Refactoring Collections in OCL

Martin Gogolla[1], Loli Burgueño[2] and Antonio Vallecillo[3]

[1] *University of Bremen, Germany.*
[2] *Open University of Catalonia, Spain.*
[3] *ITIS Software, Universidad de Málaga, Spain.*

Abstract

The current OCL 2.4 specification organizes collections in one abstract class, `Collection(T)`, and four concrete subclasses, namely `Set(T)`, `Bag(T)`, `Sequence(T)`, and `OrderedSet(T)` depending on whether the collection elements are ordered or not, and whether duplicated elements are allowed or not. These four classes provide a clear and useful partition of the whole collections space, covering all relevant aspects. However, the specification of the operations associated with these classes is rather unwieldy and inefficient in the current standard: it contains duplicated descriptions, missing operations and unspecified details. In this paper, we analyze the problems with such specifications, and propose an alternative specification that avoids duplication and missing details based on the introduction of the appropriate intermediate abstract classes that capture the common features of interest of each kind of collection.

Keywords

UML, OCL, Collection, Set, Bag, Sequence, OrderedSet

1. Introduction

Collections are fundamental data structures in any modeling or programming language [1], they allow expressing how elements are grouped according to different policies, and the valid operations that can be applied to them. The current OCL standard [2] defines four basic kinds of collections, namely `Sequence`, `OrderedSet`, `Bag` and `Set`, depending on whether the order of their elements matters or not, and whether duplicated elements are allowed or not. They are expressive enough for representing the usual groups of elements appearing in the specification of any system model, and provide a rich set of operations on the collections for querying and updating them. Furthermore, OCL 2 collections can be nested, i.e., elements of collections can be other collections, and the `collectNested` iterator and the `flatten` operation were introduced in OCL 2 to deal with them.

However, a common problem that any OCL modeler has suffered has to do with the way in which OCL operations on collections are specified in the standard – probably because of the organization of the document, and how the collection classes are structured in a single

20th International Workshop on OCL and Textual Modeling, June 25, 2021
gogolla@uni-bremen.de (M. Gogolla); lburguenoc@uoc.edu (L. Burgueño); av@uma.es (A. Vallecillo)
http://www.db.informatik.uni-bremen.de/~gogolla/ (M. Gogolla);
https://som-research.uoc.edu/loli-burgueno/ (L. Burgueño); http://www.lcc.uma.es/~av/ (A. Vallecillo)
0000-0003-4311-1117 (M. Gogolla); http://orcid.org/0000-0002-7779-8810 (L. Burgueño);
0000-0002-8139-9986 (A. Vallecillo)

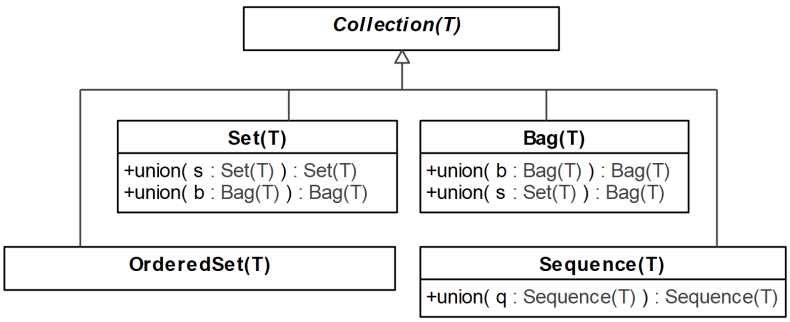

Figure 1: Operation `union()` in current OCL Collections specification.

hierarchy. For example, many common operations such as `count()`, `includes()` or "`=`" are repeated in all classes, including the general and abstract class `Collection(T)`. Other operations seem to be missing, such as `indexSetOf()` in class `Sequence`, or `union()` in class `OrderedSet`. Further details are unclear or lacking, too, like the return type of the `closure()` operation.

To illustrate one of these issues, let us consider operation `union()` (see Fig. 1). First, it is not declared in the superclass `Collection(T)`, but only in the concrete subclasses (unlike other operations, which are defined in both the superclass and the subclasses). Besides, it is not defined for `OrderedSet(T)`. The reader of the OCL specification may wonder whether there is a way to define `union()` in `Collection(T)` and to indicate that the concrete subclasses will define and implement specialized versions of `union()`. Furthermore, the two signatures for the operation defined in `Set(T)` and `Bag(T)` seem superfluous. The problem is that when a standard (OCL, in this case) provides an irregular and seemingly illogical specification with no clear justification, users and tool builders start defining their own extensions and the interoperability and rest of the advantages of using international standards is lost. In fact, we have observed that current OCL engines (in particular, Acceleo [3], USE [4] and Eclipse OCL [5]) all implement the `union()` operation on collections differently.

For example, in addition to the standard operations, Acceleo defines two new `union()` operations on `OrderedSet(T)`:

```
OrderedSet(T):: union(bag : Bag(T)) : Bag(T)
OrderedSet(T):: union(set : Set(T)) : Set(T)
```

However, USE defines only one but with a different signature:

```
OrderedSet(T):: union(oset : OrderedSet(T)) : OrderedSet(T)
```

Finally, Eclipse OCL uses a different approach and defines one operation for the superclass, which always returns a `Bag(t)`, and then another operation for an intermediate abstract collection, called `UniqueCollection(T)` (which gathers `Set(T)` and `OrderedSet(T)`), and that returns a `Set(T)`:

```
Collection(T)::union(c : Collection(T)) : Bag(T)
UniqueCollection(T)::union(s : UniqueCollection(T)) : Set(T)
```

In summary, three different and incompatible implementations of the same operation.

The goal of this paper is to explore how to specify the operations of OCL collections in a clear, regular and complete manner, and to organize them understandably and efficiently. We will show how it is possible to refactor the OCL collections class hierarchy and to present it using a graphical model, with clear OCL laws, new auxiliary types, and adequate syntactic restrictions on operations.

A central idea is to introduce four new subclasses/subtypes of `Collection(T)` that partition the Collections space, depending on whether the collection elements are ordered or not (`OrderAwareCol(T)` and `OrderBlindCol(T)`, resp.) and whether they allow duplicated elements or not (`FrequencyAwareCol(T)` and `FrequencyBlindCol(T)`, resp.), see Fig. 3.

These new subclasses will allow us to refactor the OCL operations on collections in a clear, regular and complete manner, as done for example in the Eclipse OCL implementation [5]. In addition, another very important use of these subclasses is to enable the specification of the result type of some iterators more precisely than in the current OCL standard. In particular, assuming `COL` of type `Collection(T)`, we can define:

```
COL->collect(e  | exprS) : FrequencyAwareCol(T2)
COL->sortedBy(e | exprI) : OrderAwareCol(T)
COL->closure(e  | exprT) : FrequencyBlindCol(T)
```

where the types of the expressions of the bodies of these iterators are the following: `exprS:T2`, `exprI:{Int|Real|String}`, and `exprT:T`.

The structure of this document is as follows. After this introduction, Sect. 2 highlights some of the major problems we have found when working with the Collections operations defined in the OCL 2 specification [2] and presents our proposal to address them. Then, Sect. 3 discusses related works, and in particular how some of the existing OCL engines have implemented the current standard with regard to Collection operations. We also compare it with the Java 8 [6] data structures and operations for dealing with collections. Finally, Sect. 4 concludes with an outline of future work.

2. Refactoring OCL Collections and Operations

Figure 2 gives an overview on the collection kinds and operations in the current OCL standard. Please note that we will use the notions *collection kind* and *collection type*. `Set(T)` and `Bag(T)`, for example, are called collection kinds, whereas `Set(Person)`, `Set(Real)` and `Bag(String)` are called collection types; thus one collection kind with type parameter induces many collection types with concrete type parameters [7]. In the figure, we have followed the order of the classes and operations as mentioned in the standard. The black triangle indicates parts that need discussion.

In our view, we identify the following issues in the current OCL standard w.r.t. collection kinds and their operations.

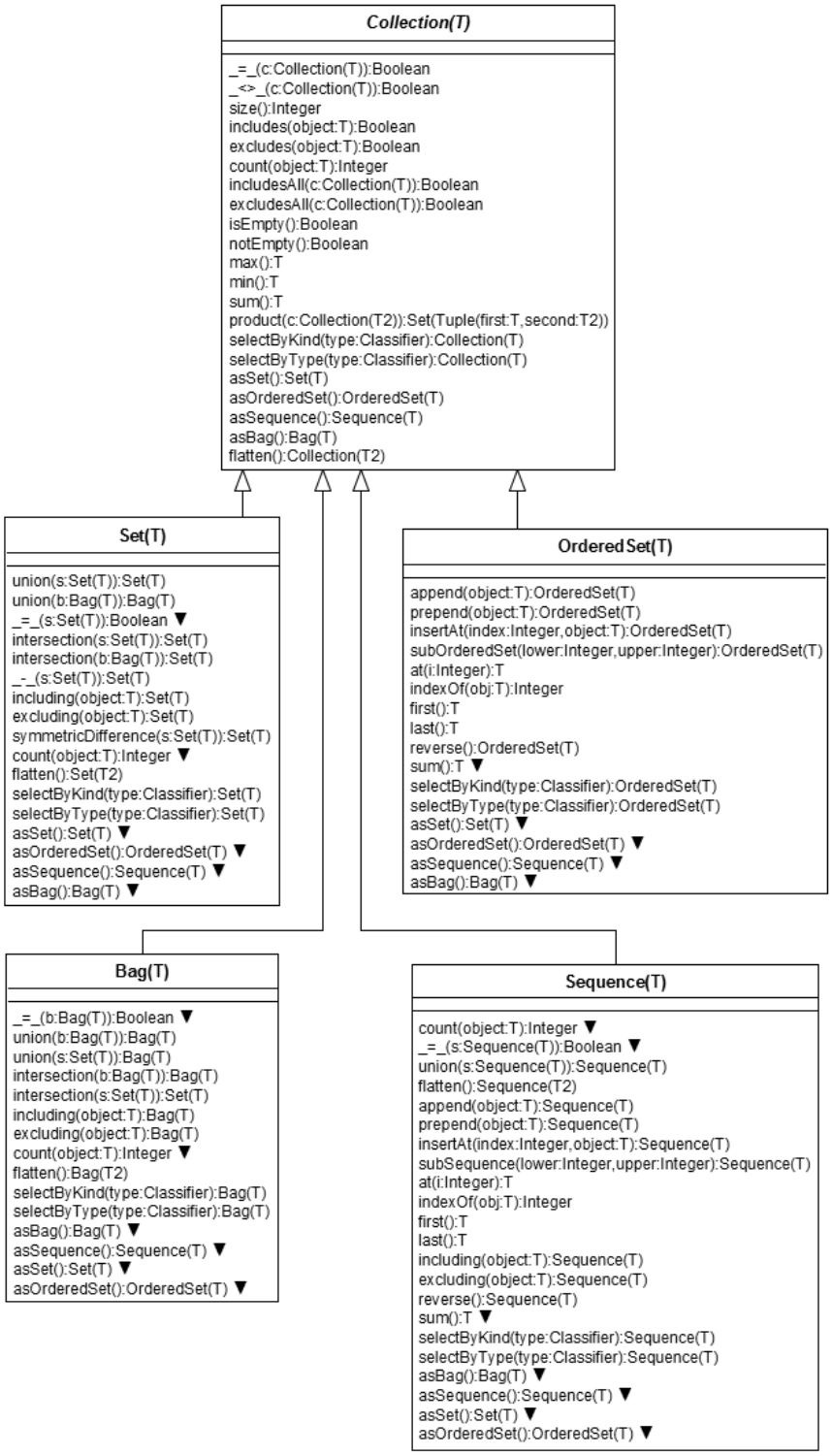

Figure 2: Collection Kinds and Operations in OCL 2.4.

Missing class diagram: In the standard, there is no class diagram giving an overview like Fig. 2, but the classes and operations are described in a textual and partly tabular way over 20 pages.

Repeated and unnecessary operations: In the collection kind `Set(T)`, we identify operations equality `=`, `count()`, `asSet()`, `asOrderedSet()`, `asSequence()`, and `asBag()`. These operations are already present in `Collection(T)`, and the additional declaration does not provide any additional information. It is true that these operations must be implemented in the specialized collection kind `Set(T)`, but the standard should clearly distinguish between operation declaration and the details of operation implementation. Analogous arguments are valid for the same operations for other collection kinds, if the operations are mentioned there—for example, equality (`=`) is not mentioned in `Set(T)` and `OrderedSet(T)`, but it is defined for the rest of the collections—this is somehow confusing. Contrarily, operation `<>` is only defined for the top-most class, `Collection(T)` but not for the rest.

Inconsistent operation order and occurrence: The same operation being defined on all collection kinds is mentioned at different positions in the standard, and thus they appear in our class diagram at different places, e.g., the operation equality (`=`) is defined for Collection(T) and for Bag(T) but not for the rest of the classes. The operation `sum()`, already defined on `Collection(T)`, is repeated in some specialized collection kinds but not in some others.

Missing operations: The collection kind `Sequence(T)` is missing an operation `indexSetOf(object:T)` (could also be called (indexesOf)) as elements may appear more than once in a sequence. We propose that, for example, `Sequence{8,7,9,7}->indexSetOf(7) = Set{2,4}` can be computed. Furthermore, the operation for subtraction (`-`) is not present in the collection kind `Bag(T)`, although it is defined for `Set(T)`. There should be no problem in computing `Bag{7,7,7,8,9,9} - Bag{6,7,9,9} = Bag{7,7,8}`. Similarly, operation `union()` should be specified for `OrderedSet(T)`, with a precisely defined behavior: including at the end of `self` the elements of the operand that were not present in `self`, respecting their order.

Missing details for predefined iterator expressions: The predefined iterator expressions lack concrete details about the parameter type, return type and the repeatability of the iterator variable. For example, the following details of the standard should be clarified: (*a*) it does not become clear when the body expression has to be a Boolean expression (e.g., for `select`) and when it may be a general expression for computation (e.g., for `collect`); (*b*) it hardly becomes clear if the return type of the `closure()` operation is `Set(T)` or `OrderedSet(T)`, and when one or the other are chosen; (*c*) it does not easily become clear that for the `exists()` operation there may be more than one iterator variable, however for `select()` it is explicitly said that there must be exactly one.

Figure 3 displays the central idea of our proposal to improve the OCL standard: In the top all four new collection kinds are show with Generalization arrows partly upwards,

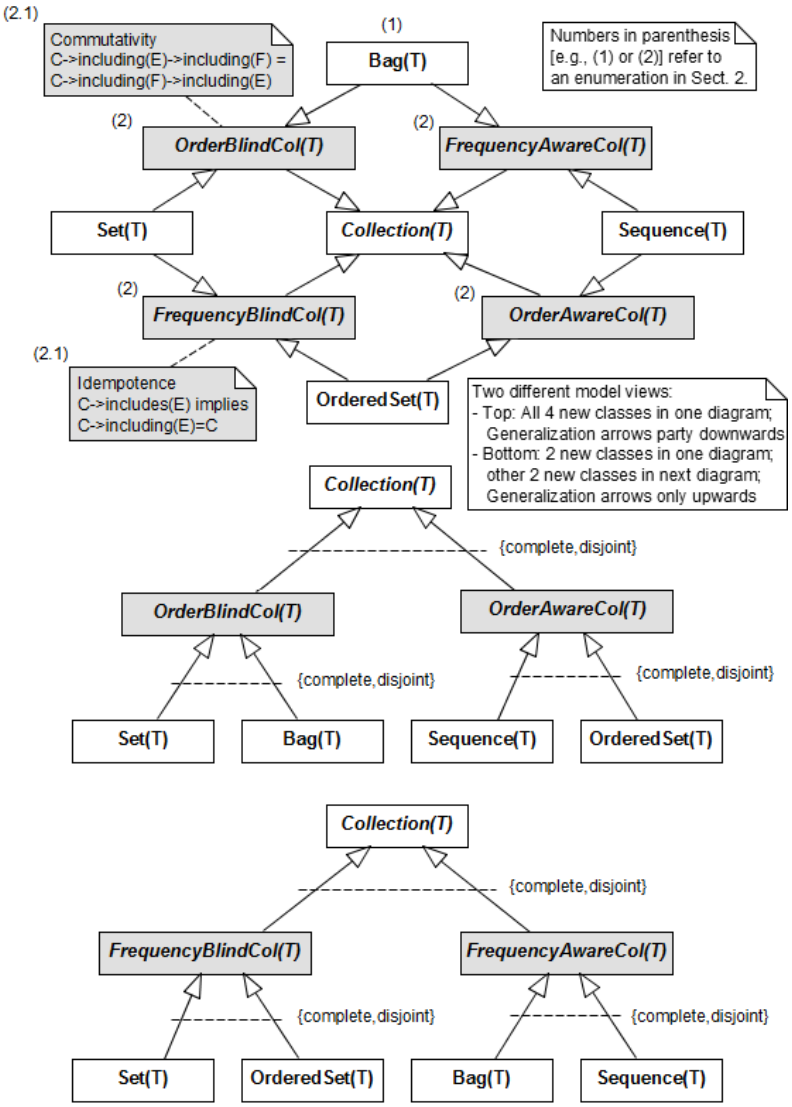

Figure 3: Two different views on new Abstract Collection Kinds for OCL.

partly downwards; in the bottom a different view on the same model is shown where Generalization arrows only point upwards. The improvements and changes are displayed in this and the next figure with a light gray background. Four new abstract collection kinds are introduced. The collection kind pair (`OrderBlindCol(T)`, `OrderAwareCol(T)`) and the pair (`FrequencyBlindCol(T)`, `FrequencyAwareCol(T)`) each builds a complete and disjoint specialization of `Collection(T)`. The first pair classifies the collection kinds according to the criterion Element-Order, the second pair conducts a classification according to the criterion Element-Frequency. Two algebraic laws for the constructor operation `including()` characterize the classification according to Element-Order and Element-Frequency: In an Order-Blind collection the operation `including()` is commutative, and

in a Frequency-Blind collection the operation `including()` is idempotent.

Figure 4 shows how the new collection kinds and our other points of criticism can be used to obtain a better definition of the OCL collection kinds and their operations. We identify in our proposal five main points of improvement.

(1) Visual model for OCL collections: The current OCL standard does not present an abstract, visual model for collections and their operations in terms of a class diagram. The purpose of UML is also to visually 'document' existing systems in an abstract way [8]. It seems rather surprising that an important component of OCL, the collections, is only described in textual form. A visual model has the power to give a precise, abstract overview, in particular for finding out what is possible in one class in comparison to another class. This is missing in the current standard document where the reader has to jump between detailed textual descriptions.

(2) Intermediate abstract collection kinds: As mentioned above, we propose four new abstract collection kinds: `OrderBlindCol(T)`, `OrderAwareCol(T)`, `FrequencyBlindCol(T)`, and `FrequencyAwareCol(T)`. The pair `OrderBlindCol(T)` and `OrderAwareCol(T)` builds a complete and disjoint specialization of `Collection(T)`. The same is true for the pair `FrequencyBlindCol(T)` and `FrequencyAwareCol(T)`. The first pair classifies a collection on whether the element insertion order is relevant, and the second pair does the classification on whether the element insertion frequency is relevant. Insertion through the operation `including()` is crucial because insertion is the constructor operation that builds up collections. Every collection can be built by starting with an empty collection and successively inserting elements. The four new collection kinds all together would be classified as `{overlapping, complete}`, but this is not shown in the figures. The two criteria order and frequency can be combined in an orthogonal way and result in the four known concrete OCL collection kinds. The two criteria have already been proposed as invariants in [9] with a slightly less concise criterion for order frequency.

Instead of using the names `FrequencyBlindCol(T)` and `FrequencyAwareCol(T)` one could call the collection kinds `CountBlindCol(T)` and `CountAwareCol(T)`. This proposal from Ed Willink would be even a bit shorter. In this contribution however, we stick to the names stated before.

(2.1) Characterization by OCL laws: The four newly introduced collection kinds can be kept apart by two OCL laws (axioms) expressing properties about how the basic constructor operation `including` works. For a collection kind, each law either holds in general or does not hold in general. The two laws and their validity are:

```
Collection(T).allInstances()->forAll(C | -- including() is commutative
  T.allInstances()->forAll(E,F |
    C->including(E)->including(F) = C->including(F)->including(E) ))
Collection(T).allInstances()->forAll(C | -- including() is idempotent
  T.allInstances()->forAll(E |
    C->includes(E) implies C->including(E) = C ))
```

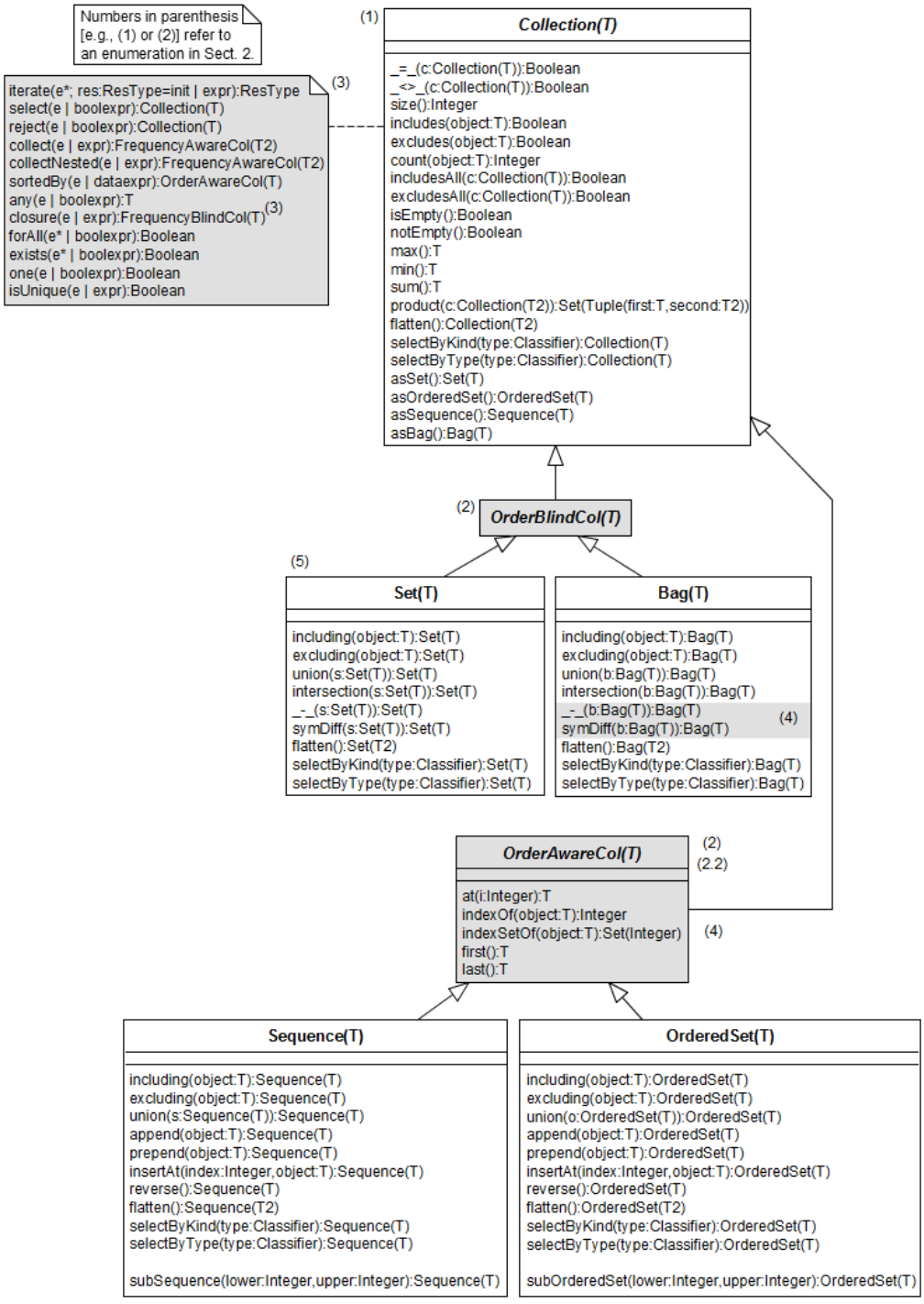

Figure 4: Refactoring Collection Kinds and Operations for OCL.

including	commutative	idempotent
Set(T)	yes	yes
Bag(T)	yes	no
Sequence(T)	no	no
OrderedSet(T)	no	yes

The current OCL standard states many details about the commonalities and differences between the four collection kinds. However, it does not discuss the above important and fundamental principles. We believe these fundamental principles of the four collection kinds are formulated concisely in these seven lines of OCL, and lead to the four abstract, intermediate collection kinds `OrderBlindCol(T)`, `OrderAwareCol(T)`, `FrequencyBlindCol(T)`, and `FrequencyAwareCol(T)` that we have introduced.

including	commutative
OrderBlindCol(T)	yes
OrderAwareCol(T)	no

including	idempotent
FrequencyBlindCol(T)	yes
FrequencyAwareCol(T)	no

Minor remarks: (*a*) The above formulas are not valid OCL expressions because the current OCL does not allow to use type parameters like the above `T`; (*b*) The previous OCL expressions use formulas with equality as `L = R`, but the order of the terms `L` and `R` does not matter; this means one could equivalently state the formulas using `R = L`, e.g., one could say `C = C->including(E)`.

(2.2) Operations lifted to abstract collection kinds: There are a few operations that are in the current OCL standard repeated in the collection kinds `OrderedSet(T)` and `Sequence(T)`, namely the operations `at()`, `indexOf()`, `first()`, and `last()`. These operations can be lifted to the new abstract collection kind `OrderAwareCol(T)`, and then they do not need to be repeated in the concrete collection kinds.

(3) Classification for iterator parameters and type: The collection iterators like `collect()` or `exists()` belong to the most important language elements to deal with OCL collections. Their description in the standard is rather distributed over the single iterators, and some distinctions between the iterators are even missing in the standard (or at least the distinctions must be deduced from other facts and are not explicitly stated), e.g., the difference between `collect()` or `exists()` in that `collect()` allows an arbitrary OCL term in its body, whereas `exists()` requires a Boolean typed body, is not explicitly apparent in the standard. Also, the return types of some iterators do not become crystal clear in the standard, e.g., the fact that `closure()` returns a set or an ordered set. We propose to explicitly denote in the class diagram for all iterators its parameter types and its return types. We also

have indicated in the class diagram whether an iterator variable `e` is allowed to occur only once or multiple different iterator variables are allowed: the notations `e` and `e*` present these distinctions.

(4) Addition of missing operations: We have added some operations that we think that are useful, but missing in the current OCL. These are operations for (*a*) determining the index set of elements in an order-aware collection, i.e., `indexSetOf()` (could also be called `indexesOf`), (*b*) calculating the difference and symmetric difference on bags, and (*c*) calculating the union of `OrderedSet` collections.

(5) Systematic order of operations: Last but not least, we re-arranged the order in which the operations occur in the collection kinds as the order in the current standard is inconsistent. So, for example, the operation `including()` is always the first operation in a collection kind, or the operations `subSequence()` and `subOrderedSet()` are presented as the last operations in the collection kinds `Sequence(T)` and `OrderedSet(T)`.

Summarizing, we would like to emphasize that OCL proposes the right four concrete collection kinds, but currently OCL does not give the right arguments and explanations why exactly these four concrete collection kinds are used. We believe that a view on collection kinds guided by the properties of the constructor operation `including()` and the introduction of the four intermediate abstract collection kinds leads to the four concrete collection kinds in a natural way.

As a result, the proposed solution in Fig. 4 provides a clear and regular refactoring of the current OCL 2 operations on collections, which avoids duplications and ensures completeness. In particular, duplications are avoided because operations are lifted to the highest class of the hierarchy where they belong. Some of the operations need to be redefined in the subclasses, particularly when the operations restrict the return type of the operation defined for the superclass.[1] Completeness is achieved by making sure no operation definition is mistakenly missing in any of the subclasses. For example, the operations difference (`-`) and `symmetricDifference()` are currently missing in OCL for class `Bag(T)`.

3. Related Work

International standards are the results of a committee consensus and therefore are not perfect documents. As such, OCL specifications are not free from subtle issues and small gaps. A very interesting paper [7] describes the history and a discussion on (some of) the main issues of the current OCL 2 specification. However, it does not explicitly cover the issues raised in this work. Another interesting paper [12] also identifies some of the issues found in OCL while producing a model for the OCL standard library so as to develop Eclipse OCL. Some of the problems with OCL collections mentioned here were

[1] Note that this *covariant overriding*, as defined in UML [8, 10], also ensures Liskov's substitutability principle [11].

also identified in that paper, and partially resolved in Eclipse OCL — although some of them still remain, see the discussion about Eclipse OCL below.

As mentioned in the introduction, one of the problems of international standards that contain incorrect aspects or gaps in certain areas, is that developers of tools that implement the standard provide separate and usually incompatible implementations for these issues. We have analyzed how three of the major implementations of OCL, namely Acceleo [3], USE [4] and Eclipse OCL [5], deal with collection operations. We excluded the Eclipse Epsilon Language (EOL) [13] because it departs from the standard OCL and does not claim fully conformance to it; e.g., it includes a wider variety of operations than the OCL standard library. Anyway, its organization of the Collections type hierarchy is similar to the one proposed here, it just doesn't have the explicit notion of `OrderAwareCol`.

First of all, we must say that all three implementations fully respect the OCL standard when it is precise and the specifications are clear. It is only in those parts that we have identified as potential issues where the different implementations diverge. Let us describe the main differences below, according to three main dimensions: where the operations are defined, i.e., in which classes; the operations that are missing; and those that were not in the OCL standard but have been introduced by the implementation.

Acceleo

- **How and where operations are defined**: The Acceleo implementation respects the hierarchy of classes defined in the OCL standard, although it refactors the definition of some operations, lifting them to the top-most class `Collection(T)`. This permits avoiding both unnecessary re-definitions in the subclasses and problems due to forgetting some operation definitions. Thus, operations `including()`, `excluding()`, `count()`, `flatten()`, `sum()`, and all conversion operations `asSet()`, `asBag()`, `asOrderedSet()`, and `asSequence()` are defined only in class `Collection(T)` but not in the subclasses. Contrarily, operations `=` and `<>` are not defined in the top-most class `Collection(T)`, but only in the subclasses.
- **Missing operations in Acceleo**: Acceleo does not implement some of the collections standard operations, such as `min()` and `max()`, neither the iterators `selectByKind()` and `selectByType()`. Operation `reverse()` for `OrderedSet(T)` and `Sequence(T)` is not defined either.
- **Introduced operations in Acceleo**: Operation `-`, only available for `Set(T)` in the OCL standard, is introduced for `OrderedSet(T)` too. Furthermore, Acceleo introduces two new operations, `=` and `<>`, for comparing an `OrderedSet(T)` with a `Set(T)`, in addition to that that compare them with `OrderedSet(T)`. Finally, operation `union()` is added to `OrderedSet(T)` with two flavors, depending on whether you want to add a `Bag` or a `Set` (but, curiously, not an `OrderedSet`).

USE

- **How and where operations are defined**: USE respects the hierarchy of classes defined in the OCL standard.

- **Missing operations**: USE fully supports all operations defined in the OCL standard.
- **Introduced operations**: In USE, operation `union()` is added to `OrderedSet(T)` to allow addition with other `OrderedSet(T)` collections.

Eclipse OCL

- **How and where operations are defined**: Eclipse OCL introduces two new intermediate abstract classes to gather common properties from the corresponding subclasses: `OrderedCollection(T)` and `UniqueCollection(T)`. They both inherit from `Collection(T)`. The former represents collections in which order matters, and defines operations `at()`, `first()`, `indexOf()`, and `last()`. The latter represents collections with no duplicated elements, and defines operations `-`, `intersection()`, `symmetricDifference()`, and `union()`. Besides, it defines iterator `sortedBy()`. They are similar to our proposed `OrderAwareCol(T)` and `FrequencyAwareCol(T)` abstract classes, respectively. Some operations are also lifted to Class `Collection(T)`, namely `union()` and `intersection()`, which are also defined differently: they have two versions depending if the parameter is a `UniqueCollection(T)` or a `Collection(T)`, returning different types (`Sets` or `Bags`). In this sense, we believe that a richer set of subclasses that discriminates between order-aware and order-blind collections such as ours would allow more precise specifications.
- **Introduced operations**: Eclipse OCL introduces two new operations in `Collection(T)` and in the four concrete subclasses: `includingAll()` and `excludingAll()`. Furthermore, operations `appendAll()` and `prependAll()` are added to both classes `Sequence(T)` and `OrderedSet(T)`. In addition, the fact that several operations become lifted either to the intermediate classes or to the top-most class `Collection(T)` makes them available for the corresponding subclasses. This permits solving some of the gaps that we have identified in the OCL standard in operations defined for `Bag(T)` and `OrdereSet(T)`, for instance.

We have also compared the OCL collections structure with that of collections in renowned programming languages. A summarized graphical view of the collections in Java 8 [6] is shown in Fig. 5. Note that, for readability purposes, we have refactored some of the associations, but the classes and interfaces faithfully represent the Java implementation. Java interfaces define the supported types, and different classes provide separate implementations depending on the internal structures used to store the collection elements. The role of OCL `Sequence` is played by Java interface `List`, which offers just specialized sets of operations; `Queue` and `Dequeue` are implementations of `OrderAwareCol` with some specific access operations; `Set` corresponds to the Java `Set`. Note that there are no Java data structures corresponding to OCL `OrderedSet` and `Bag` – Java `SortedSet` is a `Set` with a total order operation defined for the elements, which is different from the OCL `OrderedSet` where what matters is the partial order between the elements in the collection, i.e., their indexes [9]. The Java collection `LinkedHashSet` is the most similar to the OCL `OrderedSet`, although it does not support index-based access. Whilst

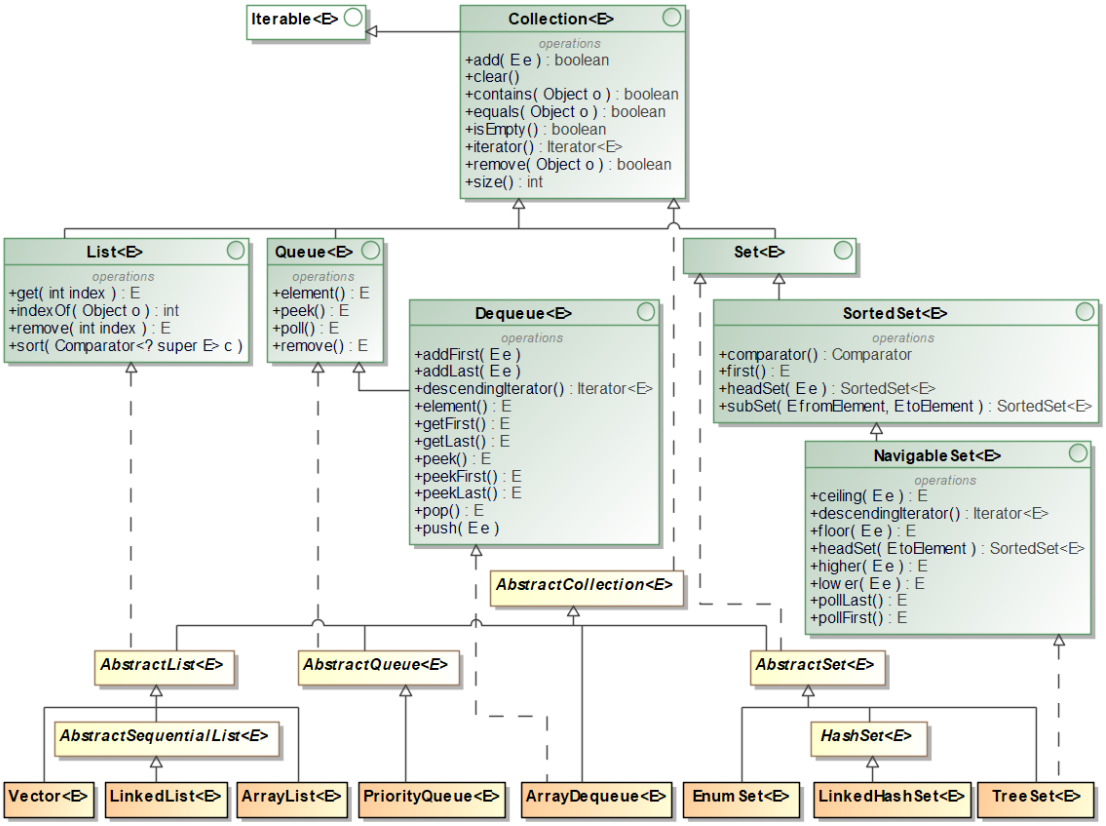

Figure 5: Java 8 Collections (from [6]).

Java does not explicitly have operations by the names of `union()`, `intersection()` and `difference()`, it does have closely related operations: the methods `addAll()`, `retainAll()` and `removeAll()` are the methods most commonly used to implement union, intersection and difference in Java. It is also worth noting that the library `java.util.stream.Stream`[2] can be used to transform any collection to a `Stream`, which is a "sequence of elements that supports sequential and parallel aggregate operations". Operations such as `distinct()` and `sorted()` can be applied on streams. Nevertheless, the OCL collection operations are not available in Java Streams either. These differences between the OCL and Java collections are rather natural because of the main focus of each language: OCL is chiefly a modeling language, whilst Java is more focused on the implementation aspects.

4. Conclusions

In this paper we have identified some of the limitations and problematic issues that the current OCL 2 standard presents with regard to collections, and proposed some

[2] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

recommendations about how they can be successfully addressed.

Our proposal has focused mainly on the structure and organization of the operations to manage collections in OCL, and on the signature of these operations. As part of our future work, we also plan to address the behavioral specification of the operations, which we also think needs to be significantly improved in the OCL 2 standard.

More precisely, the OCL 2 specification provides the semantics of some of the operations defined for collections in terms of pre- and post-conditions. This also applies to most of the operations defined for the abstract class `Collection(T)`, from which the rest of the concrete classes inherit. One of the problems is that some of the operations' specifications do not seem to consider the individual characteristics of the subclasses, and may break Liskov's substitutability principle [11]. Now that UML 2.5 has a clearer inheritance and operation overriding mechanisms [10], we believe that the precise and rigorous specification of the semantics of OCL Collection operations is possible, using behavioral subtyping [11, 14]. This would complement the present work and be useful for UML and OCL tool builders when it comes to specify the behavior of the Collection operations in an interoperable and standard way.

Acknowledgments

We would like to thank the reviewers for their insightful and very useful comments, which have helped us to improve previous versions of this paper. The feedback from Ed Willink was extremely helpful. This work has been partially funded by Research Projects PGC2018-094905-B-I00, JA–P20_00067 and TIN2016-75944-R.

References

[1] N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

[2] Object Management Group, Object Constraint Language (OCL) Specification. Version 2.4, 2014. OMG Document formal/2014-02-03.

[3] Obeo, Acceleo/OCL Operations Reference, Last accessed June 2021. URL: https://wiki.eclipse.org/Acceleo/OCL_Operations_Reference.

[4] M. Gogolla, F. Büttner, M. Richters, USE: A UML-based specification environment for validating UML and OCL, Sci. Comput. Program. 69 (2007) 27–34. doi:10.1016/j.scico.2007.01.013.

[5] Eclipse, Eclipse OCL (Object Constraint Language), Last accessed June 2021. URL: https://projects.eclipse.org/projects/modeling.mdt.ocl.

[6] Oracle, Java Platform Standard Ed. 8, Last accessed June 2021. URL: https://docs.oracle.com/javase/8/docs/api/java/util/package-tree.html.

[7] E. Willink, Reflections on OCL 2, J. Object Technol. 19 (2020) 3:1–16. doi:10.5381/jot.2020.19.3.a17.

[8] Object Management Group, Unified Modeling Language (UML) Specification. Version 2.5, 2015. OMG document formal/2015-03-01.

[9] F. Büttner, M. Gogolla, L. Hamann, M. Kuhlmann, A. Lindow, On better understanding OCL collections *or* an OCL ordered set is not an OCL set, in: Proc. of OCL@UML'09, volume 6002 of *LNCS*, Springer, 2009, pp. 276–290. doi:10.1007/978-3-642-12261-3_26.

[10] F. Büttner, M. Gogolla, On generalization and overriding in UML 2.0, in: Proc. of OCL@UML'04, 2004, pp. 1–15.

[11] B. H. Liskov, J. M. Wing, A behavioral notion of subtyping, ACM Trans. Program. Lang. Syst. 16 (1994) 1811–1841. doi:10.1145/197320.197383.

[12] E. D. Willink, Modeling the OCL standard library, Electron. Commun. Eur. Assoc. Softw. Sci. Technol. 44 (2011). doi:10.14279/tuj.eceasst.44.663.

[13] E. Epsilon, Eclipse EOL (Epsilon Object Language), Last accessed June 2021. URL: https://www.eclipse.org/epsilon/doc/eol/#collections-and-maps.

[14] P. America, A behavioural approach to subtyping in object-oriented programming languages, in: M. Lenzerini, D. Nardi, M. Simi (Eds.), Inheritance Hierarchies in Knowledge Representation and Programming Languages, John Wiley and Sons, 1991, pp. 173–190.