Description

Collection is a container for a number of data items. For example,

"someText", 1, 0, 200

Is a collection containing four items.

When it comes to XMLdation environment and XML, collections offer a way to handle XML elements which have multiplicity higher than 1 (e.g. 0..n or 1..n). In practice, items within collection are either XML elements or data within XML elements.

What situations require the usage of collection?

Collection is required whenever an aspect of the rule requires multiplicity handling. I.e. when a part of the rule is under an element which has a multiplicity of 0..n or 1..n in the schema. Multiplicity is always checked from schema, meaning that rule has to be written with a collection even if there is another rule limiting the occurrence to 1.

Below is the reference schema viewed in myXML:

Schema element	Type	Limit
Message	Message	1...1
Header	HeaderType1	1...1
Id	Max35Text	1...1
TimeStamp	DateTime	1...1
ControlSum	decimal	1...1
Debtor	PartyIdentification	0...1
Name	Max140Text	0...1
Transaction	TransactionType1	1...*
Id	Max35Text	1...1
Amount	decimal	1...1
Debtor	PartyIdentification	0...1
Name	Max140Text	0...1
Creditor	PartyIdentification	1...1
Name	Max140Text	0...1

When a rule requires Transaction to be handled in the OCL statement section, collection is required.

In addition, please bear in mind that context is always ran whenever the type defined in it is found from the XML, so defining context to be *TransactionType1* will end in a rule which can access to all elements under the type, but that rule cannot access elements above it. For example, a rule making Debtor mandatory for every Transaction, it is sufficient to write the rule under the context of *TransactionType1* (It is also possible to make this rule by using collection and writing the rule under context *Message*, but the rule is a bit more complex).

However, if the goal of the rule is to compare values between Header/Id and Transaction/Id, we have to find a context which is common for both of them (*Message),* and this will lead to a situation where

Transaction / Id is under and occurrence number of 1..n. Therefore a collection is required.

Collection types

Before handling collections, different collection types in OCL should be understood,

Collection types available are:

- Bag - no order, may contain duplicates
- Set - no order, duplicates removed
- OrderedSet - ordered, duplicates removed
- Sequence - ordered, may contain duplicates

Most frequently, Bag is the most adequate collection type for handling XML elements.

For example, let's assume we need a rule to put Transaction/Id under collection, and the inputted XML-file has the following values separated between several transaction occurrences:

- <Transaction><Id>testId-1
- <Transaction><Id>testId-2
- <Transaction><Id>AAA
- <Transaction><Id>AAA

Depicting the collection in a bag or in sequence would ensue following values in it. Bag would not necessarily have the exact order.

{"testId-1", "testId-2", "AAA", "AAA"}

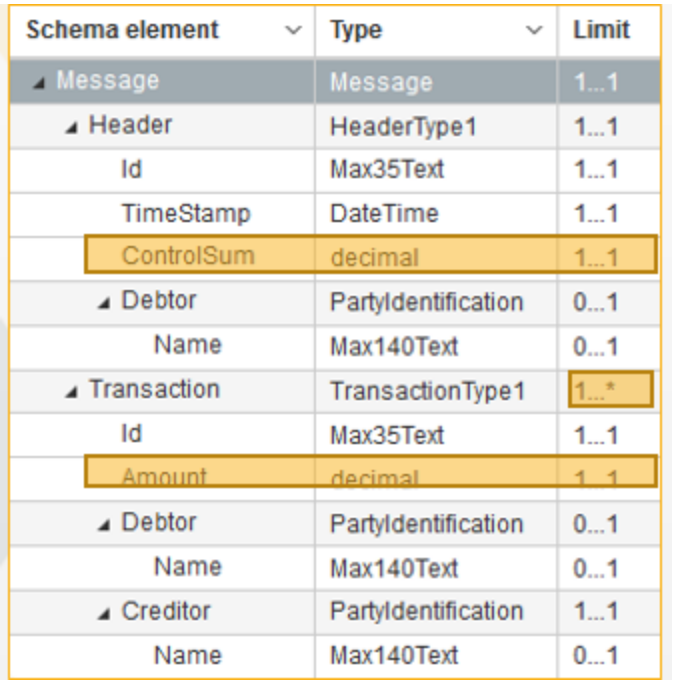
Depicing the collection in set or in orderedSet would ensue the following values in it:

{"testId-1", "testId-2", "AAA"}

The amount of items is different in bag and set, as set does not contain duplicates. Therefore comparing the amounts within a bag and set is a good way to check whether a collection contains duplicates.

Example 1: Sum of values within collection

An example scenario is that we need to calculate the sum of all Message/Transaction/Amount values and compare it to Message / Header / ControlSum value. Below is an image representation of the schema:



Schema element	Type	Limit
Message	Message	1...1
Header	HeaderType1	1...1
Id	Max35Text	1...1
TimeStamp	DateTime	1...1
ControlSum	decimal	1...1
Debtor	PartyIdentification	0...1
Name	Max140Text	0...1
Transaction	TransactionType1	1...*
Id	Max35Text	1...1
Amount	decimal	1...1
Debtor	PartyIdentification	0...1
Name	Max140Text	0...1
Creditor	PartyIdentification	1...1
Name	Max140Text	0...1

We cannot take the context *TransactionType1,* as it only gives us access for one singluar Amount element, and additionally we cannot access ControlSum with it. We have to select a common context for them both, *Message.*

From *Message*, it is fairly straightforward to target ControlSum, as the maximum occurrence numbers of ControlSum and all elements before it do not exceed one.

self.Header.ControlSum

However, we need to target and get a collection of all the Amount elements within message. Targeting it similarly as ControlSum would not work: self.Transaction.Amount, because there may be multiple transaction elements within schema. OCL would not know which of the transaction elements to target.

Here we need to use collection and decide what is the best collection type for our scenario. Order of elements within collection does not matter in this case, so we don't have to use Sequence or OrderedSet. We have to have all elements from transaction and not to remove duplicates so we cannot use either of the sets. This leaves us with Bag (sequence would work here as well).

In order to put Transaction elements into a collection, we have to use a method for it. All the methods for putting an item into a collection consists of an arrow notation after relevant item, followed with "as" and the name for desired collection type. In other words the available methods creating all four collection types are

- ->asBag()
- ->asSet()
- ->asSequence()
- ->asOrderedSet()

The relevant element in our case is Transaction, so OCL so far would be

```
self.Transaction->asBag()
```

Now we have a collection which contains all transaction elements, but we don't have an action for it yet. Now we need to target the exact element and use a method for calculating sum amount of all elements within collection, ->sum(). The next lines of OCL contain the full code for our rule

Context: Message
OCL:

```
self.Header.ControlSum =
self.Transaction->asBag().Amount->sum()
```

Please note, that the type of ControlSum and Amount is xs:decimal, so here we don't have to worry about conversion of types. Having a string data within element would already give an error from schema validation phase.

Example 2: Accessign a specific item within a collection

In the above example we did not to handle individual items within a collection. When a rule requires handling individual elements, it can be done with methods accessing a collection, e.g. ->forAll() or ->exists(). In these cases, a statement returning a boolean must be inserted inside the brackets after the method and this statement is ran against every instance of collection.

Collection methods do not require stating the collection type, and XML content is handled as if it is a sequence.

For example, if we wanted to check whether at least one Transaction / Id is identical to Header / Id, we can write the rule as follows

Context: Message
OCL:

```
self.Transaction->exists(transactionOcc | transactionOcc.Id = self.Header.Id)
```

Note the declaration of a variable, which is done after opening bracket of method and separated from the statement with vertical line, "|". The declared variable name here is "transactionOcc", and it depicts one occurrence of Transaction element. After the declaration, we are in a loop which goes through every item of Transaction. Here we can write the rest of the statement, to compare transaction Id to a header Id. Note that header Id is located from the context's point of view, even though statement is ran from inside transaction.

Below are definitions of two useful methods for handling individual items within a collection.

- *->exists()* returns true if **at least one** item in the collection returns true for the statement defined inside the brackets. Otherwise false is returned.

  Additionally the element before ->exists() has to be present. Otherwise false is returned.

- *->forAll()* returns true if **all** items in the collection return true for the statement defined inside the

brackets. Otherwise false is returned. When the element before forAll() is not present, true is returned.

Example 3 - Queries for collections

In the above example we had a rule for verifing that at least one Header / Id matches Transaction Id. In this validation report, how could we show the error in the best location possible?

The error message itself would be something along the lines of

> "Data within 'Header / Id' does not match with data within 'Transaction / Id'. At least one 'Transaction / Id' has to match with 'Header / Id'.

If we do not give a query, the error message will be returned in context, in *HeaderType1.* In some cases it can be sufficient, but it would be clearer to user if we are able to target the relevant elements. We could give the query as self.Transaction.Id, but this would return the error in every transaction/Id occurrence, which may not be senseful. In this case we can simply give the query for self.Header.Id, so the amount of error messages will always be one.

However, if we had used forAll in the above rule instead of exists, to verify that every occurrence of Transaction / Id matches with Header / Id, it would be helpful to give the error message to errenous occurrencs of Transaction. In that case, the query has to contain "reverse" forAll(), select(), which we can use to select the relevant occurrences.

So if the OCL-rule would be

Context: Message
OCL:

```
self.Transaction->forAll(transactionOcc | transactionOcc.Id = self.Header.Id)
```

Therefore the query could be:

```
self.Transaction->select(q | q.Id <> self.Header.Id)
```

Note that the variable name can be whatever the writer wants, and in this case q is selected as statement is simple. The statement part is then written as "reversion" of the rule, to return the occurrences of Id which do not match, so inequality comparison is used, <>.

This query would return the error message to statement, where Id's do not match. We can improve it to continue targeting after the select statement, by adding Id after it.

```
self.Transaction->select(q | q.Id <> self.Header.Id).Id
```

This would target the transaction Id which does not match with Header Id.