Sorted collection and map types for OCL

Kevin Lano[1], Zishan Rahman[1] and Shekoufeh Rahimi[2]

[1]King's College London, London, UK
[2]University of Roehampton, London, UK

Abstract

The OCL collection types defined in the OMG v2.4 standard are a key part of the language, and corresponding types can now be found in most modern programming languages such as Java, C# and Python. Further aggregate types, particularly maps, have been added in revised OCL versions. In this paper we give a rationale for extending the OCL collection and map type system with intrinsically-sorted versions of these aggregate types. We show that these types have strong semantic properties and that their use can improve the efficiency of generated code.

Keywords

Model-driven engineering, software specification, UML, OCL, sustainable software

1. Introduction

Collection types (sets, sequences, bags and ordered sets) are a core part of the OCL [1]. Together with these types, a powerful set of collection operators such as $\rightarrow$*select* and $\rightarrow$*collect* are also defined in OCL, based on natural mathematical operators for such collections.

The OCL v2.4 standard however has a number of gaps and inconsistencies, especially regarding *OrderedSet*, which appears in the library definitions (Chapter 11) but not in the mathematical OCL semantics (Annex A). Even in the library, several *OrderedSet* operations are incompletely defined. Map aggregate types are now a common feature of programming languages, and are a significant omission from OCL v2.4. This absence has been rectified in extended OCL versions such as Eclipse OCL and AgileUML OCL [2, 3]. Sorted aggregate types are also included in many programming languages, such as Java, C# and C++ (where the default set, bag and map types are sorted). The absence of such types in OCL makes it difficult to use OCL as a semantic representation for reverse and re-engineering of programs [4]. Sorted aggregate types can improve programming efficiency and reduce program energy use in certain cases, thus contributing to the goal of *sustainable software* [5, 6]. Including them in OCL enables design decisions for improved energy-efficiency to be expressed in software models.

In this paper we first review the situation for the established aggregate types (Section 2), and then consider what steps are necessary to incorporate sorted aggregate types into the OCL type system (Section 3). In Section 4 we consider the semantic properties of sorted OCL collections and maps. We also describe how sorted types are supported within the AgileUML toolset [7], and provide a comparative evaluation of the energy efficiency of implementations of sorted and unsorted collections (Sections 5, 6).

2. Standard OCL collection and map types

The standard OCL collection types are $Set(T)$, $Sequence(T)$, $OrderedSet(T)$ and $Bag(T)$ for type parameter T. These represent the four combinations of binary properties of a collection of being *ordered* by addition order (and *indexed* by 1..n) and of having *unique* elements. Map types are aligned to set types in this scheme (Table 1).

Set and sequence types are the central datatypes (along with maps) of formal specification languages such as VDM, B and Z [8], and are supported by most modern programming languages such as C++,

Joint Proceedings of the STAF 2025 Workshops: OCL, OOPSLE, LLM4SE, ICMM, AgileMDE, AI4DPS, and TTC. Koblenz, Germany, June 10–13, 2025.

kevin.lano@kcl.ac.uk (K. Lano); zishan.rahman@kcl.ac.uk (Z. Rahman); shekoufeh.rahimi@roehampton.ac.uk (S. Rahimi)
0000-0002-9706-1410 (K. Lano)

published 2025-12-10

C#, Java and Python. OCL sequences are not a semantic subtype of OCL sets, because the addition-order-independence axiom

$$(1): \ \forall s : Set(T);\ \forall x, y : T \cdot \\ s{\rightarrow}including(x){\rightarrow}including(y) \ = \ s{\rightarrow}including(y){\rightarrow}including(x)$$

is true for sets but not for sequences.

A bag or multiset (unordered and unindexed but possibly with duplicates) is a less-utilised datatype than sets or sequences. Bags have not been included in the core Java collection libraries (up to the current Java version 24), and are not a core facility of Python. However, C++ had multisets in the STL from C++ '99 onwards.

Bags do satisfy axiom (1) above, and hence are not a semantic supertype of sequences. They are not a semantic subtype of sets because the uniqueness axiom

$$(2): \ \forall s : Set(T);\ \forall x : T \ \cdot \ s{\rightarrow}includes(x) \ \Rightarrow \ s{\rightarrow}including(x) = s$$

is true for sets but not for bags.

The concept of ordered sets is less clear than the other collection types. They combine three aspects: (i) of being indexed, (ii) of being ordered by addition order, (iii) of uniqueness. Programming languages sometimes support the concept of an addition-ordered unique collection without indexes, such as *LinkedHashSet* in Java, and JavaScript *Set*. Support for an OCL-style 'ordered set' is more unusual: the maintenance of indexes requires an additional computational overhead. OCL ordered sets (with or without indexes) are not a semantic subtype of OCL sequences, because they do not satisfy the sequence axiom

$$(3): \ \forall s : Sequence(T);\ \forall x : T \ \cdot \ s{\rightarrow}including(x){\rightarrow}last() = x$$

moreover, they are not a semantic subtype of sets or bags, because they fail to satisfy axiom (1) [9]. Thus all four OCL collection types are unrelated by semantic subtyping. The name 'ordered set' is misleading in this respect, because it suggests that a subtyping relation exists with sets. 'Unique sequence' could be a better name (as it makes explicit that there are indexes and ordering by addition) but it also suggests subtyping of *Sequence*.

The definition of several *OrderedSet* operators in the OCL standard library (Section 11.7.3 of [1]) are incomplete or incorrect:

- $append(obj)$, $prepend(obj)$ and $insertAt(obj)$ can break the uniqueness property of the ordered set.
- No $\rightarrow including$ operation is defined for *OrderedSet*. This operation should leave the collection unchanged if the supplied element is already present, otherwise it is added as the last element (as in Section 3.25 of [3]).
- *reverse* is incompletely specified.
- Several operations are stated to 'redefine the *Set* operation', but since *OrderedSet* is not a *Set* subtype, this is not correct.

Maps or 'associative arrays' have been used in programs since the early days of programming languages, e.g., in SNOBOL4, and are now supported in Java, Python and many other modern languages. The basic map type corresponds to the set collection type, as maps are unordered/unindexed sets of mappings, without duplicates. It may be useful to define other varieties of maps, that provide sequence-like and bag-like maps, with ordering or indexes and with possible duplicate mappings, as shown in Table 1. Ordered maps are supported by Python and JavaScript (*OrderedDict* and *Map* types, respectively, although these maps are ordered without indexing), and multimaps by C++. Ordered indexed maps feature in the VB language, as the *Collection* datatype [10].

As may be expected, these four types of map are not related to each other by subtyping, for similar reasons to the situation with the corresponding set-based collections. Ordered maps are order-sensitive with respect to element (pair) addition, whilst unordered maps are not, and unique-element maps are unchanged by repeated additions of the same element, whilst multimaps are changed by each addition.

Table 1
Unsorted collection and map datatypes

	unindexed	*indexed*
non-unique	Bag MultiMap	Sequence OrderedMultiMap
unique	Set Map	OrderedSet OrderedMap

3. Sorted collections and maps

Sorting collections of data into a 'natural' order is an essential computational operation, which facilitates human interaction with the data, and makes large-scale searching and counting operations significantly more efficient. For these reasons, many programming languages have incorporated intrinsically-sorted collections, including the *set*, *multiset*, *map* and *multimap* types of C++, *SortedSet* and *SortedDictionary* in C#, and *SortedSet*, *TreeSet*, *SortedMap* and *TreeMap* types in Java. Other Java collection libraries such as Apache Collections4 provide types such as *SortedBag* and *TreeBag*. The Python *sortedcontainers* library provides sorted maps, lists and sets. The Python *SortedSet* also has indexes, so can be used as a sorted ordered set, although with $O(log\ n)$ index access.

The order relation $<$ used for sorting should be a total order on the parameter type T. Examples are numeric orderings on *Integer* and *Real*, and lexicographic ordering on *String*. For class types T used as the argument type of a sorted collection/map, it is necessary that T provides an operation $compareTo(other : T) : Integer$ which defines a total order $<$ on T by the equivalence $c1 <_T c2 \equiv c1.compareTo(c2) < 0$. In terms of [3], this means that T extends *OCLComparable*[1].

Of the sorted types, sorted sets and sorted maps are the most widely-supported in programming languages. These types are also semantic subtypes of the general unsorted set and map types (Figure 1). *SortedBag* is also a subtype of *Bag*. These subtyping relations hold because the sorted order of a sorted set/bag/map does not affect the key properties of addition-order independence or uniqueness (for sets and maps) of these aggregate types.

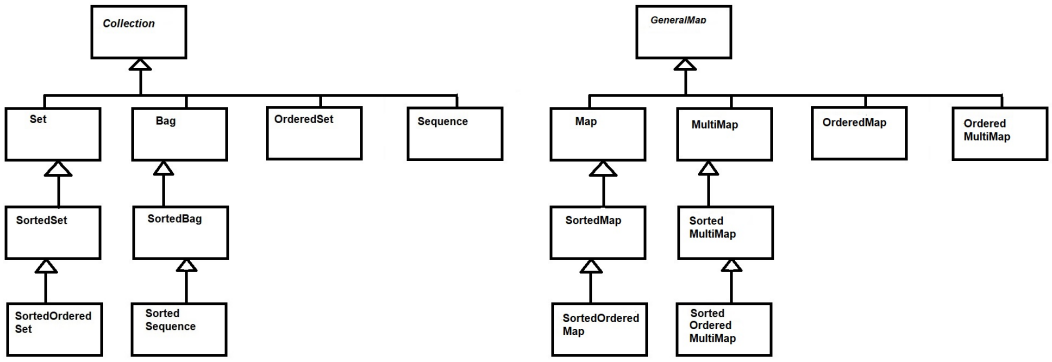

Figure 1: Extended OCL collections and map types

For a set $st : Set(T)$, the possible sequentialisations of st are defined by the $st{\rightarrow}asSequence()$ operator. The result of this operator could be any $sq : Sequence(T)$ which has

$$sq{\rightarrow}includesAll(st)\ and\ st{\rightarrow}includesAll(sq)\ and\ sq{\rightarrow}size() = st{\rightarrow}size()$$

For a sorted set $ss : SortedSet(T)$, the possible sequentialisations $ss{\rightarrow}asSequence()$ are sequences ssq which list the ss elements in $<$-sorted order. These are also possible set sequentialisations of ss, thus

[1]NB: *compareTo* is not generally commutative, as stated in [3]. Instead, $c1.compareTo(c2) = 0$ should imply $c1 = c2$.

the *SortedSet* version of the operator is semantically consistent with the *Set* version, and the conditions for Liskov substitutability are satisfied [11]. However, sorted sequences are not a subtype of sequences, because the sequence axiom (3) of Section 2 no longer holds for sorted sequences: $ss{\rightarrow}including(x)$ for sorted sequence ss adds x into the position required to preserve the sorted order of ss, not necessarily at the sequence end. Likewise, the axiom (3) in the modified form:

$$(3a): \ \forall s : OrderedSet(T);\ \forall x : T \cdot \\ s{\rightarrow}excludes(x) \ \Rightarrow \ s{\rightarrow}including(x){\rightarrow}last() = x$$

holds for ordered sets, but not for sorted ordered sets. Instead, sorted ordered sets are a subtype of sorted sets, and sorted sequences are a subtype of sorted bags: the subtypes add indexing to the base types.

An alternative organisation of OCL collections would be to define an abstract *SequencedCollection* type, for collections with a definite iteration order and operations $\rightarrow first$, $\rightarrow last$, $\rightarrow front$, $\rightarrow tail$, but not necessarily with an index. The order could either be addition order (*SequencedSet* or *SequencedBag*, with indexed subtypes *OrderedSet* and *Sequence*) or sorted order (*SortedSet*, *SortedBag* and indexed subtypes). An *UnsequencedCollection* type would group *Set* and *Bag* (Figure 2).

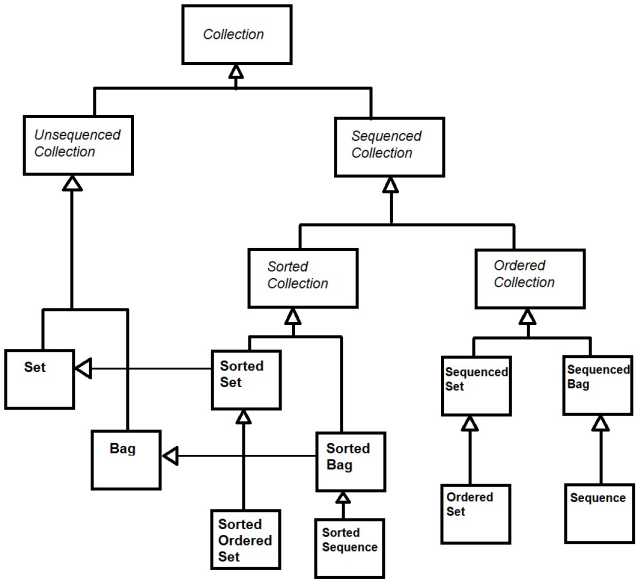

Figure 2: Alternative OCL collections types

4. Semantics for sorted collections

Sorted versions of *Set*, *Bag*, *OrderedSet*, *Sequence* and *Map* datatypes would be of utility in enabling design choices to be made explicit in software models (i.e., to maintain a collection as sorted to optimise search operations such as *includes*, *count*, etc). The plain *SortedSet*, *SortedBag* and *SortedMap* types represent the situation where indexing is not available but the efficiency advantages of sortedness are needed, together with a consistent iteration order. A complete family of sorted versions of collection and map types could be as in Table 2.

4.1. SortedSet

SortedSet is a collection type constructor which defines a subtype $SortedSet(T)$ of $Set(T)$ for each parameter type T that supports an operator $<_T$ defining a total order on T. The elements of a sorted

Table 2
Sorted collection and map datatypes

	unindexed	*indexed*
non-unique	SortedBag SortedMultiMap	SortedSequence SortedOrderedMultiMap
unique	SortedSet SortedMap	SortedOrderedSet SortedOrderedMap

set $s : SortedSet(T)$ occur in increasing order with respect to the $<$ order on T. Duplicate elements are not permitted. The following axiom holds for sorted sets:

$$(4) :$$
$$\forall\, s : SortedSet(T)\ \cdot\ let\ sq : Sequence(T) = s{\rightarrow}asSequence()\ in$$
$$\forall\, i : 1..(sq{\rightarrow}size() - 1)\ \cdot\ \ sq[i]\ <_T\ sq[i+1]$$

All the operations of *Set* in the OCL standard library also apply without change to *SortedSet*, except that $\rightarrow union$, $\rightarrow including$ and $\rightarrow symmetricDifference$ must be modified to maintain and possibly extend the sorted order of their source argument in the result. This means that the result collection contains order relations between retained and added elements, and between added elements, in addition to the existing order relations of retained elements. Operators $\rightarrow intersection$, $\rightarrow excluding$, $\rightarrow select$, $\rightarrow reject$ and $-$ simply need to preserve the existing source collection order of the retained elements (Table 3). $C(T)$ denotes any collection type.

Table 3
OCL collection operators for *SortedSet*

OCL Set operator	*Signatures for SortedSet*
$s{\rightarrow}union(s1)$	$SortedSet(T) \times C(T)\ \rightarrow\ SortedSet(T)$
$s{\rightarrow}intersection(s1)$	$SortedSet(T) \times C(T)\ \rightarrow\ SortedSet(T)$
$s{\rightarrow}excluding(x)$	$SortedSet(T) \times T\ \rightarrow\ SortedSet(T)$
$s{\rightarrow}including(x)$	$SortedSet(T) \times T\ \rightarrow\ SortedSet(T)$
$s - s1$	$SortedSet(T) \times C(T)\ \rightarrow\ SortedSet(T)$
$s{\rightarrow}select(P)$	$SortedSet(T) \times Function(T, Boolean)\ \rightarrow\ SortedSet(T)$
$s{\rightarrow}reject(P)$	$SortedSet(T) \times Function(T, Boolean)\ \rightarrow\ SortedSet(T)$
$s{\rightarrow}collect(e)$	$SortedSet(T) \times Function(T, R)\ \rightarrow\ Bag(R)$

4.2. SortedBag

SortedBag is a collection type constructor which defines a subtype $SortedBag(T)$ of $Bag(T)$ for each parameter type T that supports an operator $<_T$ defining a total order on T. The elements of a sorted bag $s : SortedBag(T)$ occur in non-decreasing order with respect to the $<$ order on T. Duplicate elements are permitted.

All the operations of *Bag* in the OCL standard library also apply without change to *SortedBag*, except that $\rightarrow union$, $\rightarrow including$ and $\rightarrow symmetricDifference$ are redefined to preserve the sorted order of their source argument elements in their result. Operators $\rightarrow intersection$, $\rightarrow excluding$, $\rightarrow select$, $\rightarrow reject$ and $-$ automatically have this property, analogously to Table 3.

4.3. SortedSequence

SortedSequence is a collection type constructor which defines a subtype $SortedSequence(T)$ of $SortedBag(T)$ for each parameter type T that supports a total order $<$[2]. The elements of a sorted

[2]The name *IndexedSortedBag* could be used instead of *SortedSequence*.

sequence $s : SortedSequence(T)$ occur in non-decreasing index order with respect to the $<$ order on T. Duplicate elements are permitted.

Some of the OCL standard library *Sequence* operators can be used without change for sorted sequences, because their definitions preserve the sortedness of the source collection in the operator result (Table 4). As for sets and bags, operators which remove source elements and which do not add or reorder elements (such as intersection, subtraction, *front*, *tail*, *select* and *reject*) can be defined to maintain the sortedness of the source in their result by simply retaining the existing order of retained source elements.

However *reverse*, *insertAt*, *append* and *prepend* do not preserve sortedness.

For a sorted sequence, $\rightarrow including$ is redefined to insert the new element in the appropriate position so that the result is $<$-sorted. $\rightarrow union$ iterates $\rightarrow including$.

Table 4
OCL collection operators for *SortedSequence*

OCL Sequence operator	*Preserves sortedness*	*Signatures for SortedSequence*
$s \rightarrow union(s1)$	$\times$	$SortedSequence(T) \times C(T) \rightarrow SortedSequence(T)$
$s \rightarrow intersection(s1)$	$\surd$	$SortedSequence(T) \times C(T) \rightarrow SortedSequence(T)$
$s \rightarrow excluding(x)$	$\surd$	$SortedSequence(T) \times T \rightarrow SortedSequence(T)$
$s \rightarrow including(x)$	$\times$	$SortedSequence(T) \times T \rightarrow SortedSequence(T)$
$s \rightarrow subSequence(i, j)$	$\surd$	$SortedSequence(T) \times Integer \times Integer \rightarrow SortedSequence(T)$
$s - s1$	$\surd$	$SortedSequence(T) \times C(T) \rightarrow SortedSequence(T)$
$s \rightarrow select(P)$	$\surd$	$SortedSequence(T) \times Function(T, Boolean) \rightarrow SortedSequence(T)$
$s \rightarrow reject(P)$	$\surd$	$SortedSequence(T) \times Function(T, Boolean) \rightarrow SortedSequence(T)$
$s \rightarrow collect(e)$	$\times$	$SortedSequence(T) \times Function(T, R) \rightarrow Sequence(R)$
$s \rightarrow front()$	$\surd$	$SortedSequence(T) \rightarrow SortedSequence(T)$
$s \rightarrow tail()$	$\surd$	$SortedSequence(T) \rightarrow SortedSequence(T)$

4.4. SortedOrderedSet

SortedOrderedSet[3] is defined analogously to *SortedSequence*, with the additional uniqueness property (2). The operator signatures for *SortedOrderedSet* operators correspond to those of *SortedSequence* (Table 4), with *SortedSequence* replaced by *SortedOrderedSet*.

4.5. SortedMap

Maps of all kinds have the operators given in Table 5. Most of the operators preserve sortedness of the first argument in their results, but do not preserve indexing, that is, applying the operator to an ordered map would require re-indexing of the result. $\rightarrow union$ (Map override) is redefined for sorted maps in order to preserve sortedness. The operation $m \rightarrow including(k, v)$ of [3] is the same as $m \rightarrow union(Map\{k \mapsto v\})$, and $m \rightarrow excluding(k, v)$ is the same as $m - Map\{k \mapsto v\}$. Table 5 gives the signatures of the operators on *SortedMap*. $M(S, T)$ denotes any type of map.

5. Implementations of sorted types

We have added support for *SortedSet*, *SortedMap* and *SortedSequence* types to AgileUML OCL from Version 2.4[4]. In general three alternative strategies can be used to extend OCL implementations with additional datatypes:

[3] Alternatively, this type could be called *IndexedSortedSet*.
[4] A parser for AgileUML OCL is defined at github.com/antlr/grammars-v4

Table 5
SortedMap operators

AgileUML map operator	*Preserves sortedness*	*Signatures for SortedMap*
$m{\rightarrow}union(m1)$	$\times$	$SortedMap(S,T) \times M(S,T) \rightarrow SortedMap(S,T)$
$m{\rightarrow}intersection(m1)$	$\surd$	$SortedMap(S,T) \times M(S,T) \rightarrow SortedMap(S,T)$
$m{\rightarrow}excludingKey(k)$	$\surd$	$SortedMap(S,T) \times S \rightarrow SortedMap(S,T)$
$m{\rightarrow}excludingValue(v)$	$\surd$	$SortedMap(S,T) \times T \rightarrow SortedMap(S,T)$
$m{\rightarrow}restrict(ks)$	$\surd$	$SortedMap(S,T) \times C(S) \rightarrow SortedMap(S,T)$
$m{\rightarrow}antirestrict(ks)$	$\surd$	$SortedMap(S,T) \times C(S) \rightarrow SortedMap(S,T)$
$m - m1$	$\surd$	$SortedMap(S,T) \times M(S,T) \rightarrow SortedMap(S,T)$
$m{\rightarrow}select(P)$	$\surd$	$SortedMap(S,T) \times Function(T, Boolean) \rightarrow SortedMap(S,T)$
$m{\rightarrow}reject(P)$	$\surd$	$SortedMap(S,T) \times Function(T, Boolean) \rightarrow SortedMap(S,T)$
$m{\rightarrow}collect(e)$	$\times$	$SortedMap(S,T) \times Function(T,R) \rightarrow Bag(R)$
$m{\rightarrow}keys()$	$\surd$	$SortedMap(S,T) \rightarrow SortedSet(S)$
$m{\rightarrow}values()$	$\times$	$SortedMap(S,T) \rightarrow Bag(T)$

1. Add the type as a new intrinsic type to the OCL type system, and implement it by code generation rules that translate the type and its operations to corresponding types/operations in the target programming language.
2. Add the type as an intrinsic type, but provide language-specific implementations where the target programming language does not provide direct support for the type.
3. Define the type within OCL as a derived type based on existing OCL types. The new type and its operations can be specified as a new library component, for which code can be generated using the pre-existing code generators.

The first option is appropriate when the type is widely-supported by programming languages. *SortedSet* and *SortedMap* satisfy this condition, and hence can be directly added to the OCL type system. For more specialised types with partial implementation language support we use the second approach, and provide language-specific implementations *SortedSequence* and *SortedOrderedSet* definitions for Java, C# and C++[5].

For *OrderedMap* the third approach is adopted. This type can be defined in UML/OCL as a generic class *OrderedMap*$<K, T>$ based on $Sequence(K)$ and $Map(K,T)$. Likewise, $MultiMap(K,T)$ can be simulated by $Map(K, Bag(T))$ [12].

6. Evaluation

To evaluate the possible energy-efficiency benefits of the sorted types, we compare the energy use of Sequence-based and SortedSet/SortedSequence versions of a specification which adds N distinct integers to an empty collection (scenario 1) or makes 10000 membership queries to collections of different sizes (scenario 2). We use the Green Algorithms tool for energy measurement [13]. Evaluation data is available at Zenodo.org[6].

For scenario 1 the growth of energy use is sublinear in the number of insertions for both sequence and sorted set versions (using Java implementations on Windows 10 or Linux platforms). The sorted set version has higher energy usage, and the sorted sequence version has a higher energy use growth rate than either of these (Figure 3). In scenario 2 there is a linear growth in energy use with the size of the collection in the sequence version, and almost constant energy use in the sorted collection versions (Figure 4).

[5] github.com/eclipse-agileuml/agileuml/libraries
[6] zenodo.org/records/15387228

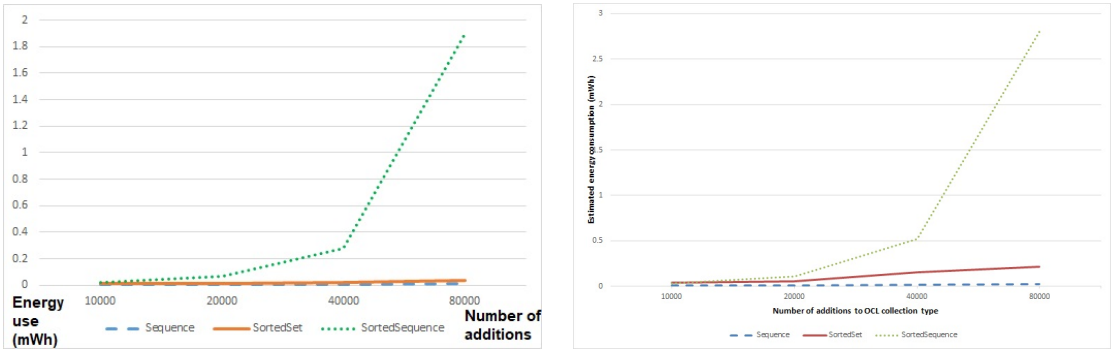

Figure 3: Sequence/sorted collection energy use (mWh) for element additions: (a) Java, Windows 10; (b) Java, Linux

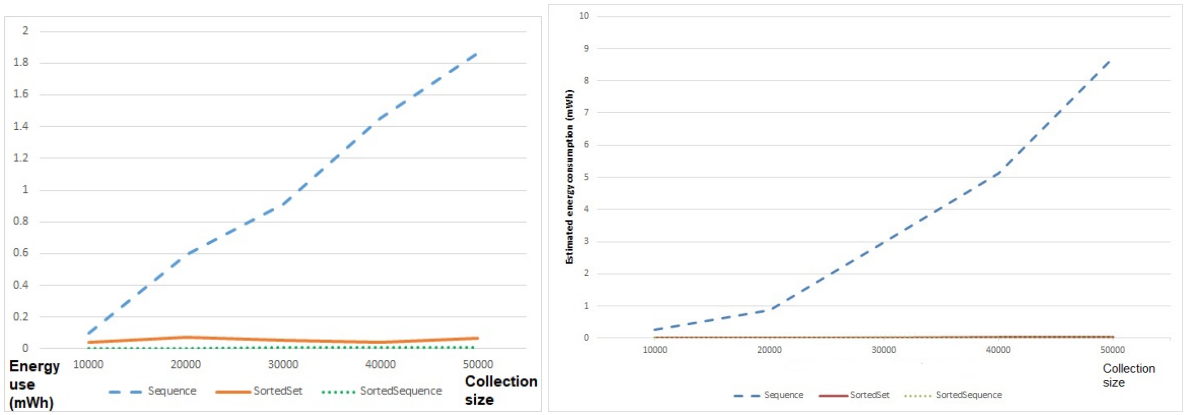

Figure 4: Sequence/sorted collection energy use (mWh) for 10000 membership tests: (a) Java, Windows 10; (b) Java, Linux

If we combine the results for N additions (Figure 3 (a)) and 10000 queries (Figure 4 (a)), we find that sorted sets and sequences are more energy-efficient than unsorted sequences in these size ranges, because the improved efficiency for searches outweighs the increase in cost for additions (Table 6).

Table 6
Energy use (mWh) of N additions + 10000 searches for collections, Java, Windows 10

N	*Sequence*	*SortedSet*	*SortedSequence*
10000	0.104	0.05	0.027
20000	0.596	0.086	0.074
40000	1.456	0.059	0.282

Similar results are obtained for C++ and Python implementations generated from the same specifications (Figures 5, 6). However in the case of Python, sorted sequences are more efficient for additions than sorted sets.

7. Related work

Various other proposals have been made to enhance OCL to add new datatypes, including collection and map types [14, 15, 12]:

- To introduce a *Map* library type [14, 15, 12] – this has been added to AgileUML OCL [2], and to Eclipse OCL [3].
- Function types and lambda expressions [15] – added to AgileUML OCL [2].

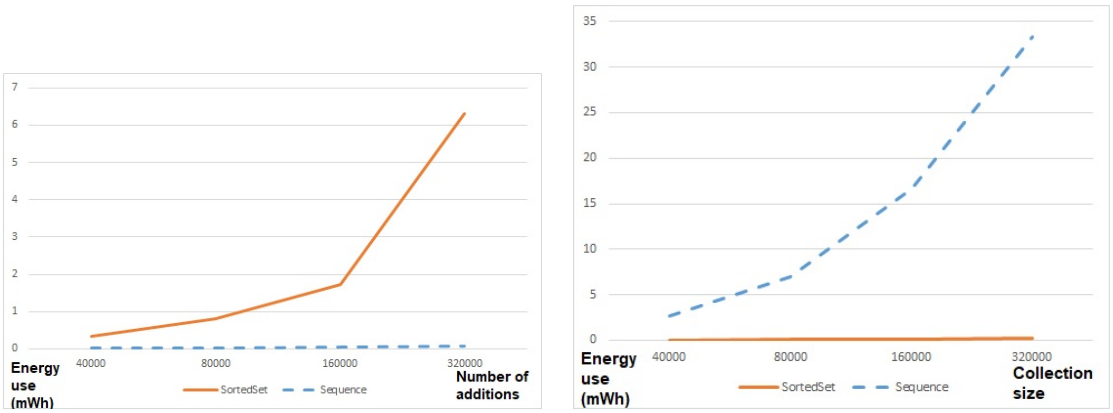

Figure 5: Sequence/sorted set energy use, Windows 10, C++: (a) element additions, (b) 10000 membership tests

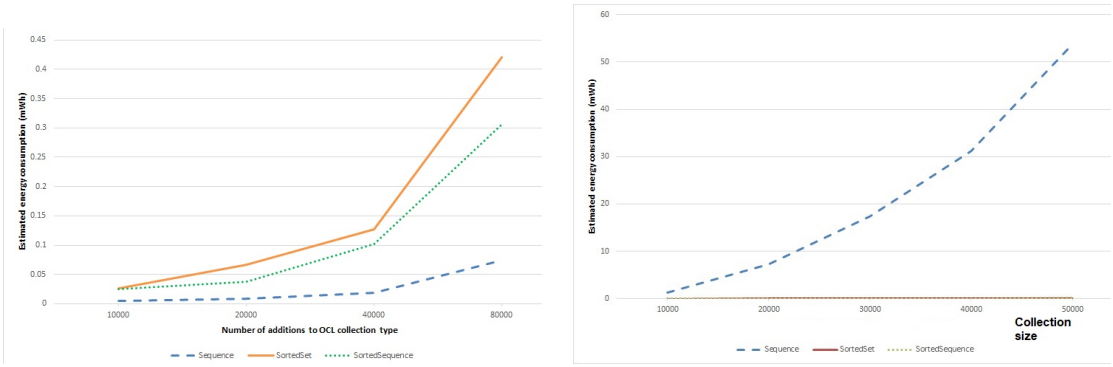

Figure 6: Sequence/sorted collection energy use, Linux, Python: (a) element additions, (b) 10000 membership tests

- Generic types [15] – a proposal for these is defined in [4].
- Libraries [14, 15] – a core set of OCL libraries have been defined as classes in AgileUML: MathLib, OclType, OclProcess, OclFile, OclRandom, etc [16]. These include implementations of *SortedSequence* and *SortedOrderedSet* collection classes in Java and C#.

Our initial motivation to further extend the collection and map types was to support the semantic analysis of programs [4]. In particular, *OrderedMap* is needed to represent the VB Collection and Python OrderedDict types, and sorted sets and maps are needed to represent the Java sorted set and map types. Subsequently, we investigated the relevance of sorted collection types for the achievement of sustainable software – software with reduced energy use [5]. It was found that sorted collections could provide more energy-efficient code in certain scenarios of use, as described in Section 6.

8. Conclusion

In this paper, we have proposed a full range of Ordered and Sorted variants of collection types and maps, and detailed the semantic aspects of these types. These extensions correspond to advanced features of modern programming languages, and can increase the expressiveness and efficiency of OCL for software specification and software production. We have shown how the sorted datatypes can be incorporated into the OCL standard library, and we have provided implementations and evaluations of several of these types.

Declaration on Generative AI

The authors declare that no generative artificial intelligence tools were used to generate text, figures, code, data, analyses, or reviews for this submission.

References

[1] Object Management Group, Object Constraint Language (OCL) 2.4 specification, 2014.
[2] K. Lano, S. Kolahdouz-Rahimi, Extending OCL with map and function types, in: FSEN 2021, 2021.
[3] Eclipse, 2022. Eclipse OCL Version 6.4.0, https://projects.eclipse.org/projects/ modeling.mdt.ocl.
[4] K. Lano, H. Siala, Using OCL for verified re-engineering, in: MoDeVVa 2024, MODELS 2024, 2024.
[5] K. Lano, L. Alwakeel, Z. Rahman, Software modelling for sustainable software engineering, in: 2nd Agile MDE Workshop, STAF 2024, 2024.
[6] S. Naumann, M. Dick, E. Kern, T. Johann, The GREENSOFT model: a reference model for green and sustainable software and its engineering, Sustainable Computing: Informatics and Systems 1 (2011) 294–304.
[7] K. Lano, The AgileUML manual, 2025. Www.agilemde.co.uk/umlrsds20.pdf.
[8] J. Spivey, The Z notation: A reference manual, Prentice Hall, 1989.
[9] F. Buttner, M. Gogolla, L. Hamann, M. Kuhlmann, A. Lindow, On better understanding OCL collections: an OCL Ordered Set is not an OCL Set, in: MODELS 2009 Workshops, LNCS 6002, 2010, pp. 276–290.
[10] Microsoft Corp., Office VBA Reference, 2022. Https://learn.microsoft.com/en-us/office/vba/api/overview.
[11] B. Liskov, J. Wing, A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. 16 (1994) 1811–1841.
[12] E. Willink, An OCL map type, in: OCL 2019 Workshop, 2019. URL: https://eclipse.dev/ocl/docs/publications/OCL2019MapType/OCLMapType.pdf.
[13] L. Lannelongue, J. Grealey, M. Inouye, Green algorithms: Quantifying the carbon footprint of computation, Advanced Science 8 (2021).
[14] L. Hamann, M. Gogolla, M. A. Lail, Categorization of approaches to extend and reuse OCL, in: MODELS '22 Companion, ACM, 2022.
[15] E. Willink, Reflections on OCL 2.0, Journal of Object Technology V (2011) 1–29.
[16] K. Lano, S. Kolahdouz-Rahimi, K. Jin, OCL libraries for software specification and representation, in: OCL 2022, MODELS 2022 Companion Proceedings, 2022.