On Better Understanding OCL Collections
or
An OCL Ordered Set is not an OCL Set

Fabian Büttner, Martin Gogolla, Lars Hamann, Mirco Kuhlmann, Arne Lindow

University of Bremen
Computer Science Department
Database Systems Group
D-28334 Bremen, Germany

Abstract. Modeling languages like UML or EMF support textual constraints written in OCL. OCL allows the developer to use various collection kinds for objects and values. OCL 1.4 knows sequences, sets, and bags, while OCL 2.0 adds ordered sets. We argue that this addition in the OCL standard was not carried out in a careful way and worsened conceptional problems that were already present previously. We discuss a new way of establishing the connection between the various collection kinds on the basis of explicitly highlighting and characterizing fundamental collection properties.

1 Introduction

During the last years, the Object Constraint Language (OCL) [OMG06,WK03] became a central ingredient in modeling and transformation languages like UML [OMG04] or EMF [BSM$^+$03]. Thus, approaches for model-driven engineering rely on it. OCL is supported by many commercial and open source tools from the OMG and Eclipse sphere. Thus an unambiguous and understandable definition of language concepts is indispensable.

OCL has been developed over the years and is described in a handful of OMG standards with different degrees of amount of care devoted to particular details. Furthermore, we would call the style of definitions used in the OCL standards mostly "operation oriented" with emphasis on pre- and postconditions which is in contrast to a style which could be coined as "property oriented" with emphasis on invariants. In particular, the OCL collections are nearly completely defined in formal terms by characterizing the behavior of single operations. However, class-scope properties of collections relating different operations are stated, if at all, only in an informal manner. This paper concentrates on formal class-scope properties of OCL collections and puts forward a new way of establishing the connection between the various collection kinds on the basis of explicitly highlighting and characterizing fundamental collection properties with invariants. These properties which are expressed as equations could be used, for example, in OCL tools for pointing the developer to semantically equivalent OCL expressions. Similar collection kinds as present in OCL have been studied in connection with complexity [TS91], functional [HB94] and logic programming [Won95] as well as in the context of conceptual database models [HL07].

The rest of the paper is structured as follows. Section 2 points to various deficiencies in the OCL standard collection definitions. Sections 3 analyses the properties of the four OCL collection kinds bag, set, sequence, and ordered set. Section 4 looks at OCL collections from the point of view of algebraic specifications. Section 5 discusses a new type hierarchy for collections including invariants which characterize central properties. The paper ends with concluding remarks.

2 Deficiencies in OCL Collection Definitions

Let us first explain difficulties which arise with the latest OCL standard. Basic properties of ordered sets are unclear when one considers only the statements made in the OCL 2.0 standard [OMG06]. For example, the question how many ordered sets exist which contain three distinct elements (x<y<z) over a type with a total order cannot be answered considering only OCL 2.0 statements. It is unclear whether there are two (x,y,z and z,y,x) or six such ordered sets (all six permutations). Closely related to this is the fact that the relationship between the concept "ordered set" and a set which has a total order on its elements and which thus could be made into a "sorted set" is not discussed. Viewed at the OCL 2.0 standard in more detail, the following issues come up.

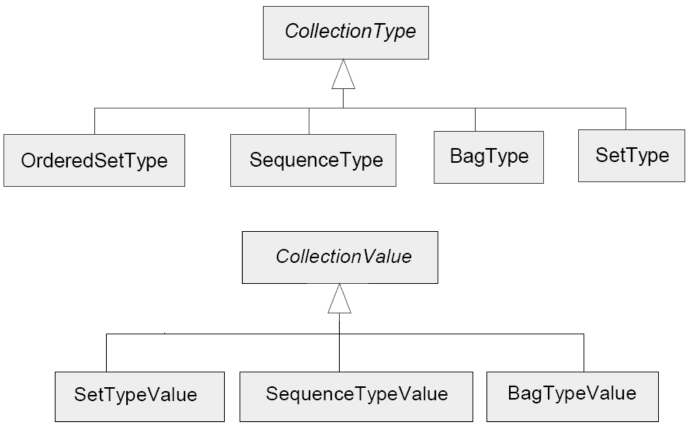

Fig. 1. Type Hierarchy and Incomplete Value Hierarchy for OCL 2.0 Collections

- The operations equality `=`, `including`, `excluding`, and the collection conversions `asBag`, `asSet`, and `asSequence` are not defined on ordered sets.
- Some operation definitions on ordered sets are incorrect. For example, the description of `append` with postconditions is erroneous, because it does

not handle the most interesting case when the parameter object which is to be appended is already present in the collection. The description of `OrderedSet::append` is textually identical to (and probably has, without any modification, been copied from) `Sequence::append`.
- As shown in Fig. 1 which is condensed from [OMG06, pages 34 and 98], ordered sets are not considered in the collection value hierarchy, although are they present in the type hierarchy.
- Ordered sets are not mentioned when collection operation like `select`, `exists` or `collect` are defined on the basis of `iterate`. Thus these operations are formally not defined on ordered sets.

Furthermore, there were already some deficiencies in the definition of OCL 1.4 collections [OMG03].

- OCL 1.4 collections are not explicitly characterized by general properties, for example by invariants, but are defined by the behavior of single operations like equality `=` or the constructor-like operations `including`. The interplay between the operations is not discussed.
- A conceptual question which is not answered in the OCL 1.4 standard is the question why a set cannot be regarded automatically as a bag whose elements occur exactly once. A similar question comes up in connection with the relationship between ordered sets and sequences.
- Furthermore the properties of conversion operations are only handled in a sketchy way: On the one hand there are explanations concerning the question which conversion operations are completely determinated by their arguments and which conversion operations are only incompletely determinated by their arguments and must therefore take implementation dependent choices; but on the other hand not all relevant important properties of these conversion operations are explicitly discussed in the OCL 1.4 standard.

3 Collection and Conversion Properties

3.1 Collection Properties

As a basis for discussion, we propose in the following two fundamental properties which an OCL collection kind might satisfy or might not satisfy. Both properties are formulated in an OCL-like style as invariants. However, as will be explained further down, both formulations are not valid current OCL expressions.

- Property `insertionOrderIndependence`: The insertion order of elements into the collection does not matter.

```
context Collection(T) inv insertionOrderIndependence:
  T.allInstances()->forAll(x,y:T|
    self->including(x)->including(y) =
    self->including(y)->including(x))
```

– Property singleElementOccurrence: An element can occur at most once in the collection or, in other words, duplicates are not allowed.

```
context Collection(T) inv singleElementOccurrence:
  self->forAll(x:T|self->count(x)=1)
```

In Fig. 2 we identify the four OCL collection kinds, the above two properties, and twelve arrows which represent the twelve collection conversions operations. For example, Bag(T)::asSequence():Sequence(T) is represented by the topmost dashed arrow. Each conversion operation is classified into one of three categories: (1) completely determinated and injective function, (2) incompletely determinated function, i.e., OCL engine implementation-dependent function, and (3) completely determinated and non-injective function.

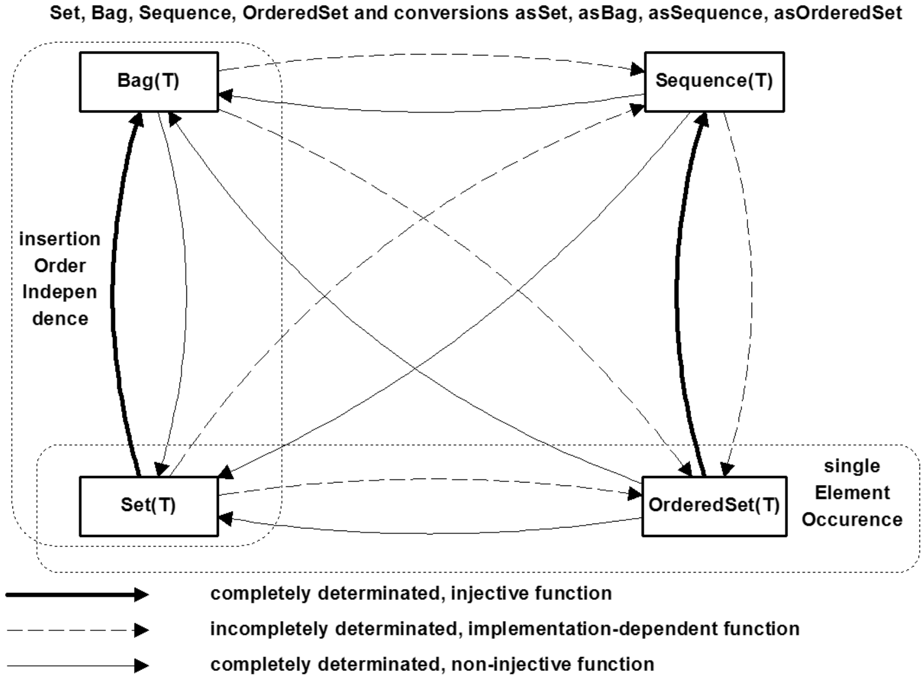

Fig. 2. Classified Conversions between OCL Collections

Let us now discuss the relationship between the two properties and the OCL collections kinds. The two basic properties insertionOrderIndependence and singleElementOccurrence are chosen in such a way that they orthogonally partition the OCL collection kinds: insertionOrderIndependence is valid on bags and sets, whereas it is invalid on sequences and ordered sets; single-ElementOccurrence is valid on sets and ordered sets, whereas it is invalid on bags and sequences. For sets both properties have to hold, for bags and ordered sets exactly one respective property is valid, and sequences do not have to obey any of these two properties.

These two central properties may be regarded as the formalization of the informal description given in the OCL standard [OMG06, page 145]:

> Set: The Set is the mathematical set. It contains elements without duplicates ... OrderedSet: The OrderedSet is a Set, the elements of which are ordered. It contains no duplicates ... Bag: A bag is a collection with duplicates allowed. That is, one object can be an element of a bag many times. There is no ordering defined on the elements in a bag ... Sequence: A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once.

Note that the wording used for the description of ordered sets "The OrderedSet is a Set" may suggest that ordered sets are subsets of sets. Our argumentation below does *not* follow this view.

The above formulations are not valid current OCL expressions because (1) a type parameter `T` is used for the context class and (2) `allInstances` is applied to the type parameter `T` which is generally not allowed, if `T` is a basic data type or `T` is again a collection type. A forbidden example situation for the first case is `Collection(Integer)`, and a forbidden example for the second case is `Collection(Sequence(Person))`.

Please note that Fig. 2 is incomplete because a conversion on each collection kind to itself is missing. According to the OCL standard, for example, `asBag()` can be applied on `Bag(T)`. Analogously, the other three collection kinds additionally possess a loop arrow from the respective type to the type itself. We have not shown these conversions in order to keep the figure simple.

Finally, we state some observations which apply only to ordered sets.

- `OrderedSet(T)` is the only collection kind where all three conversions from the other collection kinds are implementation-dependent. Therefore, a conversion from any other collection to an ordered set must make an implementation-dependent choice.
- An ordered set (in mathematics or theoretical computer science) is usually different from the OCL ordered set. There, an ordered set denotes a set together with a partial order on the set, i.e., a reflexive, antisymmetric and transitive relationship. The OCL ordered sets do *not* rely on such a relationship.

3.2 Completely Determinated, Injective Conversions

In Fig. 2 we have shown the only two injective conversions with thick arrows: `Set(T)::asBag():Bag(T)` and `OrderedSet(T)::asSequence():Sequence(T)`. The property of being injective comes from the fact that (1) a set can be interpreted in a unique way as a bag where each element occurs once and (2) an ordered set can be interpreted in a unique way as a sequence where the order of the elements in both collections coincides.

This view on OrderedSet(T)::asSequence():Sequence(T) contributes to the answer of the question raised earlier: How many ordered sets exist which exactly contain three different elements, for example, the integers 7,8,9? All six permutations can be considered as ordered sets without problems. Thus OrderedSet{8,7,9} is a valid ordered set although it is not a sorted set.

3.3 Incompletely Determinated Conversions

In Fig. 3 the five incompletely determinated conversions are pictured with dashed arrows. By using the notion "incompletely determinated" we refer to the fact that these conversions are not determined in a canonical way and that in these

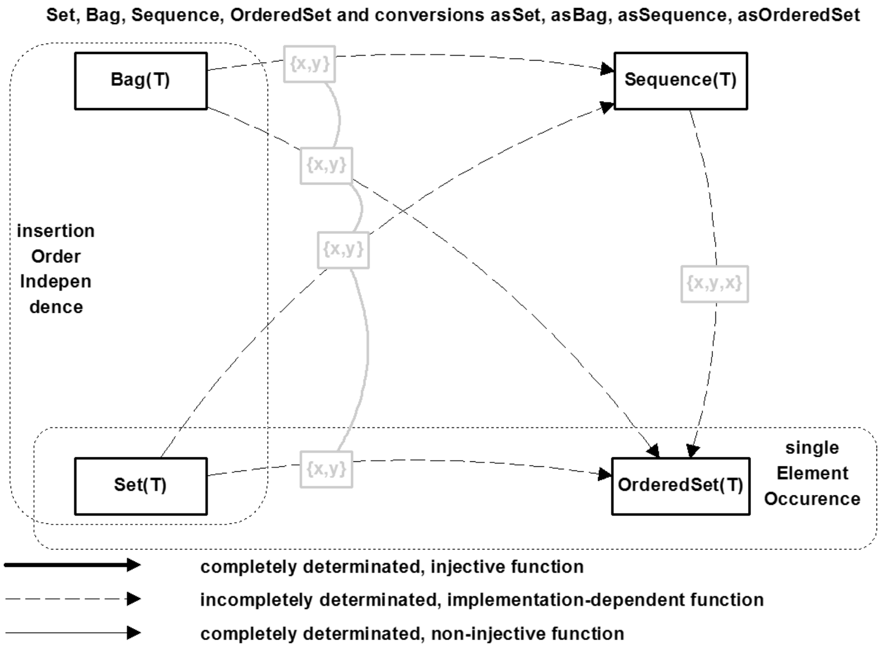

Fig. 3. Incompletely Determinated Conversions between OCL Collections

conversions implementation-dependent choices have to be made. An example argument for this indeterminateness is shown in the figure for each conversion in grey boxes. For example, Bag(T)::asSequence():Sequence(T) is incompletely determinated, because the bag Bag{x,y} can be mapped to Sequence{x,y} or Sequence{y,x}; Sequence(T)::asOrderedSet():OrderedSet(T) is incompletely determinated, because the sequence Sequence{x,y,x} can be mapped to OrderedSet{x,y} or OrderedSet{y,x}. These conversions are not completely determinated, because one order has to be fixed which is not uniquely present in the argument collection.

3.4 Completely Determinated, Non-Injective Conversions

In Fig. 4 the five completely determinated conversions which are not injective are pictured with solid arrows. One example argument for being not injective is indicated in the figure in grey boxes: For example, `Bag(T)::asSet():Set(T)` is

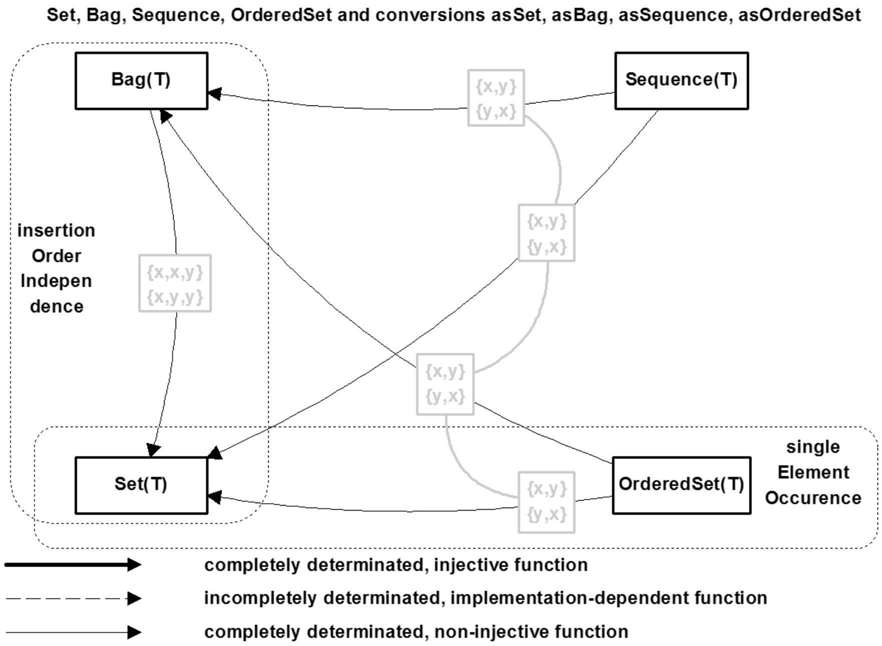

Fig. 4. Completely Determinated Conversions between OCL Collections

not injective, because the bags `Bag`{`x,x,y`} and `Bag`{`x,y,y`} are both mapped to `Set`{`x,y`}, and the conversion is completely determinated because when going from the bag to the set one simple ignores the multiple occurrences of elements. Analogous arguments for being non-injective and for determinateness hold for the other conversions.

Please note that the incompletely determinated, dashed conversions in Fig. 3 basically go from the left to the right, and the completely determinated, solid conversions in Fig. 4 go from the right to the left. This is due to the fact that the left collection kinds `Bag(T)` and `Set(T)` obey the property `insertionOrder-Independence` while this property is in general not valid in the right collection kinds `Sequence(T)` and `OrderedSet(T)`.

4 Excursus: Collections as Algebraic Specifications

We now discuss algebraic specifications [EM85,EGL89,Wir90] for OCL collections as shown in Fig. 5. Viewing these collections from a different field helps in understanding them because the view on collections is different in that it emphasizes different aspects. In particular one can nicely express the interplay between operations. Equational algebraic specifications have the advantage of (1) showing central and characteristic properties and (2) being able to automatically construct an interpretation for the respective structure, namely the so-called quotient term algebra, which is an intial algebra.

- There is a close relationship between our two central OCL requirements and the equations: The invariant `insertionOrderIndependence` corresponds to the equation `commutativity`, and the invariant `singleElementOccurrence` has the equations (defined in Fig. 5) `absorption` and `absorptionIfPresent` as its counterparts.
- One can view the specifications for sets, bags, and sequences as a hierarchy of requirements: There are two equations for sets, one of these equations is also used for bags, and there is no equation for sequences. Thus the set is more restricted than the bag which is more restricted than the sequence. Sequences are generated freely, i.e., they are described by a pure term algebra without any factorization on terms.
- We have given specifications not for general collections with type parameters, but for more concrete collections which roughly correspond to (speaking in OCL notions) `Sequence(Integer)`, `Bag(Integer)`, `Set(Integer)` and `OrderedSet(Integer)`, respectively. The general collections like `Bag(T)` could be defined in an analogous way. In particular, because so-called parametrized algebraic specifications are a well-studied concept, and type parameters do not present difficulties (in contrast to current OCL).
- The type `orderedSet` needs an auxiliary operation `includes` and conditional equations. Otherwise it would not be specifiable. Thus the type `orderedSet` belongs to a different specification class than the three other collections which are specifiable by pure equations and without auxiliary operations. The type `orderedSet` is also the only one which needs the equality on its elements whereas the other collections do not require this.
- Interestingly, in both specification formalisms, in OCL and with algebraic specification, the ordered sets play a special role and special means must be taken to describe it.
- Finally, when we want to play around with the algebraic specifications and study further structures, we discover that we could also build a collection kind which considers the equation from the bag specification for the absorption property (`including(including(S,N),N) = including(S,N)`) as its single equation. This would construct a structure that could be called "repetition free sequences", say `RepFreeSequence`. In that structure, for example, the statements `RepFreeSequence{22, 11, 11, 22, 33, 33} = RepFreeSequence{22, 11, 22, 33}` and `RepFreeSequence{22, 11, 22,`

```
srts bool, nat ------------------------------------------------------------
opns false, true : -> bool
     or : bool bool -> bool
     zero : -> nat
     succ : nat -> nat
     eq : nat nat -> bool
eqns or(B,true) = true
     or(B,false) = B
     eq(zero,zero) = true
     eq(zero,succ(N)) = false
     eq(succ(N),zero) = false
     eq(succ(N),succ(M)) = eq(N,M)

srts sequence -------------------------------------------------------------
opns emptySequence : -> sequence
     including : sequence nat -> sequence
eqns                                                       -- no equations

srts bag ------------------------------------------------------------------
opns emptyBag : -> bag
     including : bag nat -> bag
eqns including(including(B,N),M) =
       including(including(B,M),N)                         -- commutativity

srts set ------------------------------------------------------------------
opns emptySet : -> set
     including : set nat -> set
eqns including(including(S,N),M) =
       including(including(S,M),N)                         -- commutativity
     including(including(S,N),N) =
       including(S,N)                                         -- absorption

srts orderedSet ---------------------------------------------------------
opns emptyOrderedSet : -> orderedSet
     including : orderedSet nat -> orderedSet
     includes : orderedSet nat -> bool
eqns includes(emptyOrderedSet,N) = false
     includes(including(O,N),M) = or(eq(N,M),includes(O,M))
     includes(O,N)=true =>
       including(O,N) = O                          -- absorptionIfPresent
```

Fig. 5. Algebraic Specification of Collections of Natural Numbers

33} <> RepFreeSequence{22, 11, 33}` would be true. Such a collection kind is not (and from our point of view should not be) present in OCL.
- For non-specialists in algebraic specifications we emphasize that we have used above *equations* and not rewrite rules: Equations are applicable in both directions (from left to right and from right to left) whereas rewrite rules usually are applied in one direction only. For example, the semantics of specification remains the same if we write `eq(N,M) = eq(succ(N),succ(M))` instead of the equation stated above.

5 Alternative Collection Type Hierarchy and Invariants

5.1 Alternative Collection Type Hierarchy (Discussion Proposal)

Figure 6 shows our discussion proposal for a possible type hierarchy of OCL collections. Because there are already two conversion operations present in current OCL, which are also canonical, natural injections, it may seem natural to utilize the UML generalization concept in order to express this relationship. Thus `Set(T)` becomes a subtype of `Bag(T)` and `OrderedSet(T)` becomes a subtype of `Sequence(T)`. Please note, although the wording might suggest something different, `OrderedSet(T)` is not a subtype of `Set(T)`.

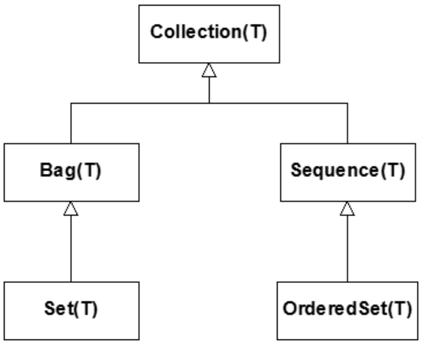

Fig. 6. Alternative OCL Collection Type Hierarchy (Discussion Proposal)

Viewing at the proposal so far, we can observe that having "natural injections" sounds like a nice criterion, but this is not the only criterion for establishing a generalization hierarchy. Another important aspect is the inheritance of properties (e.g., invariants) from a supertype to a subtype. In our case, the upper collection kinds `Bag(T)` and `Sequence(T)` fullfill, for example, the property `COLLECTION->including(x)->size() = COLLECTION->size()+1`. But this property should not be valid in the lower collection kinds. Therefore it seems advisable for us, to keep the standard collection hierarchy where the specialized collection kinds `Bag(T)`, `Set(T)`, `Sequence(T)` and `OrderedSet(T)` are subtypes of `Collection(T)`.

5.2 Collection Invariants

In any case, we can extend the collection type hierarchy with class invariants characterizing the different collections. We can connect our central requirements for collections with the respective collection kinds.

```
context Bag(T) inv insertionOrderIndependence:
  T.allInstances()->forAll(x,y:T|
    self->including(x)->including(y)=self->including(y)->including(x))

context Set(T) inv insertionOrderIndependence:
  T.allInstances()->forAll(x,y:T|
    self->including(x)->including(y)=self->including(y)->including(x))

context Set(T) inv singleElementOccurrence:
  self->forAll(x:T|self->count(x)=1)

context OrderedSet(T) inv singleElementOccurrence:
  self->forAll(x:T|self->count(x)=1)
```

After having fixed the central properties of collections we can now explain our view on the difference between sets and ordered sets. Sets must obey the requirement `insertionOrderIndependence`, while ordered sets should not follow this constraint. The usual view on generalization is that any property of a more general class is inherited to a more special class. Thus if one would make `OrderedSet(T)` a subclass of `Set(T)`, ordered sets have to be "insertion order independent" which would be a contradiction to their desired property of not obeying that constraint. In our view, a subclass should not take away a property which is present in a more general class. Furthermore, a subclass element should correspond in a unique way to a superclass element, which would not be the case here, because `OrderedSet(T)::asSet():Set(T)` is not injective. Our statement "An OCL Ordered Set is not an OCL Set" should be understood in this context. As an alternative to the notion "Ordered Set" the unhandy notion "Single Occurrence Sequence" (for which an abbreviation like "Sos" or "Soq" could be used) may be more appropriate.

The handling of the type parameter `T` remains to be studied for future work. However, in order to avoid expressions like `T.allInstances()` one could (at least in the above example) use the context variable `self`. The invariant would then be read as indicated below. This opens the possibility of giving at least a preprocessor semantics (textual replacement semantics) to type parameters where each collection type occurrence must be accompanied by respective constraints where the type parameter is replaced by an actual value.

```
context Bag(T) inv insertionOrderIndependence_allInstancesReplaced:
  self->forAll(x,y:T|
    self->including(x)->including(y)=self->including(y)->including(x))
```

Employing the defined type hierarchy it is also possible to state basic invariants about fundamental collection operations like `including`, `excluding`, `includes` and `excludes`. In particular, such invariants can make statements about the relationship between these operations.

```
context Collection(T) inv includingImpliesIncludes:
  T.allInstances()->forAll(x:T|
    self->including(x)->includes(x))

context Collection(T) inv excludingImpliesExcludes:
  T.allInstances()->forAll(x:T|
    self->excluding(x)->excludes(x))

context Collection(T) inv includesXorExcludes:
  T.allInstances()->forAll(x:T|
    self->includes(x) xor self->excludes(x))
```

Another interesting area is to focus only on the connection between `including` and `excluding`. Interestingly, an invariant which directly requires something like `COLLECTION->including(x)->excluding(x) = COLLECTION` does neither hold for bags, sets, sequences nor ordered sets. However, a variant realizing the underlying idea is valid in all respective collections. This invariant must take into account the particular behavior of the constructor `including` and the operation `excluding` which eliminates all occurrences of its argument provided as a parameter.

```
context Collection(T) inv includingExcluding:
  T.allInstances()->forAll(x:T|
    self->including(x)->excluding(x)=self->excluding(x))
```

Furthermore determinateness properties of conversion operations can be expressed in this style with invariants as well. For example, the direct conversion from sequences to sets should be identical to a conversion where a conversion to bags is added or a conversion to ordered sets is added (or both, in any order). This is formally expressed in the invariants below and pictured in Fig. 7. We formulate these invariants for sequences, whereas analogous statements also hold for ordered sets.

```
context Sequence(T) inv asBagDoesNotChangeAsSet:
  self->asSet() = self->asBag()->asSet()
context Sequence(T) inv asOrderedSetDoesNotChangeAsSet:
  self->asSet() = self->asOrderedSet()->asSet()
context Sequence(T) inv asBagAsOrderedSetDoesNotChangeAsSet:
  self->asSet() = self->asBag()->asOrderedSet()->asSet()
context Sequence(T) inv asOrderedSetAsBagDoesNotChangeAsSet:
  self->asSet() = self->asOrderedSet()->asBag()->asSet()
```

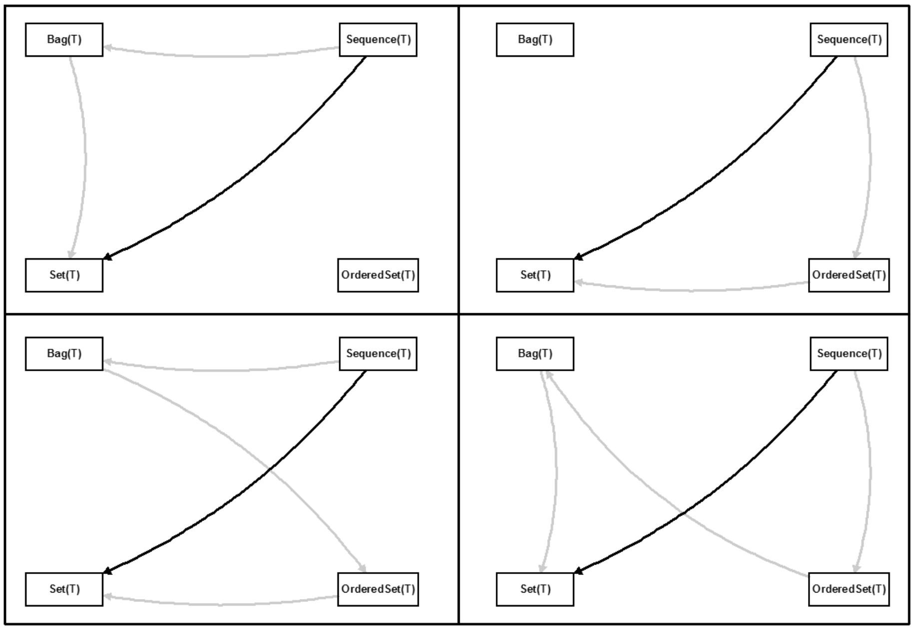

Fig. 7. Determinateness for `Sequence(T)::asSet():Set(T)`

A fundamental property of conversion operations is the requirement that the membership in the converted collection and the result collection coincides. We formulate this requirement for `asBag`. Analogous requirements must be made for `asSet`, `asSequence` and `asOrderedSet`.

```
context Collection(T) inv asBagPreservesMembership:
  T.allInstances()->forAll(x:T|
    self->includes(x) = self->asBag()->includes(x) and
    self->excludes(x) = self->asBag()->excludes(x))
```

All in all, we believe that formulating constraints in a more property oriented style with invariants is an addition to the more operation oriented style with pre- and postconditions in the OCL standards. With such invariants one can directly express properties which have to be deduced from pre- and postconditions in the operation oriented style. At least such invariants can help to make the intention of the standard more explicit and can be employed for the description of properties within OCL libraries [BD07,Opo09] or as test cases in OCL benchmarks [GKB08].

6 Conclusion

This paper has presented a concept for dealing with OCL ordered sets which is implemented as proposed here in USE [GBR07]. For this implementation we had to make several design decisions, for example, concerning the conversion operations or concerning the operation append on ordered sets which is in our implementation without effect if the parameter element is already present. We have discussed an alternative type hierarchy for OCL collections and several invariants which determine the interplay between collection operations.

Future research will cover a number of points to work on. An open question is how type parameters have to be dealt with. The preprocessor semantics mentioned above should be worked out in detail. Furthermore, additional details concerning the interplay between collection operations are probably missing. We understand our paper as a discussion offer to the OCL community in order to make a better proposal for the standard OCL library.

Acknowledgements

Thanks to the referees for the constructive comments and to the workshop participants, in particular Birgit Demuth and Markus Stumptner, for asking good questions.

References

[BD07] Matthias Bräuer and Birgit Demuth. Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support. In Holger Giese, editor, *MODELS Workshops*, LNCS 5002, pages 182–193. Springer, 2007.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[EGL89] Hans-Dieter Ehrich, Martin Gogolla, and Udo Walter Lipeck. *Algebraische Spezifikation Abstrakter Datentypen - Eine Einführung in die Theorie*. Leitfäden und Monographien der Informatik. Teubner, Stuttgart, 1989.

[EM85] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification*. Springer, Berlin, Germany, 1985.

[GBR07] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[GKB08] Martin Gogolla, Mirco Kuhlmann, and Fabian Büttner. A Benchmark for OCL Engine Accuracy, Determinateness, and Efficiency. In Krzysztof Czarnecki, editor, *Proc. 11th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2008)*, pages 446–459. LNCS 5301, Springer, Berlin, 2008.

[HB94] Paul F. Hoogendijk and Roland Carl Backhouse. Relational Programming Laws in the Tree, List, Bag, Set Hierarchy. *Science of Computer Programming*, 22(1-2):67–105, 1994.

[HL07] Sven Hartmann and Sebastian Link. Collection Type Constructors in Entity-Relationship Modeling. In Collette Parent, Klaus-Dieter Schewe, Veda C. Storey, and Bernhard Thalheim, editors, *Proc. Int. Conf. Conceptual Modeling (ER'2007)*, LNCS 4801, pages 307–322. Springer, 2007.

[OMG03] OMG, editor. *Object Constraint Language, Version 1.4*. OMG, 2003. OMG Document, `www.omg.org`.

[OMG04] OMG, editor. *OMG Unified Modeling Language Specification, Version 2.0*. OMG, 2004. OMG Document, `www.omg.org`.

[OMG06] OMG, editor. *Object Constraint Language, Version 2.0*. OMG, 2006. OMG Document `formal/06-05-01`, `www.omg.org`.

[Opo09] Joana Opoka. OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions of the Object Constraint Language. In Bran Selic and Andy Schürr, editors, *Proc. 12th Int. Conf. MODELS'2009, LNCS, Springer*, 2009.

[TS91] Val Tannen and Ramesh Subrahmanyam. Logical and Computational Aspects of Programming with Sets/Bags/Lists. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez-Artalejo, editors, *18th Int. Colloquium Automata, Languages and Programming (ICALP'1991)*, LNCS 510, pages 60–75. Springer, 1991.

[Wir90] Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 2003. 2nd Edition.

[Won95] Limsoon Wong. Polymorphic Queries Across Sets, Bags, and Lists. *ACM SIGPLAN Notices*, 30(4):39–44, April 1995.