be returned during an ordered iteration to abandon the current and all future steps.

A forever loop is awkward in OCL 2.0, but it can be achieved by a function f() that recurses to collect the value of g() for each loop element:

```
def: f(x : Integer, r : Set(Result)) : Set(Result)
= f(x+1, r->including(g(x)))
```

This will obviously run out of stack in a naive implementation. As of OCL 2.3 [10] there is a closure iteration with which we can write a infinite loop.

```
1->closure(e | e+1)
```

The closure can be broken after computing something useful by:

```
OrderedSet{seed}->closure(e |
    if ... then e else e->including(...) endif)
```

For the harder case of a Sequence or Bag return, a first pass closure can compute the Sequence of Integers one per result followed by a second pass to collect the required result for each iteration index.

8 CONCLUSION

We have identified the inadequacy of the OCL specification as the primary impediment to provision of OCL libraries. In resolving the problems, we find that many facilities not clearly available in OCL 2.4 are necessary to make a library implementation possible.

- An import capability loads the library declarations.
- Static features facilitate typical non-OO usage such pow(2,4).
- Foreign operation calls allow direct re-use of e.g. Java.
- Shadow objects allow operations to return 'new' objects.
- Maps facilitate coherent all-instances functionality.
- Co-iterators eliminate gratuitous re-calculation.

and in future

- Mutable static parts support a typical counter idiom.

The introduction of foreign feature calls and the consideration of static features is new to this paper. Other facilities have been presented before, but their integration to make a custom library feasible is again new to this paper.

REFERENCES

[1] Thomas Baar. 2011. On the Need of User-defined Libraries in OCL. (01 2011). https://doi.org/10.14279/tuj.eceasst.36.447.451

[2] Achim Brucker, Dan Chiorean, Tony Clark, Birgit Demuth, Martin Gogolla, Dimitri Plotnikov, Bernhard Rumpe, Edward Willink, and Burkhart Wolff. 2013. *Report on the Aachen OCL Meeting.* http://www4.informatik.tu-muenchen.de/publ/papers/CKR+99.pdf.

[3] Jordi Cabot and Martin Gogolla. 2012. Object Constraint Language (OCL): A Definitive Guide, Vol. 7320. 58–90. https://doi.org/10.1007/978-3-642-30982-3_3

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

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

[6] Kevin Lano and Shekoufeh Kolahdouz Rahimi. 2022. OCL libraries for software specification and representation. In *22nd International Workshop on OCL and Textual Modeling (OCL 2022).* Montreal.

[7] Luis Mandel and María Cengarle. 1999. On the expressive power of pure OCL. 854–874. https://doi.org/10.1007/3-540-48119-2_47

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

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

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

[11] Object Management Group. 2014. Object Constraint Language, Version 2.4. formal/2014-02-03.

[12] Object Management Group. 2016. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. formal/2016-06-03.

[13] 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/

[14] Edward Willink. 2011. Modeling the OCL Standard Library. (01 2011). https://doi.org/10.14279/tuj.eceasst.44.663.673

[15] Edward Willink. 2018. *Deterministic Lazy Mutable OCL Collections.* 340–355. https://doi.org/10.1007/978-3-319-74730-9_30

[16] Edward Willink. 2018. Shadow Objects. In *18th International Workshop on OCL and Textual Modeling (OCL 2016).* Copenhagen.

[17] Edward Willink. 2019. An OCL Map Type. In *OCL 2016: Workshop on OCL and Textual Modeling.* Munich. http://www.eclipse.org/modeling/mdt/ocl/docs/publications/OCL2019MapType/OCLMapType.pdf.

[18] 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