OCL Collection Optimization - Part 2 - Components

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 apparent similarity of OCL's Set and Sequence to Java's Set and List encourages an OCL implementer to re-use the Java functionality. However OCL and Java are not the same. The immutability of OCL Collections can result in inefficient churning of intermediate results. We revisit the implementation of collections to provide determinism at no-cost, and identify a new characterization of iterations that supports compile-time optimizations.

Keywords

OCL, Collection optimization, Iteration optimization

1. Introduction

OCL [1] evolved to satisfy UML's [2] need for textual model constraints where graphics was not appropriate. When Eclipse OCL was used to rectify the very many problems [3] in the pre-UML-2.5 OCL, UML's OCL was executed for the first time. Previous execution by commercial UML tools had been based on a manual transliteration with appropriate caches and fixes . The results for direct automated execution were very very disappointing. After some crazy speed problems were fixed, one significant remaining concern seemed to be the cost of cascades of collection operations.

"Deterministic Lazy Mutable OCL Collections" [4] reported on some attempts to improve collection performance. The mutable collections improved `iterate` accumulation from O(N*N) to O(N). A unified deterministic collection showed promise but failed because the unified collection had no collection kind and so could use the wrong semantics for e.g. = (equals). Lazy execution was difficult to achieve without unwanted incompatibilities.

In this paper we apply an extra level of indirection [5] to make a unified deterministic collection feasible. We also re-specify iterations without using iterate. This paves the way for part 3 of this paper to exploit laziness.

In this part we first review typical collection implementations in Section 2. Then in Section 3 we present a revised approach that offers better prospects for optimization. In Section 4, we re-specify iterations in a manner that suits implementation. The comparative performance of our new approach is presented in Section 5. In Section 6 we report on related work and in Section 7 we summarize.

2. Collections

The current OCL specification defines four concrete implementations of the abstract `Collection` class. These support the four permutations of unique/non-unique, ordered/not-ordered content[1].

OCL tools implemented using Java of course seek to re-use Java types, so a typical implementation of the four specified kinds is summarized in Table 1.

The `Sequence` and `OrderedSet` forms provide for the obvious cases of creation/load-ordered content that may or may not be unique[2].

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)

[1] Discussion following Gogolla [6] agreed on count-aware/count-blind and order-aware/order-blind as clearer terminology.
[2] Gogolla [7] has suggested that consistent 3-letter names such as Seq and Ord would be more readable.

Specified Kind	Implementation Class	Contents
Sequence	enhanced Java List	Array of elements
Set	enhanced Java Set	Hashtable of elements
OrderedSet	adjusted Java LinkedHashSet	Hashtable of elements with linked list
Bag	custom Bag using Java Map	Map of element value to occurrence count

Table 1
Typical OCL Collection implementations

The additional `Bag` and `Set` forms allow the metamodeler to specify that order is not significant.

Unfortunately, OCL 1 did not provide the `OrderedSet` type and so for many years `Set` was used instead. This imposed a loss of ordering and introduced non-determinacy, which is a very strange attribute of an otherwise strong language. OCL 2 provided `OrderedSet` without tackling the indeterminacy is OK culture.

From an implementation perspective, re-using Java's non-determinate HashSet is convenient and trades the loss of 'irrelevant' order for a much faster implementation of includes/contains content testing. This is a bad trade-off since, at best, the indeterminacy can make debugging difficult. At worst, an unjustified confidence that ordering is not significant may lead to unpredictable failures. A different implementation of `Set` could usefully preserve the creation / load ordering and, as we shall see, may actually be more efficient.

`Bag` is perhaps the least useful of the collections. It is the inevitable result of a collection of not necessarily unique results derived from a set of elements. Traditional Java provides no counterpart for OCL's bag, so a bespoke implementation is necessary. This will probably provide a fast `Bag.count()`. However, this operation is often not used, so the effort to provide eager support is dubious.

2.1. Uniqueness / Count-awareness

The uniqueness of elements provided by an `OrderedSet` or `Set` is of course important to avoid duplicate results. However, uniqueness is potentially expensive since creation of a set incurs an at least O(NlogN) cost rather than O(N) for a `Sequence`. However, once this creation cost has been incurred, operations such as `includes` compute in O(1) rather than O(N) time. A Java Set has no knowledge of how it will be used, so it makes the sensible trade-off to bound creation cost at O(NlogN) and testing at O(1) using a sophisticated memory expensive approach. As we shall see in Section 5, for at least small collections, a low-overhead representation may outweigh the correspondingly slower O(N*N) and O(logN) performance.

2.2. Order-awareness

The ordering of elements in a collection may also be important and fundamental. For instance, a person's first, second, third, ... names are clearly ordered. So we use `OrderedSet` or `Sequence` kinds.

Sometimes an order is useful but derived. For instance, houses may have one positional order for a postman's round or a second alphabetical order within a find-address-popup-menu. These two derived orders may differ from a third more fundamental order associated with internal database entries. The internal order should not be exposed to the user, so specification of a `Set` of houses is appropriate even though the underlying implementation has an order. In practice, all collections have an order that is defined by the order of creation or loading of model elements. Behind the scenes all collections can be ordered. `Set` can be used to prevent OCL expressions exploiting the private implementation order, rather than being an excuse for indeterminacy. The private order is of course exposed by iterations.

3. CoCollections

Java collections are designed to support frequent mutation by operations such as `Collection.add()`. The consumer of a Java Set is a Java program whose hard-to-analyze side-effects make it very difficult to tailor performance to the usage. Java therefore provides a good general purpose implementation that avoids over-complex derived state. (Derived state could accelerate rare operations but penalize common operations with incremental update costs.)

In contrast, OCL collections are immutable; each operation such as `Collection.including()` results in a new collection. The consumer of an OCL collection is a side-effect-free OCL expression. The usage context can be exploited and since the final state of the collection is known, derived state can be eagerly or lazily computed and cached once; no incremental update will be required. Operations such as `count` or `includes` are applicable to all collection kinds and can benefit from a fast lookup.

Specified Kind	Implementation Classes
Sequence	Sequence view, CoCollection
Set	Set view, CoCollection
OrderedSet	OrderedSet view, CoCollection
Bag	Bag view, CoCollection

Table 2
Revised OCL Collection implementations with CoCollections

We therefore propose to replace the four heavyweight classes for each collection kind, by four lightweight view classes that delegate to a unified `CoCollection` as shown in Table 2. As indicated in Figure 1, the `CoCollection` holds the `elements` of a collection in a fixed-sized ordered array of not-necessarily unique elements, thereby directly supporting a `Sequence` and the `->asSequence()` ‘conversion’. Other kinds and ‘conversion’s are supported by additional lazily computed contents.

When unique behavior is required, the optional `isFirst` is computed so that a unique iteration of a `CoCollection` skips the non-`isFirst` elements that duplicate earlier indexes. We may therefore rewrite `aSequence->asSet()->forAll(p | ...)` as:

```
let coCollection = aSequence->coCollection() in
  coCollection.elements->forAll(p | coCollection.isFirst(p) implies ...)
```

The `isFirst` bit array can be efficiently computed from the `hash2index` ‘map’. The ‘map’ is realized as an array of entries from each element’s hashcode to its index within the `elements` array. The entries are sorted in hashcode order. A binary search can therefore locate the index of an element via its hashcode for an `includes` operation in O(logN) time. Duplicate entries have duplicate hashcodes in adjacent entries, so the `count` operation counts all adjacent entries whose index identifies the element of interest. Significant speed-up of `intersection` is possible as we shall see in Section 5, where the performance of variants of the `CoCollection` approach optimized for diverse collection sizes are contrasted with a Java HashSet approach.

On a modern 64-bit computer, the size of a sequence-only `CoCollection` is 8 bytes per element; just the same as a Java ArrayList. Once the lazy set-functionality is activated, the size grows to 17 bytes per element, significantly less than about 36 bytes per element for Java HashSet or 44 bytes for a deterministic Java variant of HashSet. A further lazy growth can support sharing of partial `closure` results for the common use case where overlapping closures are computed for many elements of a collection.

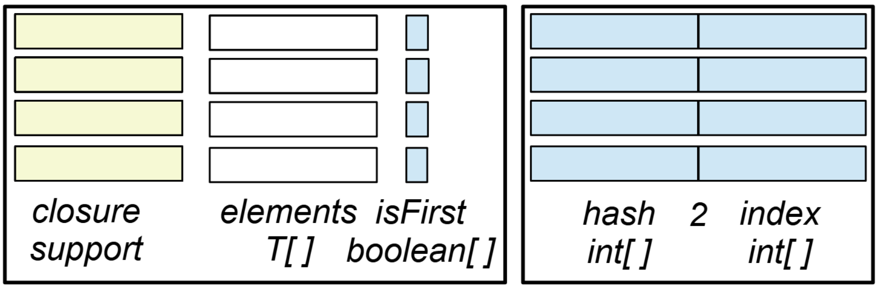

Figure 1: CoCollection arrays

Whether the lazy growth is worthwhile can be subject to heuristics observing whether usage of count within an iteration loop is likely to benefit from re-use whereas an isolated use of count may be better realized as a naive linear traversal.

3.1. Redundant Uniqueness

A further speed trade-off is available for the aSequence->asSet()->forAll(p | ...) rewrite when the cost of the loop body ... is smaller than the coCollection.isFirst(p) repetition guard. The guard can be omitted since neglecting to prune duplicate forAll terms does not change the result. We can rewrite as aSequence->forAll(p | ...) omitting the ->asSet(). The small cost of coCollection.isFirst(p) is known. If we can confidently estimate the cost of ..., the appropriate trade-off is possible.

Since OCL is side effect free, we can commute uniqueness enforcement with other computations so that the following are all equivalent:

```
aCollection->asSet()->select(...)->asSet()
aCollection->select(...)->asSet()->asSet()
aCollection->select(...)->asSet()
```

The latter saves the cost of a set enforcement but again incurs the additional cost of a redundant select(...) for each repeated element. Whether this is a good trade off cannot be accurately determined at compile time without profiling information. However, the cost of select(...) can be estimated heuristically at compile time. Combined with knowledge of the collection size at run-time, a plausible eager/lazy trade-off can be achieved.

3.2. Explicit indexOf

The simple implementation of indexOf for an OrderedSet or Sequence involves a linear search of typically half the content to locate the matching value; O(N).

This can be accelerated dramatically using the CoCollection's sorted array of hash2index entries. The O(N) linear-search can therefore be replaced by the much faster binary search; O(logN). The size of the entry array is also significantly smaller than a full map.

3.3. Implicit indexOf

A common use of indexOf occurs when correlating two lists such as the parameter and argument lists of an operation call. The iteration over one list provides one element (parameter) whose index is used to locate the corresponding second element (argument). In the following f(par, arg) performs the correlation check.

```
    pars->forAll(par | let arg = args->at(pars->indexOf(par)) in f(par, arg))
```

The inefficiency of this is offensive; why is a content search necessary to discover the index that the iteration already knows? Eclipse OCL supports a coIterator [8] that provides the index of a list iteration (or the value of a map iteration).

```
    pars->forAll(par with index | let arg = args->at(index) in f(par, arg))
```

(The with keyword separates the regular iterator par from its coIterator index.)

Avoiding the unnecessary indexOf call may avoid the need to populate the CoCollection hash2index cache that provides the indexOf support.

4. Iterations

The generic iterate iteration is specified in OCL 2.4 §11.8 as the basis for all iterations.

> *The semantics of each iterator expression is defined through a mapping from the iterator to the 'iterate' construct.*

Emulating the definition-rewrite as an implementation-rewrite to use `iterate` can be a poor basis for tooling. Consider `aCollection->collect(p | ...)` where `aCollection` is a collection of `T`-typed elements and `...` is the lambda-expression for the loop body . It may be rewritten as:

```
    aCollection->iterate(p; acc = Bag(T){} | acc->includes(...))
```

This is very inefficient since each sequential per-element update of the accumulator creates a new collection. The overall performance will be O(N*N) for count-blind and O(N*N*logN) for count-aware collections. A more efficient direct implementation can exploit the invisibility of `acc` outside of the iteration and so use a hidden mutable Java collection [4]. This can improve performance to O(N) for count-blind and O(N*logN) for count-aware collections.

`iterate` semantics are too strong for the procedural `any`, `exists`, `forAll`, `isUnique` and `one` iterations that must proceed sequentially, but which may terminate early; 'iterate' does not allow for early termination.

`iterate` semantics are too strong for the declarative `closure`, `collect`, `collectNested`, `reject`, `select` and `sortedBy` iterations that may execute concurrently; 'iterate' does not allow for concurrency.

We therefore introduce a new characterization of OCL iterations by rewriting to one of two new underlying iterations, a helper operation and a special place-holdingvalue.

4.1. Declarative gathers

The new `gather` iteration gathers the results of a per-element lambda-expression to form a new collection. It is identical to `collectNested` but fundamental, rather than a remedy for the pragmatic `collect`. The `Sequence` overload of `gather` is modelled using the Eclipse OCL standard library [9] 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 collectNested(V)(i : T[?] | lambda : Lambda T() : V[?]) : Sequence(V) {
    body: self->gather(i | 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();
  }
  operation compact() : Sequence(T) {
    post: result->excludes(ω);
  }
}
```

The collection class template parameter `T` and iteration 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 `collectNested` rewrite is trivial; functionality is delegated directly to `gather`.

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 in an `if`. The `if` variously selects the wanted `i` or the unwanted ω.

The ω special value is a place-holding collection element for a collection element that is not-available. It is intentionally missing; there is nothing `invalid` or `null` about ω. Any computation using ω is not-available, since ω should have been optimized away. ω is a compile-time artifice to ensure that the `gather` result is the same-size as its source. In part 3 [10] this facilitates `gather-gather` optimizations.

The compact helper operation converts a collection that may involve not-available ω values to a regular collection without these values. Higher index elements are moved down to eliminate the gaps.

The collect iteration, not shown, appends a flatten call to the collectNested rewrite. Most usages of collect do not use nested collections and so the redundant flatten can be omitted.

The reject rewrite, not shown, is similar to select. It just swaps then and else terms.

4.2. Procedural searches

The new search iteration supports the any, exists, forAll, isUnique and one iterations. The traditional iterate supports one iterator, one intermediate result accumulator and one lambda-expression for the next-accumulator-value update. The semantics are informal. The new search supports multiple iterators with two additional lambda-expressions. The semantics are defined using a postcondition. The additional break lambda-expression determines whether the iteration can terminate early. The additional return lambda-expression determines the return value. The result type may therefore differ from the accumulator type as required by the one iteration.

The one-iterator variants of search, and the rewrite from exists, are modeled using the Eclipse OCL standard library language as:

```
 type Sequence(T) : SequenceType conformsTo OrderedCollection(T)
 {
   iteration search(V,W)(i : T[?]; acc : W | next : Lambda W(T) : W,
        break : Lambda W(T) : Boolean, return : Lambda W(T) : V[?]) : V[?] {
     post: let accs:Sequence(W), breaks:Sequence(Boolean), returns:Sequence(V) in
        self->forAll(i with x |
            let nextAcc = if x = 1 then acc else accs->at(x-1) endif.next(i) in
            let nextBreak = nextAcc.break(i) in
            accs->at(x) = if nextBreak then ω else nextAcc endif
            and breaks->at(x) = nextBreak
            and returns->at(x) = nextAcc.return(i)
        ) and result = if notEmpty() then returns->at(self->size())
                        else acc.return(null) endif;
   }
   iteration exists(i : T[?] | lambda : Lambda T() : Boolean[?]) : Boolean[?] {
     body: self->search(i; acc : Boolean[?] = false |
              let e = i.lambda() in
              if e = true then true
              elseif acc.oclIsInvalid() then acc
              elseif e.oclIsInvalid() then e
              elseif acc = null then null
              elseif e = null then null
              else false endif,
              acc = true,
              acc);
   }
   iteration one(i : T[?] | lambda : Lambda T() : Boolean[?]) : Boolean[?] {
     body: self->search(i; acc : Integer = 0 |
              if i.lambda() then acc + 1 else acc endif,
              acc > 1,
              acc = 1);
   }
   operation compact() : Sequence(T) {
     body: self->search(i; acc = Sequence{} |
                   if i = ω  then acc else acc->append(i) endif, false, acc);
   }
 }
```

The search postcondition uses the artifice of uninitialized Sequences that serialize each of the intermediate results. Each uninitialized Sequence element is ‘initialized’ exactly once by constraining its value before it is used in a later iteration.

The exists rewrite has a complicated lambda-expression to support 4-valued Booleans: any true returns true, else any invalid returns invalid, else any null returns null, else false. The second lambda-expression breaks when the accumulator is true. The third lambda-expression returns true, invalid, null or false.

The rewrite is much simpler when symbolic analysis is able to prove that i.lambda() is 2-valued.

```
    x->search(i; acc : Boolean[1] = false | i.lambda(), acc, acc)
```

4.3. closure

The `closure` iteration does not fit the above characterization since an efficient implementation involves a hidden map [8]. With the aid of the hidden map and using `...` to represent the arbitrary lambda-expression, `aCollection->closure(...)` may be rewritten as

```
aCollection->gather(p | closureMap->at(p))->flatten()->asSet()
```

The hidden map contains each of the required closure partial results. The map contents satisfy the following recursive constraint.

```
closureMap->forAll(k with v |
        v = v->gather(...)->gather(q | closureMap->at(q))->flatten()->asSet()
    )
```

The value of each `k,v` map entry is the (transitive) aggregation of the map entries.

The hidden map(s) may be cached as part of the `CoCollection` to avoid re-computation when the `closure` is re-used with the same lambda-expression.

4.4. sortedBy

Evaluation of `aCollection->sortedBy(k | f(k))` similarly involves creation of a hidden map from sort key to original element.

```
let key2element = aCollection->gather(k | f(k) with k) in
map->keys()->sort()->gather(k | key2element->at(k))
```

The `key2element` map is populated with `f(k),k` by gathering the `f(k)` for each `aCollection` element `k` as a map from `f(k)` to `k`. The result is returned by sorting the keys and gathering the original `aCollection` elements in the order of the sorted keys. (`sort` should be an OCL library operation.)

The hidden map is unlikely to be re-usable, so there may be little benefit in caching it.

4.5. operations

Many operations such as `includes` are rewrite-able using `search`.

5. Results

The advantages (and disadvantages) of the new kind-specific view+CoCollection approach may be assessed by contrasting with a direct Java HashSet implementation for the following particularly ‘set’-intensive problem involving three ‘immutable’ collections.

> Given two array lists of N distinct random integer values, each with N/4 common values
> compute a Set from each array list
> compute the Set of the N/4 intersections

The execution times prove to be quite variable despite running the test code 2000 times to warm up the VM before starting measurements. The ‘problem’ is that we are dealing with a separate memory allocation for each model element. Consequently many memory accesses incur cache line collisions for a typical architecture that supports just two targets for each cache line. This ‘problem’ is worse for the HashSet that allocates 4 words for each hash node. In contrast, the new approach has just a couple of large arrays. Since the results are so variable, each result is reported as the minimum from 1000 runs; the minimum is the result that shows the least degradation due to cache collisions. The average is often twice the minimum with the maximum as much as eight times.

The results in Figure 2 provide performance results from 0 to 1,000,000 model elements on log-log scales. The results are normalized to ns/model element so a low-down horizontal line shows good O(N) performance. In practice many of the results show a gentle O(NlogN) rise. The O(1) overheads show as a downward slope where there are few elements to share the overhead. (The 0 model element point is plotted as 0.15 model element and normalized as 1 model element.)

The NonDeterminateJavaSet and NonDeterminateWellHashedJavaSet curves show the use of a Java HashSet without any attempt to maintain ordering; the performance is therefore not deterministic.

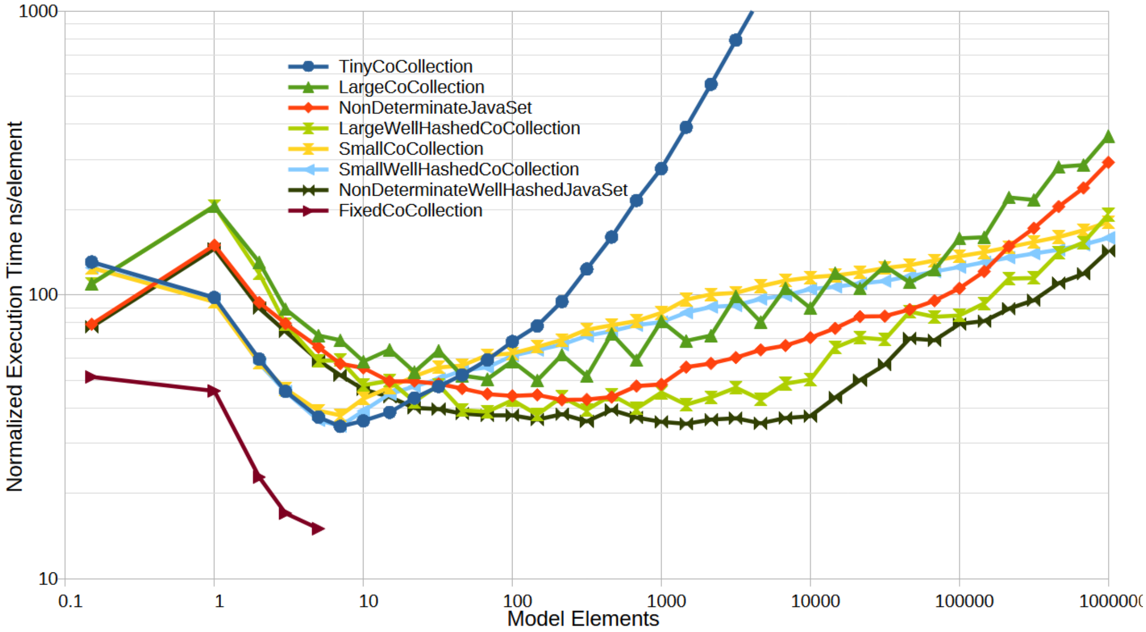

Figure 2: Performances constructing two sets and an intersection

For large collections the HashSet can be as much as four times faster than any of our `CoCollection` approaches.

Two curves are shown since a distinctly better performance is achieved when perfect hash codes guarantee uniqueness. For Class-type elements for which the object address could provide a unique value, it might be expected that Java's System.identifierHashCode would be a good hash; it isn't. For the 'well-hashed' curves, the hash is allocated from an incrementing count, as could be achieved by a model loader/manager. The NonDeterminateJavaSet curve demonstrates far more hash table collisions than the NonDeterminateWellHashedJavaSet.

The SmallCoCollection curves show a design expected to be suitable for small collections. It degrades steadily with size but is actually better than NonDeterminateJavaSet for huge models. It is much less sensitive to hashcode quality.

The TinyCoCollection curve uses an insertion rather than quick sort to prepare the `hash2index` entry array. As expected, this is O(N*N) for larger collections. It is better than the SmallCoCollection below 50 model elements.

The LargeCollection augments the SmallCoCollection with a HashSet to accelerate look-ups as suggested by [4]. This helps significantly when the hash codes are perfect.

Finally the FixedCoCollection 'curve' shows the performance of bespoke implementations for 0, 1, 2, 3 or 4 model elements. These are significantly faster through reduced overheads, but more significantly because there are no memory allocations for `elements` or `hash2index` arrays. The ' `elements`' memory is realized on the stack where it avoids cache conflicts and may be cached in registers. Use of the `hash2index` array is replaced by linear searches.

Small collections are very common, so clearly the FixedCoCollections can give determinism and a significant five-fold improvement over a Java HashSet. Up to 20 model elements can be faster and deterministic with a TinyCoCollection. For larger collections the cost of determinism may be two-fold. Note that since OCL collections are immutable, choosing the optimum collection approach can be done prior to construction.

The CoCollection approach provides more opportunities for sharing, so the two source sets may already be available. It is therefore interesting to look at just the create-intersection cost. The corresponding performances are shown in Figure 3. For really small collections, the FixedCoCollection gives determinism and a ten-fold improvement over a HashSet. For larger collections the Tiny or SmallCoCollection gives determinism and a two to three-fold improvement.

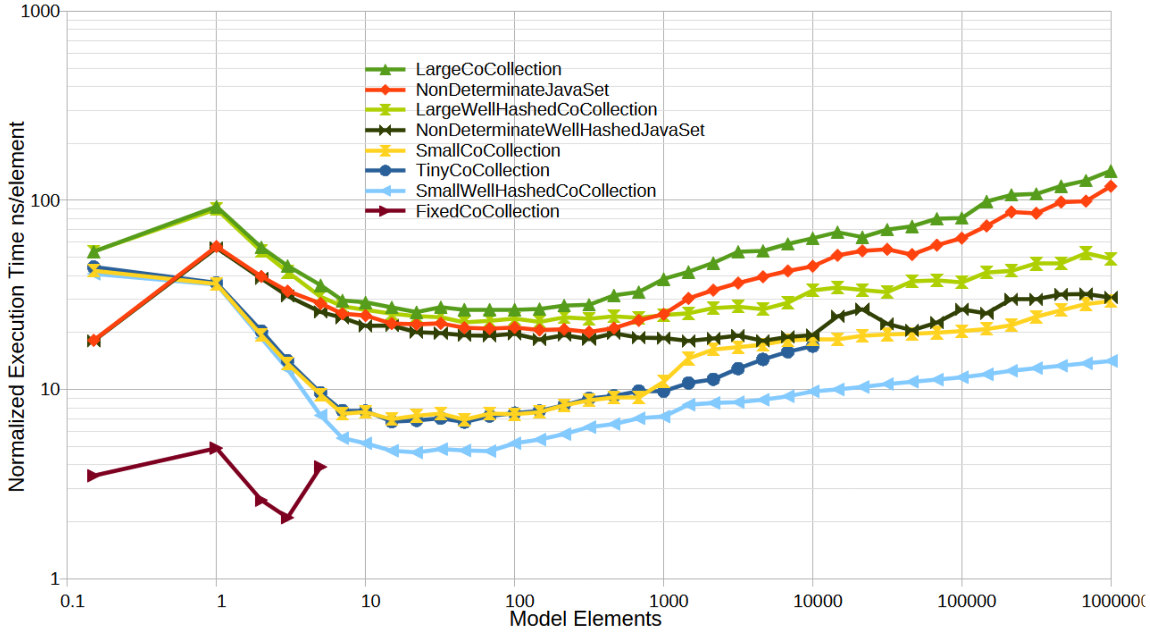

Figure 3: Performances constructing an intersection

The third paper [10] in this series exploits the `CoCollection` representation to share computational effort across cascades of collection operations.

6. Related Work

Each of Dresden OCL [11], Eclipse OCL [12] and USE [13] pursue the same Java implementation approach for OCL collections; a family of kind-specific classes provides OCL semantics for a hidden internal representation using distinct Java classes, some standard, some bespoke. There is no sharing of a unified internal representation as proposed for the `CoCollection`.

This work builds on the successes and remedies the failures of [4].

Many authors [3] have found aspects of the OCL specification worthy of criticism, but have held back, perhaps having too much respect for a specification that at its heart is obviously sensible.

This paper is perhaps unique in challenging the OCL 2.4 §11.8 statement delegating semantics to 'iterate'. Adherence to this statement imposes a procedural approach despite many operations such as `select` being amenable to a declarative approach.

OCL♮ [14] uses `iterate` as the sole mechanism for deconstruction.

7. Summary

We have reformulated the implementations of the four collection kinds using lightweight kind-specific views of a shared unified deterministic ordered `CoCollection`. Our results show that determinism can be provided with a threefold space saving, with a five-fold speed up for very small collections, and without speed penalty for collections of up to about 50 model elements and at moderate speed penalty for larger collections. Our results indicate a useful further speed-up when the sharing opportunities of `CoCollections` are exploited. Rather than a cost, there are functional benefits for making OCL deterministic.

We have re-specified the iteration operations replacing the inadequate `iterate` by fully-specified declarative `gather` and procedural `search`. The rewrites are suitable for implementation.

Normalization of collection expressions to one representation and two iterations paves the way for cascade operation in our third paper.

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] C. Wilke, B. Demuth, UML is still inconsistent! How to improve OCL Constraints in the UML 2.3 Superstructure, Electronic Communications of the EASST 44 (2011).

[4] E. D. Willink, Deterministic Lazy Mutable OCL Collections, in: Federation of International Conferences on Software Technologies: Applications and Foundations, Springer, 2017, pp. 340–355. doi:10.1007/978-3-319-74730-9_30.

[5] B. Lampson, Principles for computer system design, in: ACM Turing award lectures, 1993, p. 1992.

[6] M. Gogolla, L. Burgueño, A. Vallecillo, Refactoring Collections in OCL., in: STAF Workshops, 2021, pp. 142–148.

[7] M. Gogolla, What You Always Wanted to Know about OCL and Never Dared to Ask, in: OCL 2019 @ Models 2019, 2019.

[8] E. Willink, An OCL Map Type, in: OCL 2019 @ Models 2019, 2019. URL: http://www.eclipse.org/modeling/mdt/ocl/docs/publications/\protect\penalty\z@OCL2019MapType/OCLMapType.pdf.

[9] E. Willink, Modeling the OCL standard library, Electronic Communications of the EASST 44 (2011). doi:10.14279/tuj.eceasst.44.663.673.

[10] E.D.Willink, OCL Collection Optimization - Part 3 - Cascades, in: submitted to OCL 2025 @ STAF 2025, Koblenz, 2025.

[11] Dresden OCL Project, ???? Http://www.dresden-ocl.org/index.php/DresdenOCL.

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

[13] USE, The UML-based Specification Environment, ???? Http://useocl.sourceforge.net/w/index.php/Main_Page.

[14] F. Steimann, R. Clarisó, M. Gogolla, OCL Rebuilt, From the Ground Up, in: 2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, 2023, pp. 194–205.