OCL Collection Optimization - Part 3 - Cascades

Edward D. Willink[1,2,*,†]

[1] Willink Transformations Ltd, Reading England
[2] Eclipse Foundation

Abstract

Collections and iterations provide much of the power of OCL. The specification of the OCL Standard Library as operations and and iterations encourages implementation and execution using building blocks. We identify how this approach leads to inefficient execution. We show how providing a sounder formality for iterations and restructuring collections with co-collections enables functionality to be shared across cascades of collection operations.

Keywords

OCL, Collection optimization, Iteration optimization

1. Introduction

OCL [1] evolved to satisfy UML's need for textual model constraints where graphics was not appropriate. Unfortunately early tooling was either of insufficient quality or not used and so the OCL constraints that formed part of UML models prior to UML 2.5 contained hundreds of errors.

For UML 2.5 [2], Eclipse OCL [3] was used to remove all syntactical errors such as mismatched parentheses or the use of the OCL 1 Enumeration Literal syntax. Nearly all semantic errors such as inappropriate operators or operations were also resolved. No assessment of the functional appropriateness of the constraints was made.

During development of Eclipse OCL's Validity View, UML's OCL constraints were accidentally used directly on the UML metamodel. Execution at least started, but took far too long to make any useful observations of the utility of the constraints. Investigation of the unacceptable speed diagnosed that long cascades of collection operations contributed to very high order execution times. The manual transcriptions and caches of real UML tools avoid the high orders. Automated OCL tooling clearly needs improving to emulate what manual implementations do. But how? Finally inspiration struck.

In this paper we briefly review the collection innovations from part 2 [4]. In Section 2 we examine the traditional execution flow for a simple example and discuss its limitations and the remedy that a `gather-gather` optimization offers. In Section 3 we identify some idiomatic usages in the UML's OCL; one is resolved by a `gather-gather` optimization. The other motivates a `gather-search` optimization that we discuss in Section 4. In Section 5 we present results that show the potential benefits of the `gather-gather` optimization. Then we discuss Further Work in Section 6 and Related Work in Section 7. Finally we conclude in Section 8.

In part 2, we proposed to replace the implementation (but not specification) of the four collection kinds by a single basic sequence with an optional CoCollection in which support for the set and bag aspects could be lazily computed, cached and shared. We also proposed new declarative `gather` and procedural `search` iterations to overcome many limitations of `iterate` and so support rewriting of many iterations and operations in ways that we expoit in this paper.

OCL 2025 at STAF 2025, 10 - 13 June 2025 Koblenz, Germany
ed@willink.me.uk (E. D. Willink)
0000-0003-3124-4019 (E. D. Willink)

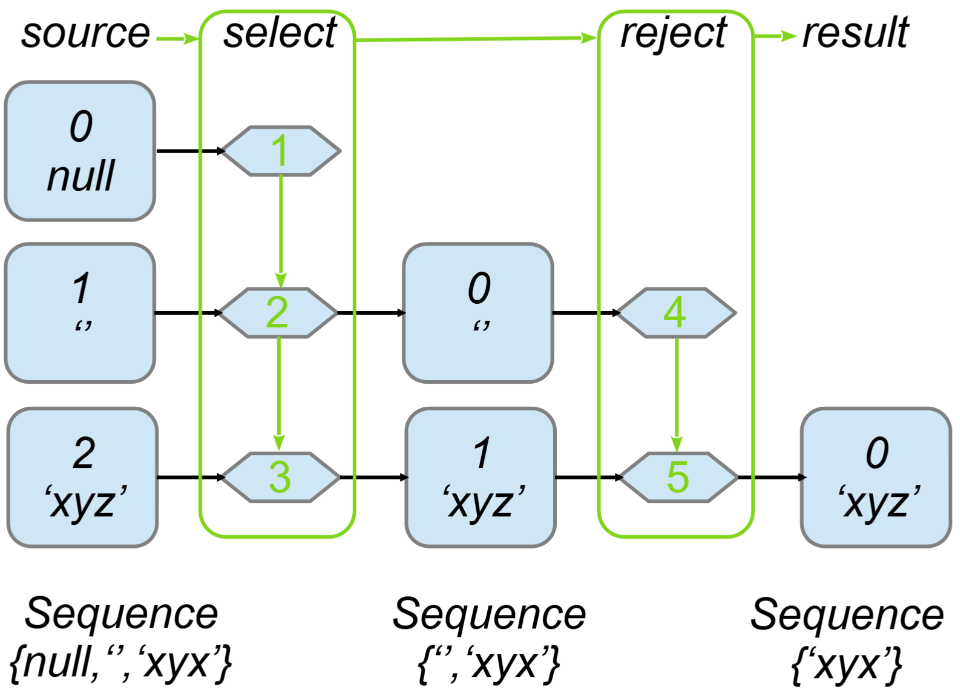

Figure 1: Simple Example (traditional approach)

2. Cascade Execution

The accidental use of UML's OCL identified collection cascades as a performance problem, so we will examine how collection cascade execution may be deficient and may be improved.

Throughout this section we will use a simple example for a NamedElement - a class with a `name` property.

```
source->select(name <> null)->reject(name = '')
```

A collection of `source` elements is successively pruned by first selecting those with a non-null name and then rejecting those with an empty name.

2.1. Traditional Execution Flow

A typical computation flow for our example is shown in Fig 1, using `Sequence{null, '', 'xyz'}` as the source value. In the figure shaded rounded rectangles denote a single collection element, labeled by its 0-based sequence index and its String value. The unshaded rounded rectangles identify a computational building block with lozenges inside for each actual computation. The lozenges are numbered in overall order of execution.

The `Sequence{null, '', 'xyz'}` `source` value at left is passed to the `select` iteration building block which processes each element in turn to produce the `Sequence{'', 'xyz'}` intermediate result. Control is then passed to the `reject` iteration building block so that each element of the intermediate result is processed to produce the overall `Sequence{'xyz'}` `result` at right. The vertical then horizontal order of evaluation is shown by green arrows and the 1..5 labels on the computation lozenges. This order of execution is very natural when an implementation is based on subroutines for each operation/iteration building block.

We would obviously like our execution to perform all the necessary computations while performing as few extra activities as possible. The processing in Fig 1 does not satisfy this desire. Each execution of `name <> null` and `name = ''` is clearly necessary, as is access to the `source`, and assignment of the `result`. But almost everything else is questionable bloat.

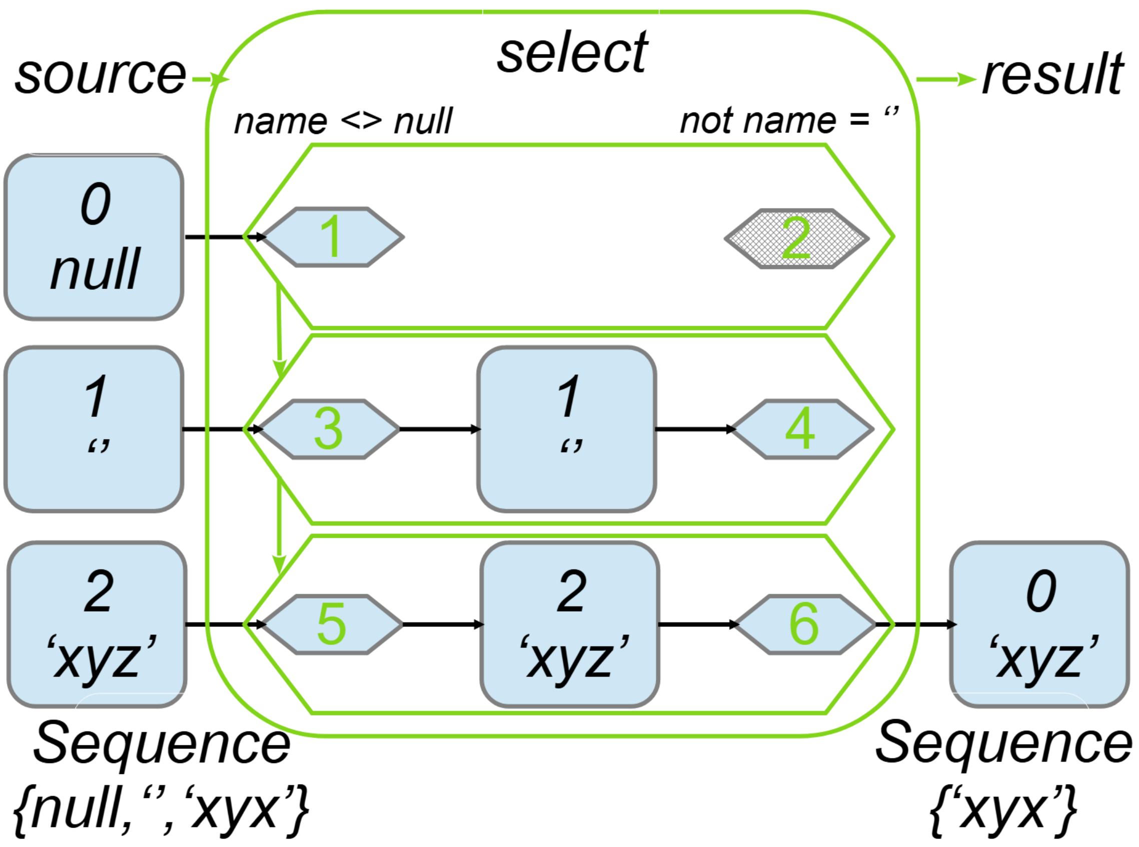

Figure 2: Simple Example (intuitive approach)

The execution creates an unnecessary intermediate result and traverses two collections for many of the result elements. (Worse, if the intermediates and results are maintained as OCL collections, an unnecessary OCL collection is churned for each iteration step; a fresh collection is successively created for `Sequence{}` and then `Sequence{''}` before the useful `Sequence{'','xyz'}` intermediate is passed.

2.2. Faster Execution Flow

It is intuitively obvious that our example could be rewritten as

```
source->select((name <> null) and not (name = ''))
```

Figure 2 shows the correspondingly better horizontal then vertical execution. The intermediate passed between computations is a single element rather than a collection. This element is easily accommodated by a hardware register avoiding the need to allocate, populate and traverse memory for an intermediate collection. (The second execution lozenge is hatched out since it has nothing to process.)

However the OCL specification provides no formality for this intuition. It is debatable whether the single-select rewrite is more readable, so it is unreasonable to expect an OCL author to write in this tooling-friendly rather than reader-friendly style. An efficient OCL tool should perform the rewrite automatically.

Recognition of `select` followed by `reject` could be characterized as a high level form of Peephole Optimization. With 10 distinct iterations and four distinct collection kinds we could require (4*10)*(4*10)=1600 candidate peephole optimizations to be assessed and coded for all possible 2-term permutations; yet more for 3-term.

In the part 2 sister paper, we re-characterized collection operations and iterations using the new `gather` and `search` iterations. We will now use these to derive the intuitively obvious approach in a way that can be automated with no peephole optimizations for `gather`.

2.3. gather-gather optimization

The Sequence overload of gather is modeled using the Eclipse OCL standard library as:

```
type Sequence(T) : SequenceType conformsTo OrderedCollection(T) {
{
  iteration gather(V)(i : T[?] | lambda : Lambda T() : V[?]) : Sequence(V) {
    post: self->forAll(i with x | result->at(x) = i.lambda());
  }
  iteration select(i : T[?] | lambda : Lambda T() : Boolean[1]) : Sequence(T) {
    body: self->gather(i | if i.lambda() then i else ω endif)->compact();
  }
}
```

The collection class template parameter T and operation template parameter V define the type signature. gather has a single iterator i whose type corresponds to the source collection element type; it may be null. The iteration body is modeled as a lambda expression named lambda and typed as a Lambda from a T source without parameters to a V return that may be null. The iteration result is a Sequence(V).

The postcondition specifies that the result is element-wise the result of computing the lambda expression for the input element.

The specification of select once again has a source-typed iterator i, and a lambda-expression, that is now required to have a Boolean result. The body defines an implementation of the iteration that rewrites to the gather iteration with a nested lambda-expression that wraps the selection. The OCL specification suggests that select and other iterations can be realized by iterate. This is inefficient since iterate's accumulator is procedural; the accumulator successively accumulates each result element, creating a new result collection at each step. In contrast, the rewrite to use gather is declarative with the special value ω used as a placeholder for the not-available value on the dead computation path. The ω terms are squeezed out by the compact operation.

This recharacterization supports the rewrite of our example (with explicit iterators):

```
source->gather(i | if (i.name <> null) then i else ω endif)
       ->compact()
       ->gather(j | if not j.name = '' then j else ω endif)
       ->compact()
```

We may simplify the if and eliminate the first compact by handling the ω downstream.

```
source->gather(i | if i.name <> null then i else ω endif)
      ->gather(j | if (j <> ω) and (not j.name = '') then j else ω endif)
      ->compact()
```

Both gathers operate in lock-step element-wise with no dependency on sibling iterations, so we may merge the second into the first using the result of the first as the iterator of the second.

```
source->gather(i |
           let j = if i.name <> null then i else ω endif in
           if (j <> ω) and (not j.name = '') then j else ω endif)
      ->compact()
```

Substituting the let-variable for j and simplifying gives:

```
source->gather(i | if (i.name <> null) and (not i.name = '')
               then i else ω endif)
      ->compact()
```

This provides a rational automate-able derivation of the intuitively obvious.

The corresponding processing and control flow is shown in Fig 3. The select and reject are aggregated by the gather. The compound processing for name <> null and not (name = '')) computations sequences 'horizontally' rather than 'vertically'. As well as the changed order of evaluation, Fig 3 shows a hatched background with dead/not-available values represented as ω.

There is now no unnecessary intermediate result and no need for two collection traversals. The comparative performance of alternate implementations of this example are presented in Section 5.

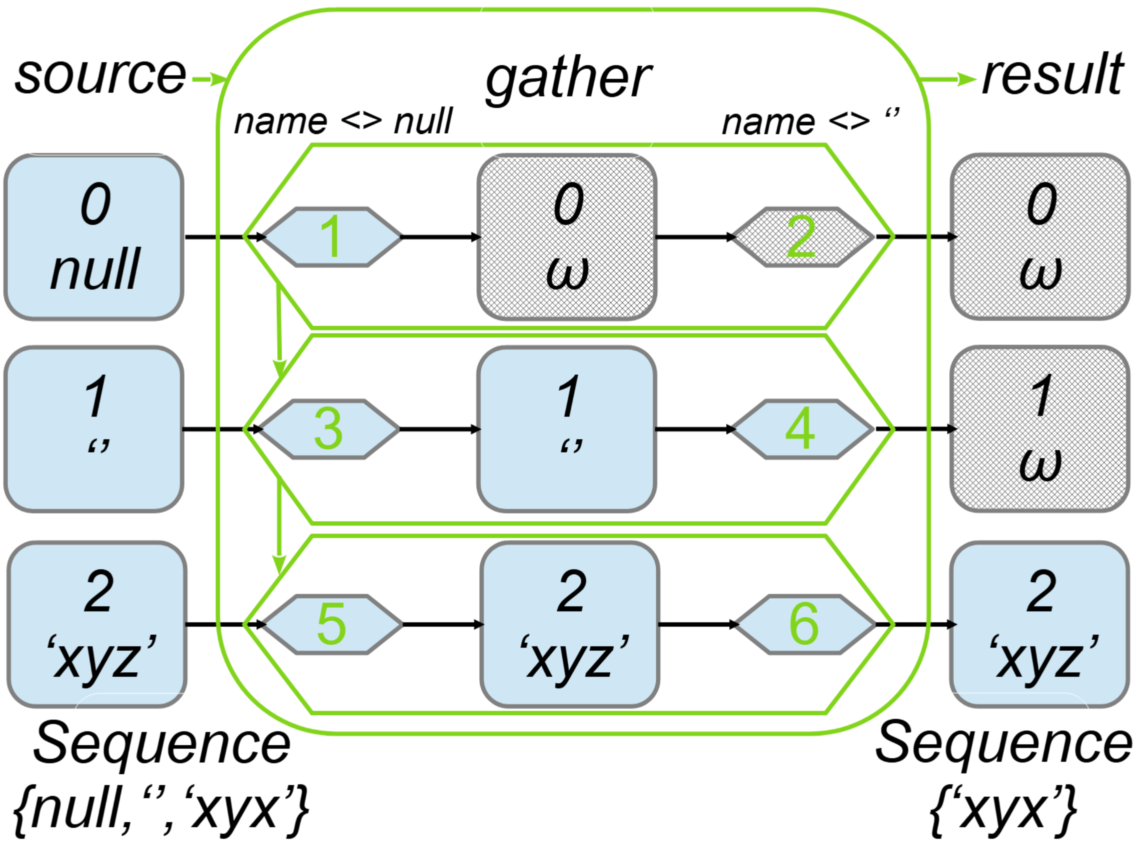

Figure 3: Simple Example (improved approach)

2.4. gather-gather optimization for sets

With many, perhaps all, declarative iterations and operations amenable to rewriting to use `gather`, there will be many opportunities for `gather-gather` cascades to be identified and optimized.

However there is one major impediment to this optimization; the execution of each per-element lambda-expression must be mutually independent. This condition is not satisfied by set iterations over a non-set since each element execution must check that it is not duplicating an earlier execution. The simple solution is to reify the set between the `gathers`, but that incurs the cost of creating the set.

The `CoCollection` provides an alternative solution, which may often be faster and certainly smaller. The ‘uniqueness’ is realized as a potentially one-bit per-element Boolean flag that indicates whether the element is the first element with the associated element value. This contrasts with at least 128 bits per element for a Java HashSet node. Use of a `CoCollection` supports the rewrite of set creation and then some collection operation as in `x->asSet()->collectNested(...)` to use `gather` as:

```
x->asSet()->gather(...)->compact()
```

The `asSet` can be replaced by a `coCollection.isFirst` guard.

```
let coCollection = x->coCollection() in
    x->gather(i | if coCollection.isFirst(i) then ... else ω endif)
     ->compact()
```

The element interdependence is now localized in the `coCollection` for which the first invocation of `isFirst` will trigger the lazy evaluation. The bodies of multiple `gathers` involving sets can now be merged. However in practice data dependencies on the `coCollection` may inhibit most merges. If one set is computed from another set, then obviously the first set is needed to avoid duplication.

There may however be a trade-off when the cost of duplication is low. Rather than de-duplicating immediately, it may be more efficient to perform duplicate calculations and de-duplicate later. This may well be worthwhile if the duplicated lambda-expressions are simple. It may also be worthwhile if the overall calculation may terminate early. It is then desirable to favor maximizing horizontal progress to discover an early termination over eager vertical determination of uniqueness. A compiler can estimate the cost of the two approaches and generate code accordingly.

3. Idioms

Analysis of the UML 2.5 OCL constraints reveals a few idioms that are easily accommodated.

3.1. selectByKind

OCL 2.4 introduced the `x->selectByKind(T)` operation to replace the clumsy `x->select(oclIsKindOf(T))->collect(oclAsType(T))`. However at least 10 usages for UML 2.5 remain. These are automatically resolved by successive rewrites to

```
x->gather(p | if p.oclIsKindOf(T) then p else ω endif)
  ->compact()
  ->gather(oclAsType(T))
  ->compact()

x->gather(p | if p.oclIsKindOf(T) then p else ω endif.oclAsType(T))
  ->compact()

x->gather(p | if p.oclIsKindOf(T) then p.oclAsType(T) else ω endif)
  ->compact()
```

The final `p.oclAsType(T)` is necessary to ensure that `T` is seen by downstream expressions. It is not necessary when executing, since the preceding `p.oclIsKindOf(T)` guarantees that it is a no-op. The rewrite to `x->gather(...)->compact()` gives the same result as `selectByKind(...)`. Execution of the `gather(...)` can of course compact as it goes rather than produce a bloated intermediate that is then compacted.

Normalization to use `gather` renders `selectByKind` redundant, although the user may nonetheless choose to use `selectByKind` for readability.

3.2. select-then-count

Another common idom in the UML 2.5 OCL constraints is epitomized by

```
behavior.ownedParameter->select((direction =
  UML::Classification::ParameterDirectionKind::return))->size() = 1
```

This could have used the `one()` iteration. Similarly

```
allNamespaces()->select(ns | (ns.name = null))->notEmpty()
```

could have used the `exists()` iteration. However

```
parameterSubstitution->select(b | (b.formal = p))->size() <= 1
```

has no shorter form.

These examples all demonstrate a common theme; a computation to count rather than use the results.

We can rewrite `size` using a `search` for use when following a `gather` so let us investigate the `gather-search` optimization.

4. gather-search optimization

We have described the `compact` operation, but not defined it. With the aid of the new `search` iteration we can define it together with helpful variants. The definition of these is modeled as follows.

```
type Sequence(T) : SequenceType conformsTo OrderedCollection(T)
{
  iteration search(V,W)(i : T[?];
                        acc : W |
                        next : Lambda T() : W,
                        break : Lambda T() : Boolean,
                        return : Lambda T() : V[?]) : V[?];
                       ) : V[?];

  operation compact() : Sequence(T) {
    body: self->search(i; acc : Sequence(T) = Sequence{} |
                 if i =ω then acc else acc->append(i) endif,
                 false, acc);
  }

  operation compactingSize() : Integer {
    body: self->search(i; size : Integer = 0 |
                 if i =ω then size else size+1 endif,
                 false, size);
  }

  operation compactingSizeLessThan(sizeThreshold : Integer) : Boolean {
    body: self->search(i; size : Integer = 0 |
                 if i =ω then size else size+1 endif,
                 size >= sizeThreshold, size < sizeThreshold);
  }
}
```

The `search` iteration like `iterate` processes each input element sequentially with an iterator, accumulator and lambda-expression to determine the `next` accumulator value. Syntactically they are separated by `;` and then `|`. `iterate`'s limitations on early exit, and the requirement for the accumulator to be the result are removed by defining two further lambda-expression arguments. `break` is true for the search to terminate early. `return` is the value to be returned. To accommodate this flexibility, the outer class template parameter `T` for the collection element type is augmented by the iteration template parameters `V` and `W` for result and accumulator types respectively.

The basic `compact` operation appends non-ω inputs to create the output. It does not stop early.

Similarly the `compactingSize` operation does not stop early. It counts rather than appends non-ω inputs to create the output. This avoids the need to create and repeatedly update a collection to accumulate unused results.

The more powerful `compactingSizeLessThan` also counts non-ω inputs, but breaks when `size >= sizeThreshold` and returns `size < sizeThreshold`. This not only avoids the unnecessary collection updates but also avoids unnecessary upstream computations.

A compiler optimizing

```
x->select(...)->reject(...)->notEmpty()
```

may initially rewrite as:

```
x->gather(... and not ...)->compact()->notEmpty()
```

then recognising the `compact()->notEmpty()` idiom rewrite as

```
x->gather(... and not ...)->compactingSizeNotEmpty()
```

Finally rewriting the `compactingSizeNotEmpty()` and merging the per-element bodies gives:

```
x->search(i;  hasElement : Boolean = false |
    if  ... and not ... then true else hasElement endif,
    hasElement, hasElement)
```

A cascade of declarative gathers can therefore be merged with a procedural search. Overall a single iteration is performed under control of the search.

In addition to the search iterations:

`any`, `exists`, `forAll`, `isUnique`, `one`

the following OCL collection operations project a simple metric from a collection making reification of the collection unnecessary:

`count`, `excludes`, `excludesAll`, `includes`, `includesAll`, `isEmpty`, `max`, `min`, `notEmpty`, `size`, `sum`

4.1. search-search and search-gather optimization

The `search` iteration is procedural and often produces a single control value. There are therefore very limited prospects for collection optimizations following a `search`.

The compacting phase of `gather` can usefully be configured to count result terms rather than populate a redundant compacted result. And, when the absence of `invalid` and `null` is provable, the counting can terminate once a threshold has been passed.

5. Results

We can demonstrate the effect of good and bad implementation options with the rather extreme example of a `HasValidlyNamedElement` constraint:

```
x->select(name <> null)->reject(name = '')->notEmpty()
```

Table 1 shows the execution time for four different implementation options with model sizes varying from 10 to 1,000,000 model elements for `x`. Each implementation is hand-coded in Java to demonstrate what an OCL to Java code generator might aspire to. The following pseudo-OCL defines each implementation using `let...in...` to emphasize the intermediate variables of each implementation.

```
Set,Set   let s1 = x->select(name <> null) in
          let s2 = s1->reject(name = '') in
          s2->size() > 0
Set       let s = x->select((name <> null) and not (name = '')) in
          s->size() > 0
Count     x->iterate(e; acc : Integer = 0 |
            if (name <> null) and not (name = '') then acc+1 else acc endif
          ) > 0
Exists    x->exists((name <> null) and not (name = ''))
```

The Set,Set implementation performs a `select` followed by `reject` followed by `size` exactly as specified by the problem. A distinct set is created after each iteration.

The Set implementation merges the `select` and `reject` followed by `size`. Only one set is created.

The Count implementation merges the `select` and `reject` and counts the results. No set is created.

The Exists implementation also merges the `select` and `reject` but breaks on the first valid name.

The results in Table 1 show expected O(NlogN) trends where a set must be created and O(N) for the linear count. The Set result demonstrates the benefit of merging to perform a single `select`. The Count result demonstrates the further benefit from just counting the per-element results. Finally the Exists result exploits a short circuit to terminate computation as soon as a definitive result is found.

Components	10	100	1000	10000	100000	1000000
Set,Set	0.000001	0.000006	0.000055	0.000565	0.014466	0.209042
Set	0.000000	0.000003	0.000023	0.000244	0.005122	0.067205
Count	0.000000	0.000000	0.000000	0.000002	0.000015	0.000188
Exists	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

Table 1
Select-Reject Example Performance in seconds

The results are averaged over 100 runs to diminish the unpredictability caused by cache alignment fortuity. But perhaps 5% randomness remains.

The important conclusions are as expected. Set computation is expensive[1]; two set operations are slower than one, one is slower than none. Avoiding unnecessary work by omitting unused results can be very beneficial.

6. Further Work

The optimizations in this paper are very much a work in progress. Results demonstrating the promise of the approach are based on hand-coded Java; at the extremes the results demonstrate nearly 100% and nearly 0% optimization.. Considerable work is needed to integrate into Eclipse OCL. Only then can we discover the effectiveness of the optimizations on a large corpus such as the UML OCL constraints applied to a non-trivial model such as the UML metamodel.

7. Related Work

Cuadrado [5] provides a catalog of OCL optimizations with the intent of tidying up the rather crazy OCL resulting from template-driven automated OCL synthesis. It is not expected that hand-written OCL would be improved. Most of the optimizations are subsumed by the symbolic Constant Folding and Common Subexpression Elimination of Eclipse OCL's Java Code Generator. Whereas Cuadrado attempts to inline let-expressions in the interest of 'readability', Eclipse OCL's CSE maximizes let-variables so that partial results are amenable to Loop Hoisting. Many let-variables makes for a much simpler Java Code Generator avoiding complexities when an `invalid` is thrown or caught within a complex expression. can be hoisted. Cuadrado's 'iterators' optimizations are a form of peephole optimization subsumed by the more general `gather-gather` optimization presented in Section 2.3.

Cuadrado provides a comprehensive review of papers optimizing generation or rewriting of OCL. This paper seems to be unique in tunneling below the OCL specification to identify lower level primitives that improve performance.

8. Summary

We have exploited the unification of the four collection kinds as a kind-specific views of a `CoCollection` to reduce costs and share memory for diverse representations.

We have exploited the new declarative `gather` and procedural `search` iterations to normalize many collection operations and iterations by rewriting.

We have recognized that many results are just counted rather than used and so we define compacting variants to avoid creating the unused results.

With just one collection representation and two iteration representations, we are able to identify and optimize cascades of collection operations.

[1]The Java HashSet is pretty good. Only slightly better performance is possible for smaller sets.

The effectiveness of the optimizations can vary from 0% to 100%. Future work will determine whether the optimizations are sufficient to make use of auto-generated code from OCL feasible for UML validation.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] Object Management Group, Object Constraint Language, Version 2.4, 2014. URL: https://www.omg.org/spec/OCL/2.4/PDF, formal/2014-02-03.

[2] Unified Modeling Language, Infrastructure, version 2.5, OMG Document Number: formal/2015-03-01 ed., Object Management Group, 2015. URL: http://www.omg.org/spec/UML/2.5/.

[3] Eclipse OCL Project, 2025. URL: https://projects.eclipse.org/projects/\protect\penalty\z@modeling.mdt.ocl.

[4] E.D.Willink, OCL Collection Optimization - Part 2 - Components, in: submitted to OCL 2025 @ STAF 2025, Koblenz, 2025.

[5] J. S. Cuadrado, A verified catalogue of OCL optimisations, Software and Systems Modeling 19 (2020) 1139–1161.