OCL Collection Optimization - Part 1 - AllInstances

Edward D. Willink[1,2,*,†]

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

Abstract
The allInstances() operation is used by many authors to obtain a source domain for a grandiose invariant. However the use of allInstances() is considered poor practice by many authors. We examine the good and bad features of allInstances and identify caching mechanisms that can mitigate the bad features. Finally we identify compile-time rewrites that can make allInstances() obsolete at run-time but conclude that allInstances() really is a poor practice to be resolved by authors.

Keywords
OCL, Collection optimization, Caches

1. Introduction

OCL [1] evolved to satisfy UML's [2] need for textual model constraints where graphics was not appropriate. Authors more familiar with logic than the niceties of OCL find it convenient to express a grandiose constraint such as:

> *All People have a name.*

This could transliterate to an OCL invariant as

```
inv PersonHasName:
  Person.allInstances()->forAll(name <> null)
```

In this paper, in Section 2, we first review the problems associated with this transliteration. A simplistic recognition of these problems has caused a number of authors including myself [3] to strongly discourage use of `allInstances()`. As long ago as 2002, the Amsterdam Manifesto [4] identifies *"allInstances" considered dangerous*. Then in Section 3, we solve the problems within the scope of typical OCL tooling before identifying caching mechanisms to improve the tooling and rewrites that could make `allInstances()` obsolete. In Section 4, we make some observations about the usage and performance of the UML 2.5 OCL constraints. Finally we discuss related work in Section 5 and conclude in Section 6.

2. allInstances problems

Exploiting the example OCL snippet above as part of a Complete OCL document requires the invariant to have a package and class context. Perhaps:

```
package humans
context Person
  inv PersonHasName:
    Person.allInstances()->forAll(name <> null)
endpackage
```

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)

The seemingly redundant `context Person` is pretty ugly. Why can't packages have invariants?[1]

Typical OCL tools interpret the OCL quite literally and so when validating a model, every time an instance of `Person` is encountered, all invariants applicable to a `Person` are validated. With N `Person` instances, the invariant runs N times. But the invariant specifies a loop over all `Person` instances so the `name <> null` body is executed N*N times. This is very inefficient and potentially confusing as each violation is reported N times. Literal execution can be even worse. Each execution of `Person.allInstances()` searches the entire 'model' to locate the instances. A single total model search can be expensive. N total model searches is very expensive. The overall cost is worse than O(N*N*N) for what should be an O(N) problem.

Recognition of the multiple causes of poor tooling performance motivates the strong discouragement.

3. allInstances solutions

3.1. Package Constraint

The obvious answer to 'Why can't packages have invariants?' is to add support for package constraints/invariants. The UML Abstract Syntax, to which OCL is aligned, supports constraints on any `Namespace` so `Package` constraints are supported by the modelling. However UML constraints are an underspecified dustbin for which the semantics of a few usages are only partially specified. Behind the scenes, any tool vendor is free to recognise that packages have constraints with ill-defined semantics. A sensible implementation may inspire similar support by other tools. For UML, a package constraint can be drawn today. For OCL, a minor enhancement to the Complete OCL grammar is needed parse package invariants.

3.2. Practical alllinstances

The dreadful cubic performance of a naive implementation of `allInstances` is easily improved to quadratic by lazily caching the results from each distinct invocation.

An OCL-based application such as full model validation or model transformation comprises many OCL constraints that can be analyzed once before execution starts. This analysis can identify all the `allInstances` calls (and also all the implicit opposite navigations). These can all be eagerly cached in a single pass following or during model loading.

A further improvement is possible by skipping the eager caches dependent on elements that have no instances in the models.

The sizes of the caches can be reduced when multiple `allInstances` are used for distinct types with an inheritance relationship.

3.3. self-less allInstances

The grandiose example can be categorized as a self-less constraint since there is no use of the self-context in the contraint. It is easily rewritten to use the context class as `self`.

```
context Person
  inv HasName:
    self.name <> null
```

This avoids the need for a total model search and naturally validates the constraint exactly once for each Person instance. A performance speed-up from O(N*N*N) to O(N) should be observed.

The rewrite should be more readable since the `allInstances`, a `forAll`, and iterator can be removed. It is easier to understand a constraint on one example instance rather than all instances.

[1] There is an outstanding OMG OCL issue for package invariants.

3.4. self-ish allInstances

When `allInstances` occurs in an OCL expression that also uses the `self` context a rewrite is still possible.

3.4.1. gratuitous self-ish allInstances

The UML 2.5 OCL constraints have four redundant usages of `allInstances` such as.

```
context ConnectableElement::end() : Set(ConnectorEnd)
  body:
    ConnectorEnd.allInstances()->select(role = self)
```

It appears that the relevant UML author did not appreciate that all associations are navigable in all directions in OCL. Consequently the constraint just expresses the qualified navigation.

```
context ConnectorEnd
  body:
    self[ConnectorEnd::role].role->includes(self)
```

Since the opposite of role is explicitly defined, we can write

```
context ConnectorEnd
  body:
    self.end.role->includes(self)
```

This provides the redundant textual definition of the end property that is already defined 'graphically'. Gratuitous self-ish `allInstances` should be deleted since their sole utility is to confuse the reader.

3.4.2. other self-ish allInstances

Other examples of `allInstances` in conjunction with `self` can also be rewritten exploiting the ability to navigate in all directions. Logically since all associations can be navigated in either direction, it must be possible to reverse the navigation. A particular navigation to `self` from the `x` instance returned by an `T.allInstances()` can be reversed to navigate from `self` to `x.oclIsKindOf(T)`. However, it is unclear whether realistic complex navigation involving helper operations can be inverted without manual help.

The following example from UML 2.5 searches all `Association` instances for the few whose end types are a `UseCase self`

```
context uml::UseCases::UseCase
  inv binary_associations:
    Association.allInstances()
    ->forAll(a |
          a.memberEnd.type->includes(self) implies
          a.memberEnd->size() = 2)
```

We can start at a `UseCase self` and reverse navigate to the relevant `Association`.

```
context uml::UseCases::UseCase
  inv binary_associations:
    let a : Association = self.typedElement.association in
    a.memberEnd->size() = 2
```

The rewrite is clearer and faster. It avoids cubic execution costs. A further proportional saving arises from only evaluating the constraints for `UseCase` instances thereby avoiding redundant processing for the many (most) `Associations` that have nothing to do with `UseCases`.

3.5. what instances?

Avoiding the use of a total model search avoids another problem. How and where should the all-instances be located? Clearly it cannot be absolutely all instances since there may be instances behind some firewall somewhere. The search must be restricted to one or perhaps more relevant models in ways that the OCL specification does not define, but which might perhaps relate to a MOF Extent.

OCL is re-used by QVT [5] where there are distinct input and output domains. Which domains does an `allInstances` apply to? Does an execution of `allInstances` have to wait till all output instances are created?

These problems are avoided by a navigation from a distinct object.

A practical compromise is to search all models transitively referenced from all the input models. For EMF [6], that is all Resources in the ResourceSet, but does it also include metamodels and built-in Resources not in the ResourceSet? When evaluating `Class.allInstances()` for a UML model, are both user and UML classes returned?

3.6. Instance Cost

`allInstances()` may return a very large collection of instances. Is there memory to be saved?

A model is a graph of relationships (edges) between model elements (nodes) that are instances of the classes defined by the metamodel. The realization of each relationship involves at least one 'pointer' to the target node and often another 'pointer' to the source node of the relationship.

Many years ago , these pointers were realized using 16 bits, but that was inadequate for huge models. Paging approaches were needed.

The advent of 32 bit processors removed the problem but was a little wasteful for the many applications for which 65536 model elements is more than adequate. EMF reacted to this wastage by providing an ability to pack many Boolean attribute values into a single `eFlags` byte.

Now that 64 bit processors are almost universal, the use of 64 bit pointers is even more wasteful. Memories have got much larger so does it matter? At first sight no. But behind the scenes the performance of an application using thousands of model elements is determined by the performance of the memory system and its caches. These caches typically support many cache lines with at most two lines at addresses that share the same least significant bits. If three accesses contend for the same cache line, performance degrades.

The width of cache lines is also increasing, so access to a cache line may make 256 bits available from a single memory access. It is therefore beneficial to contrive to get multiple values from a single cache line. Unfortunately the size of a Java HashNode is too big and so two memory fetches may be required. Memory systems may antipate usage by assuming the next cache line will be needed; this helps the Java Hash Node.

Anyway using smaller model element pointers can be beneficial by improving cache hits.

It is tempting to use 16 bits for perhaps most model applications with a fallback to 24/32/64 bits for large/huge/pathological model sizes. This allows smaller references in models that can be expanded to actual addresses by a simple model-element-index to address lookup. Unfortunately the inverse search to encode an address as a model element index is liable to need a hashtable. The potential memory savings and speedups are very likely to be outweighed by additional calculations and tables.

4. Observations

The many syntactical and semantic errors [7] in the pre UML 2.5 OCL constraints were resolved for UML 2.5 using Eclipse OCL [8]. During development of the Eclipse OCL Validity View, the OCL exposition of the UML 2.5 constraints was accidentally executed; in 2013, it was unacceptably slow. By 2017, further development including `allInstances` caches improved the execution time to a couple of minutes. Does `allInstances` need further work for UML?

The UML constraints affecting structural models only have two usages of `allInstances`

4.1. Node.allInstances()

```
context Property
  inv:
    deployment->notEmpty() implies
      owner.oclIsKindOf(Node) and
      Node.allInstances()
        ->exists(n | n.part->exists(p | p = self))
```

The UML metamodel has numerous Property instances, but no deployments and no Nodes. The `deployment->notEmpty()` should be `false` and so the `Node.allInstances()` need never execute. However regular OCL operation calls require source and arguments to be evaluated before the operation is evaluated. When treating `implies` and `and` as regular operations, the guards correctly contribute to results, but they do not inhibit evaluation of their arguments. Hence `Node.allInstances` was needlessly evaluated for every `Property`. Very wasteful when every invocation involved a total model search to locate the zero instances. Only slightly wasteful when repeatedly re-using a cached zero instances.

4.2. Classifier.allInstances()

A Stereotype may only generalize or specialize another Stereotype.

```
context Stereotype
  inv:
    allParents()->forAll(oclIsKindOf(Stereotype)) and
    Classifier.allInstances()
        ->forAll(c | c.allParents()
            ->exists(oclIsKindOf(Stereotype)) implies
              c.oclIsKindOf(Stereotype)
          )
```

The Standard Profile provides about 30 Stereotype instances for which about 70 Classifier instances are permuted so this constraint is in use and worth optimizing. However it does not fit our categorisation. The first line uses `self` so no rewrite to use `self` is necessary. The later use of `Classifier.allInstances()` suggests a rewrite to a `Classifier` context. The root usage of `and` is a hint that two constraints have been combined with the first perhaps guarding against duplicate / inappropriate diagnosis by the second.

The first part can be paraphrased as 'all-super-types are Stereotypes' which is what 'A Stereotype may only generalize or specialize another Stereotype' specifies. The remainder of the constraint following the `and` is an extra independent self-less expression needlessly guarded by the actual constraint. It can be paraphrased as 'all Classifiers that have a Stereotype super-type are Stereotypes'.

This is an example of a multi-invariant; both halves of the `and` would be clearer if expressed separately.

```
inv Part1:
  Part1()
inv Part2:
  Part1() implies Classifier.allInstances()
      ->forAll(c | c.allParents()
          ->exists(oclIsKindOf(Stereotype)) implies
            c.oclIsKindOf(Stereotype)
        )
context Stereotype::Part1() : Boolean
  body: allParents()->forAll(oclIsKindOf(Stereotype))
```

If OCL recognised invariants as operations from `self` to `Boolean`, a constraint would be directly callable:

```
inv Part1:
  allParents()->forAll(oclIsKindOf(Stereotype))
```

```
inv Part2:
  Part1 implies Classifier.allInstances()
      ->forAll(c | c.allParents()
          ->exists(oclIsKindOf(Stereotype)) implies
            c.oclIsKindOf(Stereotype)
        )
```

This improves readability for humans and supports finer-grained diagnostics. It helps tooling prioritize inter-dependent constraints and avoid duplicate diagnosis.

A separate `Classifier` constraint for Part2, however, would inflict `Stereotype` functionality on `Classifier`. Better to consider what we are really trying to constrain. We can then write the modular and symmetrical:

```
context Stereotype
```

A Stereotype may only generalize or specialize another Stereotype.

```
  inv SuperTypesAreStereotypes:
    self.allParents()->forAll(oclIsKindOf(Stereotype))
```

A generalization or specialization of a Stereotype must be another Stereotype,

```
  inv SubTypesAreStereotypes:
    self.allChildren()->forAll(oclIsKindOf(Stereotype))

context Classifier.allChildren() : Set(Classifier)
  body: self.generalization.specific->closure(generalization.specific)

context Classifier.allParents() : Set(Classifier)
  body: self.generalization.general->closure(generalization.general)

context Classifier
  inv AcyclicSuperTypes:
    self.allParents()->excludes(self)
```

In the above, `allParents` rewrites the UML version using `closure` rather than a recursive function call. `allChildren` which is missing from the UML specification uses a similar exposition. The `AcyclicSuperTypes` is a clearer exposition of UML's `no_cycles_in_generalization`; it provides the missing enforcement of 'another Stereotype' for `SuperTypesAreStereotypes` and `SubTypesAreStereotypes`.

The above is much more readable, more modular and faster than the UML exposition; it avoids `allInstances`.

4.3. Recent progress

Subsequent development to sharpen the OCL to Java generation in Eclipse OCL supported per-operation overloads. Only the source argumnent is evaluated to allow the appropriate overload to be resolved. Evaluation of arguments is deferred until the operation requires them. A guarding value can therefore provoke an immediate return without unnecessary argument evaluation. In 2025, the execution time is now down to 10 seconds; useable but 1 second or less is desirable to allow validation after every action in an editor.

This paper is the first of a trio of papers on Collection Optimization. The other two papers address the inefficiencies of individual collections [9] and of cascades [10] of collection operations. Once fully implemented a further improvement in performance is to be expected for collection operations.

Use of the OCL to Java converter in conjunction with a pre-generated OCL Java representation of the UML metamodel should give significant improvements for the general non-collection control flow.

4.4. Further Work

Consideration of the usage of `allInstances()` in the UML specification reveals just 12 usages, 4 of which are gratuitous. 7 look straightforward to rewrite with an appropriate context. The remaining constraint is nearly 20 lines. `ActivityPartition.represents_classifier` is probably in need of some intelligent attention. There is therefore little benefit in developing a faithful automated rewrite for a small amount of suspect code.

The strong discouragement of `allInstances` is well justified and should be retained.

5. Related Work

The Amsterdam Manifesto [4] identified two problems with `allInstances`. The need for a finite domain of instances is resolved by the simple expedient of special casing `allInstances` for `Integer`, `Real` and `String` as `invalid`. The problem of the context in which instances are located remains implementation dependent.

Wei [11] reviews and contrasts `allInstances` caching strategies confirming their utility and assesses a further smart that restricts the eager analysis to relevant containment trees. Disappointing benefits are reported for the extra smart, perhaps because various levels of prolific `allInstances` are assessed rather than the occasional use as in UML. The paper neglects implicit opposites that are amenable to the same analyses and smarts, but not static smarts. Static analysis is not valid if an input model conforms to an extended metamodel. An almost static dynamic load-time analysis is necessary.

An extension to infinite models is discussed by Tisi [12].

Many authors have discouraged use of `allInstances`, many in 'informal publications' [3], [13], [14], [15], [16].

6. Summary

We have identified that naive execution of `allInstances` may involve cubic costs where a linear cost should be achievable.

We have identified that package constraints can temper the cubic costs to quadratic.

We have identified caches that can differently temper the cubic costs to quadratic.

We have identified that self-full rewrites makes `allInstances` redundant. The cost should be linear and no instance caches are required.

We finally conclude that the best solution for `allInstances` remains to discourage it. Authors should rewrite since development of a foolproof automated rewrite is not justifiable.

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] E.D.Willink, 2010. URL: https://www.eclipse.org/forums/index.php?t=msg&th=166482.

[4] S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, A. Wills, The Amsterdam manifesto on OCL, in: Object Modeling with the OCL: The Rationale behind the Object Constraint Language, Springer, 2002, pp. 115–149.

[5] Object Management Group, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3, 2016. Formal/2016-06-03.
[6] Eclipse EMF Project, 2025. URL: https://projects.eclipse.org/projects/\protect\penalty\z@modeling.emf.emf.
[7] 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).
[8] Eclipse OCL Project, 2025. URL: https://projects.eclipse.org/projects/\protect\penalty\z@modeling.mdt.ocl.
[9] E.D.Willink, OCL Collection Optimization - Part 2 - Components, in: submitted to OCL 2025 @ STAF 2025, Koblenz, 2025.
[10] E.D.Willink, OCL Collection Optimization - Part 3 - Cascades, in: submitted to OCL 2025 @ STAF 2025, Koblenz, 2025.
[11] R. Wei, D. S. Kolovos, An efficient computation strategy for allinstances ()., in: BigMDE@ STAF, 2015, pp. 32–41.
[12] M. Tisi, R. Douence, D. Wagelaar, Lazy evaluation for ocl., in: OCL@ MoDELS, 2015, pp. 46–61.
[13] mike, 2017. URL: https://stackoverflow.com/questions/31767506/nested-ocl-foralls.
[14] T. Mossakowski, C. Lüth, 2012. URL: https://www.informatik.uni-bremen.de/agbkb/lehre/ss12/foma/slides/handouts-06.pdf.
[15] J. Chimiak-Opoka, B. Demuth, Teaching OCL Standard Library: First Part of an OCL 2. x Course, Electronic Communications of the EASST 34 (2010).
[16] A. D. Brucker, B. Wolff, A note on design decisions of a formalization of the OCL (2002).