QVT 1.2 Revision Task Force

Ballot 2

"Preview 1"

15 January 2014

QVT 1.2 RTF Formal Issues Resolution Vote - Ballot No. 2

Poll start date: Wednesday, 22 January 2014 (01:00 AM EDT - 06:00 GMT)

Poll closing date: Wednesday , 5 February 2014 (07:00 PM EDT - 24:00 GMT)

What is being voted on: Proposed issue resolutions for the set of issues listed in the tables on the following pages. The proposer is listed for each resolution, the full text of the issues and corresponding resolutions can be found in the section following the issue tables.

- Only officially registered RTF members in good standing are allowed to vote
- Voters who do not vote in two successive ballots lose their good standing and will be removed from the RTF membership; they can only be reinstated by a TC vote
- Quorum for a vote is half the registered RTF members
- Simple majority (of non-abstaining votes) decides the vote
- Votes should be sent via email to the chair of the RTF(ed@willink.me.uk) as well as to the qvt-rtf@omg.org list.
- Members can submit their vote anytime between the poll start date and the poll closing date.
- During the polling interval, members are allowed to change their votes. The most recent vote will be assumed to supersede all previous votes cast by a member.
- The possible ways to vote are:
  - Yes
  - No
  - Abstain (Note: “Abstain” does not influence the voting result but *does* count towards quorum)
- Votes can be cast either for individual issues or for the entire block (but not a mix of both)

Block Vote

<Company name> votes {Yes | No | Abstain} for the entire block of proposed issue resolutions identified below and specified in the appendix to this document.

Individual Issues Vote (NB: ONLY if you choose not to use the Block Vote option above!) <Company name> votes as follows on the proposed issue resolutions specified in and specified in the appendix to this document. Then vote for one issue resolution per line, like this:

12345 {Yes | No | Abstain}

No Vote

A brief explanation for each No vote should be provided.

QVT 1.2 RTF Membership

Representative	Organisation	Status
Manfred Koethe	88solutions	
Pete Rivett	Adaptive	
Bernd Wenzel	Fachhochschule Vorarlberg	
Michael Wagner	Fraunhofer FOKUS	
Jishnu Mukerji	Hewlett-Packard	
Didier Vojtisek	INRIA	
Xavier Blanc	Laboratoire Informatique de Paris 6	
Nicolas Rouquette	NASA	
Andrius Strazdauskas	No Magic, Inc.	
Edward Willink	Nomos Software	(CHAIR)
Victor Sanchez	Open Canarias, SL	
Philippe Desfray	Softeam	
Laurent Rioux	THALES	

Revision Details

Table of Contents

Disposition: Resolved

Issue 13222: Explanation of the 'Model::removeElement' operation needs clarification.

Source:

Open Canarias, SL (Mr. E. Victor Sanchez, vsanchez(at)opencanarias.com)

Summary:

Suggestion: Change original explanation text: "Removes an object of the model extent. All links that the object have with other objects in the extent are deleted." by something similar to the following one: "Removes an object from the model extent. The object is considered removed from the extent if it is not a root object of the extent, and is not contained by any other object of the extent."

Resolution:

Simple change.

Revised Text:

In 8.3.5.4 Model::removeElement change

> Removes an object of the model extent. All links that the object have with other objects in the extent are deleted

to

> Removes an object of the model extent. All links between the object and other objects in the extent are deleted. References from collections to the ends of deleted links are removed. References from non-collections are set to null.

Disposition: **Resolved**

Issue 13264: Pag 63, Section 8.2.1.1 OperationalTransformation.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: "isubclass of Module (see: ImperativeOCL package)"

Discussion: Module doesn't belong to ImperativeOCL package.

Suggestion: remove "(see: ImperativeOCL package)"

Resolution:

Simple change.

Revised Text:

In the final editorial paragrpah of 8.2.1.1 OperationalTransformation change

is a subclass of `Module` (see: `ImperativeOCL` package),

to

is a subclass of `Module`,

Disposition: **Resolved**

Issue 13265: Page 65, Notation Section 8.2.1.3 Module.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: "configuration property UML::Attribute::MAX_SIXE: String

discussion: providing a context to a configuration property doesn't seem to make sense.

suggestion remove "UML::Attribute::"

Resolution:

It's not obviously sensible, but is it actually wrong?

If there are many configuration properties, it may be convenient to partition them hierarchically. cf. java.vm.specification.vendor.

The punctuation is however confusing; add a space.

Revised Text:

In 8.2.1.1 OperationalTransformation change

> Properties that are configuration properties are declared using the **configuration** qualifier keyword.
> **configuration property** UML::Attribute::MAX_SIZE: String;

to

> Properties that are configuration properties may have hierarchical scope. They are declared using the **configuration** qualifier keyword.
> **configuration property** UML::Attribute::MAX_SIZE : String;

Disposition: **Resolved**

Issue 13268: Page 73: Section 8.2.1.11 Helper.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: "the invocation of the operation returns a tuple"

discussion: it could be understood as an OCL Tuple, which doesn't apply.

suggestion: Replace "a tuple" by "an ordered tuple".

Analysis:

The QVTo specification is very loose in its usage of tuple /ordered-tuple. Time to analyze whether OrderedTupleType has any utility

The Pack/Unpack use case

```
var a:A := ...
var b:B := ...
var t := Tuple{a, b};    // OrderedTupleLiteralExp
...
var (a1,b1) := t;        // UnpackExp
```

An OrderedTupleLiteralExp followed by an UnpackExp allows miscellaneous information to be packed and unpacked without naming each element.

This may sometimes give a lexical access saving in constrast to using OCL Tuples:

```
var t := Tuple{a:A = ..., b:B = ...};    // TupleLiteralExp
...
-- var a1 := t.a;
-- var b2 := t.b;
```

As the example shows, the overall saving is not necessarily large or even positive. OrderedTuples introduce a hazard from mis-aligned positional parts.

However, this functionality is sufficiently clearly specified that outright removal would be a breaking change.

Recommendation: deprecate OrderedTupleLiteralExp/UnpackExp etc as unnecessary bloat.

The OrderedTuple::at() use case

8.2.2.7: *Tuple elements can be accessed individually using the at pre-defined operation.*

Oops. The OrderedTuple::at() operation is not specified anywhere else.

In particular, if at() was specified, its return type would be data dependent demonstrating the worst characteristics of reflection.

In contrast, access to a Tuple part has a well-defined type and, if a reflective Tuple access was provided, there is a possibility that it could be type safe.

Recommendation: Eliminate the suggestion that an at operation might exist.

The Helper/Mapping return use case

This issue started because of the bad wording for helper returns. If we try to construct a complete example we highlight new problems.

The major example is in 8.1.2

```
mapping Foo::foo2atombar () : atom:Atom, bar:Bar
merges foo2barPersistence, foo2atomFactory
{
   object atom:{name := "A_"+self.name.upper();}
   object bar:{ name := "B_"+self.name.lower();}
}
```

This uses named returns and consequently an OCL Tuple return and so poses no real confusion

Continuing we find

```
   var (atom: Atom, bar: Bar) := f.foo2atombar();
```

which unpacks an OCL Tuple return. This is not supported by an UnpackExp that requires an OrderedTuple. Surely it should be

```
   var t := f.foo2atombar();
--   var atom := t.atom;
--   var bar := t.bar;
```

There does not appear to be a full example of an OrderedTuple return. Reworking the example to use one, we could elimninate the return names giving

```
mapping Foo::foo2atombar () : Atom, Bar
merges foo2barPersistence, foo2atomFactory
{
   Tuple{
       object atom:{name := "A_"+self.name.upper();},
       object bar:{ name := "B_"+self.name.lower();}
    }
}
```

and provide a specification that a list of unnamed return types e.g. "Atom, Bar" is a shorthand for "Tuple(Atom, Bar)"

A Mapping of this form cannot do piecemeal assignment to the named result variables, so a great deal is lost and very little gained.

Recommendation: implicit OrderedTuple return types have dubious benefits and are barely motivated by the specification so we can eliminate them

A return must either be an optionally named Type or an OCL Tuple of named Types.

The asOrderedTuple Use Case

The Object::asOrderedTuple() : OrderedTuple(T) library function provides a limited reflective capability with a rather poor description.

Converts the object into an ordered tuple. If the object is already an ordered type, no change is done.

What is an ordered type? Is a Sequence or List an ordered type?

If the object is an OCL Tuple, the list of attributes become the anonymous content of the ordered tuple.

Is this really intended? I would expect the values to become the content. Either way this is a non-determinstic conversion.

Otherwise, the operation creates a new tuple with a unique element being the object.

I guess it means a single element OrderedTuple containing the source, but for a class, I would really like to see the properties/values.

The above functionlity would appear to be provided by

Tuple::keys() : Set(TypedElement) to get a set of the Tuple name+type parts.
Tuple::at(TypedElement) : OclAny to get a selected part value
and extended by
OclElement::asTuple() : Tuple to give a Tuple of name+type+value parts.

Recommendation: deprecate asOrderedTuple in favour of Tuple::keys/at in OCL.

Resolution:

The usage or OrderedTuple seems misguided since an ordinary OCL Tuple does very similar things without a positional hazard and without noticeable lexical overhead.

Only the case of TupleLiteralEx/UnpackExp is plausibly defined in QVT 1.1 and neither of the QVT implementations (SmartQVT or Eclipse QVT) support OrderedTuple.

Therefore eliminate TupleLiteralExp, UnpackExp and OrderedTupleType and OrderedTupleLiteralPart.

Note that there are no editing instructions for the grammar. It is not clear that OrderedTuples were really supported at all.

Revised Text:

In 8.1.12 replace

```
var f := lookForAFooInstance();
var (atom: Atom, bar: Bar) := f.foo2atombar();
```

by

```
var f := lookForAFooInstance();
var t := f.foo2atombar();          – Tuple(atom: Atom, bar: Bar)
var atom := t.atom;
var bar := t.bar;
```

In 8.1.14 replace

These are *mutable lists*, *dictionary types*, and *ordered tuples*.

by

These are *mutable lists* and *dictionary types*.

In 8.1.14 remove

An *ordered tuple* is a tuple whose slots are unnamed. It can be manipulated in a more flexible way than tuples with named slots since it can be packed and unpacked more directly.

var mytuple := **Tuple**{1,2,3}; // an ordered tuple literal
var (x,y,z) := mytuple; // unpacking the tuple into three variables

In Fig 8.4 remove

UnpackExp

In Fig 8.7 and accompanying explanatory text remove

OrderedTupleType, OrderedTupleLiteralExp, OrderedTupleLiteralPart

In 8.2.1.15 MappingOperation replace

After call resolution, all the parameters of the mapping are passed as a tuple. The parameters include, in this order: the context parameter

by

After call resolution, the parameters of the mapping include: the context parameter

In 8.2.2.16 ReturnExp replace

If the operation declares more than one result parameter, the value is assigned to the tuple representing the collection of results. This requires that the type of the value passed to the result expressions is a tuple.

by

If the operation declares more than one return parameter, the result is a tuple with a tuple part for each return parameter. When the return expression value is omitted, this tuple is created automatically from the assignable result parameters. Alternatively the return expression value may be an explicitly constructed tuple with one part for each return parameter.

Remove 8.2.2.22 UnpackExp

Remove 8.2.2.27 OrderedTupleType

Remove 8.2.2.31 OrderedTupleLiteralExp

Remove 8.2.2.32 OrderedTupleLiteralPart

In 8.3.3.1 repr remove Object::asOrderedTuple

Disposition: **Resolved**

Issue 13272: Page 83: Notation Section 8.2.1.22 MappingCallExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: This is called the “collect” shorthand

discussion: Since OCL provides a "collect" shorthand (f.i. doing aCollection.anOperationCall()), I would rather call "xcollect" shortand to avoid confusions.

suggestion: Replate "collect" by "xcollect".

Resolution:

OCL provides an 'implicit collect' shorthand, so 'imperative collect' is a better contrast..

Revised Text:

In 8.1.6 Inlining Mapping Operations replace

the *collect shorthand*

by

the *imperative collect shorthand*

In 8.2.1.21 MappingCallExp replace

the “collect” shorthand

by

the “imperative collect” shorthand

In 8.2.2.8 SwitchExp replace

the “collect” shorthand

by

the “imperative collect” shorthand

Disposition: **Resolved**

Issue 13273: Page 83: Notation Section 8.2.1.22 MappingCallExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: // shorthand of self.ownedElement->xcollect(i) i.map class2table();

discussion: It seems that the imperativeIteratExp has not been correctly notated.

suggestion: replace the text above by: // shorthand of self.ownedElement->xcollect(i | i.map class2table());

Resolution:

Yes.

Revised Text:

In 8.2.1.21 MappingCallExp replace

```
self.ownedElement->map class2table();
// shorthand of self.ownedElement->xcollect(i) i.map class2table();
```

by

```
self.ownedElement->map class2table();
// shorthand of self.ownedElement->xcollect(i | i.map class2table());
```

Disposition: **Resolved**

Issue 13274: Page 84: Notation Section 8.2.1.22 MappingCallExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: ->forEach (cl) cleaningTransf.map removeDups(cl);

discussion: EBNF and forExp suggest using always braces to enclose the forExp body.

Suggestion: Enclose the forEach's body with '{' and '}'. Note that there are two forExps in this section

Resolution:

Yes.

Revised Text:

In 8.2.1.21 MappingCallExp replace

```
  // first pass: cleaning the UML classes
  uml->objectsOfType(Class) // invoking the imported transformation
    ->forEach (cl) cleaningTransf.map removeDups(cl);
  // second pass: transforming all UML classes uml
    ->objectsOfType(Class)->forEach (cl)
    cl.map umlclass2javaclass (); // equivalent to: this.map
  umlclass2javaclass(cl)
```

by

```
  // first pass: cleaning the UML classes
  uml->objectsOfType(Class) // invoking the imported transformation
    ->forEach (cl) {
      cleaningTransf.map removeDups(cl);
    }
  // second pass: transforming all UML classes uml
    ->objectsOfType(Class)->forEach (cl) {
      cl.map umlclass2javaclass();
    }// equivalent to: this.map umlclass2javaclass(cl)
```

Disposition: **Resolved**

Issue 13275: Page 86: Notation Section 8.2.1.23 ResolveExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: // shorthand for mylist->forEach(i) i.late resolve(Table)

discussion: EBNF and forExp suggest using always braces to enclose the forExp body.

Suggestion: Enclose the forEach's body with '{' and '}'.

Resolution:

Yes.

Revised Text:

In 8.2.1.22 ResolveExp replace

```
    myprop := mylist->late resolve(Table);
    // shorthand for mylist->forEach(i) i.late resolve(Table)
```

by (and correct font size of)

```
    myprop := mylist->late resolve(Table);
    // shorthand for mylist->forEach(i) { i.late resolve(Table); }
```

Disposition: **Resolved**

Issue 13277: Page 87: Section 8.2.1.24 ObjectExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: "When an object expression is the body of an imperative collect expression (see xcollect in ImperativeLoopExp)"

discussion: the text should point ImperativeIterateExp instead.

suggestion: replace ImperativeLoopExp by ImperativeIterateExp.

Resolution:

Yes.

Revised Text:

In 8.2.1.24 ObjectExp replace

see xcollect in ImperativeLoopExp

by

see xcollect in ImperativeIterateExp

Disposition: **Resolved**

Issue 13278: Page 87: Notation Section 8.2.1.24 ObjectExp (03).

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: // shorthand for list->xcollect(x) object X{ … }

discussion: It seems that the imperativeIteratExp has not been correctly notated.

suggestion: replace the text above by: // shorthand for list->xcollect(x | object X{ … });

Resolution:

Yes.

Revised Text:

In 8.2.1.24 ObjectExp replace

// shorthand for list->xcollect(x) object X{ … }

by

// shorthand for list->xcollect(x | object X{ … })

Disposition: **Resolved**

Issue 13280: Page 90: Notation Section 8.2.2.4 WhileExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: compute (x:MyClass := self.getFirstItem()) while (x<>null) { … }

discussion: the compute expression's body requires the enclosing braces

Suggestion: replace the text above by "compute (x:MyClass := self.getFirstItem()) { while (x<>null) { … }}"

- Page 92: Semantics Section 8.2.2.4 ForExp

Problems text:

```
Collection(T)::forEach(source, iterator, condition,body) =
  do {
    count : Integer := 0;
    while (count <= source->size()) {
      var iterator := source->at(count+1);
      if (condition) continue;
      body;
      count += 1;
    };
  };
Collection(T)::forOne(source, iterator, condition,body) =
  forEach (i | condition) {
    body;
    break;
  }
```

Discussion: It seems that ForEach and forOne expression are not correctly expressed in QVTo terms.

Suggestion: Change the text above by:

```
Collection(T)::forEach(source, iterator, condition,body) =
  do {
    count : Integer := 1;
    while (count <= source->size()) {
      var iterator := source->at(count);
     if (condition) body;
      count += 1;
    };
  };
Collection(T)::forOne(source, iterator, condition,body) =
  forEach (iterator | condition) {
    body;
    break;
  }
```

Resolution:

Yes.

Yes.

Revised Text:

In 8.2.2.4 WhileExp replace

```
    compute (x:MyClass := self.getFirstItem()) while (x<>null) { … }
```

by

```
    compute (x:MyClass := self.getFirstItem()) { while (x<>null) { … } }
```

In 8.2.2.6 ForExp replace

```
    Collection(T)::forEach(source, iterator, condition,body) =
      do {
        count : Integer := 0;
        while (count <= source->size()) {
          var iterator := source->at(count+1);
          if (condition) continue;
          body;
          count += 1;
        };
      };
```

by

```
    Collection(T)::forEach(source, iterator, condition,body) =
      do {
        count : Integer := 1;
        while (count <= source->size()) {
          var iterator := source->at(count);
          if (condition) body;
          count += 1;
        };
      };
```

and

```
    Collection(T)::forOne(source, iterator, condition,body) =
      forEach (i | condition) {
        body;
        break;
      }
```

by

```
    Collection(T)::forOne(source, iterator, condition,body) =
      forEach (iterator | condition) {
        body;
        break;
      }
```

Disposition: **Resolved**

Issue 13282: Page 95: Notation Section 8.2.2.7 ImperativeIterateExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: list[condition]; // same as list->xselect(i; condition)

discussion: From the specified notation, the xselect expression is not well written.

suggestion: replace ';' by '|'

Resolution:

Yes.

Revised Text:

In 8.2.2.7 ImperativeIterateExp replace

list[condition]; // same as list->**xselect**(i; condition)

by

list[condition]; // same as list->**xselect**(i | condition)

Disposition: **Resolved**

Issue 13283: Page 95: Associations Section 8.2.2.8 SwitchExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: elsePart : OclExpresion {composes} [0..1]

discussion: For uniformity, the multiplicty should appear after the type of the association.

suggestion: change positions between "{composes}" and "[0..1].

Resolution:

Yes, and correct spelling of OclExpression too.

Revised Text:

In 8.2.2.8 SwitchExp replace

elsePart : OclExpresion {composes} [0..1]

by

elsePart : OclExpression [0..1] {composes}

Disposition: **Resolved**

Issue 13284: Page 100: Superclasses Section 8.2.2.8 LogExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: OperationCallExp

discussion: From figure 8.4 and 8.6, we can guess that LogExp should inherits from both, OperationCallExp and ImperativeExpression.

suggesition: add ImperativeExpression as a LogExp's superclass.

Resolution:

Yes.

Revised Text:

In 8.2.2.19 LogExp replace

Superclasses
`OperationCallExp`

by

Superclasses
`OperationCallExp, ImperativeExpression`

Disposition: **Resolved**

Issue 13285: Page 103: Associations Section 8.2.2.23 InstantiationExp.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: // column := self.attribute->forEach new(a) Column(a.name,a.type.name);

discussion: the foreach exp is not well written:

suggestion: replace the text above by "// column := self.attribute->forEach(a) { new Column(a.name,a.type.name) };

Resolution:

Yes.

Revised Text:

In 8.2.2. 23 InstantiationExp replace

// column := self.attribute->forEach **new(**a**)** Column(a.name,a.type.name);

by

// column := self.attribute->forEach(a) { **new** Column(a.name,a.type.name); }

Disposition: **Resolved**

Issue 13286: Page 103: Figure 8.7.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem: There are two figures to represent the type extensions.

suggestion: Remove the first figure and name the sencond one as Figure 8.7 - Imperative OCL Package - Type extensions.

Resolution:

There are actually 5 sub-diagrams of which the first two are duplicated by the third and fourth; presumably an editing artefact when the fifth was introduced for ListLiteralExp.

Revised Text:

In Figure 8.7 remove the first two sub-diagrams.

Disposition: **Resolved**

Issue 13288: Page 105: Associations Section 8.2.2.26 DictionaryType.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: (see DictLiteralValue).

discussion: DictLiteralValue deosn't exist. It must be DictLiteralExp.

suggestion: Replace "DictLiteralValue" by "DictLiteralExp".

Resolution:

Yes.

Revised Text:

In 8.2.2.26 DictionaryType replace

(see `DictLiteralValue`)

by

(see `DictLiteralExp`)

Disposition: **Resolved**

Issue 13290: Page 108: Section 8.3 Standard Library.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem's text: The OCL standard library is an instance of the Library metaclass.

discussion: It should obviously refer to the QVT Operational Mapppings standard library.

suggestion: replace "OCL" by "QVT Operational Mappings".

Resolution:

Yes.

Revised Text:

In 8.3 Standard Library replace

> This section describes the additions to the OCL standard library. The OCL standard library is an instance of the Library metaclass.

by

> This section describes the additions to the OCL standard library to form the QVT Operational Mappings Library, which is an instance of the Library metaclass.

Disposition: **Resolved**

Issue 13913: Typo in 'Model::rootObjects' signature.

Source:

Open Canarias, SL (Mr. E. Victor Sanchez, vsanchez(at)opencanarias.com)

Summary:

Section '8.3.5.3 rootObjects' has a typo in the signature of the operation defined: Model::rootobjects() : Set(Element). In compliance with the surrounding definitions and with the title of the section itself, it should read 'Model::rootObjects() : Set(Element), that is, the first letter of 'Objects', capitalized.

Suggestion: Replace 'Model::rootobjects() : Set(Element)' by the new 'Model::rootObjects() : Set(Element)'

Resolution:

Yes.

Revised Text:

In 8.3.5.3 rootObjects change

```
Model::rootobjects(): Set(Element)
```

to

```
Model::rootObjects(): Set(Element)
```

Disposition: **Resolved**

Issue 13989: Typos in signatures of "allSubobjectsOfType" and "allSubobjectsOfKind".

Source:

Open Canarias, SL (Mr. E. Victor Sanchez, vsanchez(at)opencanarias.com)

Summary:

Sections 8.3.4.7 and 8.3.4.9, page 110:

Operation signatures for 8.3.4.7 and 8.3.4.9 do not correspond with the title of sections.

Please update 8.3.4.7 from "Element::subobjects(OclType) : Set(Element)" to "Element::allSubobjectsOfType(OclType) : Set(Element)".

Please update 8.3.4.9 from "Element::subobjectsOfKind(OclType) : Set(Element)" to "Element::allSubobjectsOfKind(OclType) : Set(Element)"

Resolution:

Yes.

Revised Text:

In 8.3.4.7 allSubobjectsOfType change

```
Element::subobjects(OclType) : Set(Element)
```

to

```
Element::allSubobjectsOfType(OclType) : Set(Element)
```

In 8.3.4.9 allSubobjectsOfKind change

```
Element::subobjectsOfKind(OclType) : Set(Element)
```

to

```
Element::allSubobjectsOfKind(OclType) : Set(Element)
```

Disposition: **Resolved**

Issue 14549: Wrong Chapter reference on Page 2 Language Dimension

Source:

InterComponentWare AG (Markus von Rüden, markus.vonrueden(at)icw.de)

Summary:

On Page 2 (Page 18 in PDF) there is a wrong reference in chapter 2.2 Language Dimension at 1. and 2.

"Core; The Core language is described in Chapter 11.."

"Relations: The Relations language is described in Chapter 9. ..."

It should be 9 and 7 ;)

Resolution:

The Chapter 10 reference is wrong too.

Further wrong Chapter 9/11 references exist in the main text.

The above references are manual text that do not use FrameMaker cross-reference facilities.

Loading the QVT 1.1 FrameMaker files reveals 6 unresolved cross-references.

OMG specifications have Clauses not Chapters.

Revised Text:

<Review all cross-references and correct stale definitions>

<Replace all usage of Chapter by a cross reference to the corresponding clause>

Disposition: **Resolved**

Issue 14619: QVT 1.1 Opposite navigation scoping operator (Correction to Issue 11341 resolution).

Source:

Nomos Software (Dr. Edward Willink, ed(at)willink.me.uk)

Summary:

The resolution for Issue 11341 uses '.' as a property scoping operator.

This is inconsistent with OCL for which class scoping always uses '::' e.g enumeration is class::literal operation call is class::operation.

Note the clarifying discussion in OCL, justifying Class.allInstances, explains that dot is only for operations on instances.

Resolution:

Simple correction of '.' to '::' in the grammar.

Revised Text:

In 7.13.5 change

```
   <keyProperty> ::= <identifier> | 'opposite' '(' <classId> '.' <identifier> ')'
```

to

```
   <keyProperty> ::= <identifier> | 'opposite' '(' <classId> '::' <identifier> ')'
```

and

```
   <propertyTemplate> ::= <identifier> '=' <OclExpressionCS> |
   'opposite' '(' <classId> '.' <identifier> ')' '=' <OclExpressionCS>
```

to

```
   <propertyTemplate> ::= <identifier> '=' <OclExpressionCS> |
   'opposite' '(' <classId> '::' <identifier> ')' '=' <OclExpressionCS>
```

Disposition: **Resolved**

Issue 14620: QVT 1.1 Inappropriate ListLiteralExp inheritance (Correction to issue 12375 resolution).

Source:

Nomos Software (Dr. Edward Willink, ed(at)willink.me.uk)

Summary:

The resolution for Issue 12375 specifies that ListLiteralExp has CollectionLiteralExp as a superclass.

This leaves the behaviour of the inherited kind and part attributes unspecified and rather embarrassing.

ListLiteralExp (like DictLiteralExp) should inherit directly from LiteralExp.

Resolution:

Simple change.

Revised Text:

In 8.2.2.33 ListLiteralExp change

Superclasses
`CollectionLiteralExp (From EssentialOCL)`

to

Superclasses
`LiteralExp (From EssentialOCL)`

In Figure 8.7 change ListLiteralExp superclass from CollectionLiteralExp to LiteralExp.

Disposition: **Resolved**

Issue 15424: Figure 7.3.

Source:

Institute for Defense Analyses (Dr. Steven Wartik, swartik(at)ida.org)

Summary:

I am confused by something in the MOF QVT specification (version 1.1) and am not sure if it's an error or my own misunderstanding. Figure 7.3 (p. 24) has links from two instances of ObjectTemplateExp to a single instance of PropertyTemplateItem (the one in the middle). Figure 7.6 (p. 30) shows a composite association between ObjectTemplateExp and PropertyTemplateItem. Why then are there two links to the instance in Figure 7.3? Doesn't a composite association mean that a PropertyTemplateItem can be owned by only one ObjectTemplateExp?

Resolution:

The problem is that UML Object diagrams omit the decorations from instantiated relationships. Shrt of moving to a non-standard UMLX diagram, we can perhaps get away with some explanations.

Revised Text:

Above Figure 7.3 add

> Instance diagrams omit the usual UML decorations, so the reader should note that a top to bottom composition layout is used in Figure 7.3. Thus the PropertyTempateItem for 'attribute' is composed by the ObjectTemplateItem for 'Class' and composes the ObjectTemplateItem for 'Attribute'.

Disposition: **Resolved**

Issue 15917: bug in the uml to rdbms transformation example.

Source:

Xavier Dolques (xavier.dolques(at)laposte.net)

Summary:

In the UML to rdbms transformation code given in section A.2.3 :

-- mapping to update a Table with new columns of foreign keys

mapping Association::asso2table() : Table

when {self.isPersistent();}

{

init {result := self.destination.resolveone(Table);}

foreignKey := self.map asso2ForeignKey();

column := result.foreignKey->column ;

}

The mapping asso2table is supposed to update a table by adding new columns, but with the last line of the mapping, the existing columns are all replaced by the new ones. I suggest to replace the last line with :

column += result.foreignKey->column ;

Resolution:

Yes.

Revised Text:

In A.2.3 mapping Association::asso2table() change

```
column := result.foreignKey->column ;
```

to

```
column += result.foreignKey->column ;
```

Disposition: **Resolved**

Issue 15978: clause 8.3.1.4 Exception needs to document the taxonomy of Exception types in QVT1.1.

Source:

NASA (Dr. Nicolas F. Rouquette, nicolas.f.rouquette(at)jpl.nasa.gov)

Summary:

In particular, this taxonomy should explicitly include the AssertionFailed exception type that clause 8.2.2.20 refers to for an AssertExp.

Suggest defining a String message attribute for Exception; this would facilitate retrieving the message from a raise expression (clause 8.2.2.15)

Suggest defining AssertionFailed as a subtype of Exception.

Suggest defining 2 attributes in AssertionFailed corresponding to the severity and log expressions of the AssertExp (clause 8.2.2.20)

Resolution:

Yes. We need a toxonomy.

At the root is clearly Exception and this needs no parameters.

We need to introduce a derived StringException for the standard string-valued RaiseExp, and oops we need to modify the grammar to support this shorthand.

The LogExp-valued AssertionFailed is another derivation of Exception; no severity since it is always fatal.

Revised Text:

In 8.2.2.15 RaiseExp change

> The notation uses the raise keyword with the exception name as body. The exceptions can be provided as simple strings. In that case the implicit referred exception is the user Exception defined in the QVT standard library and the string is the argument of the raise expression.
>
> myproperty := self.something **default raise** "ProblemHere";

to

> The notation uses the raise keyword followed by the exception type name and arguments for one of the expression type name constructors.
>
> myproperty := self.something **default raise** StringException("ProblemHere");
>
> The exceptions can be provided as simple strings. This is a shorthand for raising a StringException with the string as the constructor argument
>
> myproperty := self.something **default raise** "ProblemHere";

Following 8.3.1.4 Exception add additional sub-sub-sub-sections

> <new sub-sub-sub-section> **StringException**
>
> The StringException supports the simple string-valued shorthand of a RaiseExp.
>
> **Superclasses**
> Exception
> **Constructor**
> StringException**(**reason `: String)`

Attributes
reason : String [1]

The reason provided on construction.

<new sub-sub-sub-section> **AssertionFailed**

The AssertionFailed exception supports a fatal AssertExp.

Superclasses
Exception
Constructor
AssertionFailed(reason : LogExp)
Attributes
reason : LogExp [1]

The reason provided on construction.

In 8.4.7 change

```
<raise_exp> ::= 'raise' <scoped_identifier> ('(' <arg_list>? ')')?
```

to

```
<raise_exp> ::= 'raise' <scoped_identifier> ('(' <arg_list>? ')')?
                | 'raise' <STRING>
```

Disposition: **Resolved**

Issue 18572: QVT atomicity.

Source:

Nomos Software (Dr. Edward Willink, ed(at)willink.me.uk)

Summary:

Clause 7.7 mandates fixed-point semantics for in-place transformations.

Please ask my bank to use a repeat-until-no-change in-place transformation for my next pay cheque:

new-balance = old-balance + deposit.

More seriously, the repeat is at the relation level, so if there are multiple applicable relations, the in-place result is specified to experience multiple updates in an indeterminate order. If relation A and then relation B are repeated to exhaustion, is relation A repeated again to accommodate relation B's changes?

Start again. QVTr and QVTc are declarative, therefore they express a single complex truth about the final outputs with respect to the original inputs. There are no observable intermediate steps. It is the responsibility of the transformation engine to ensure that the multiple actual output changes are observed as a single atomic transaction. In particular for in-place transformations, the engine must ensure that no input value is accessed after it is updated for output.

In regard to fixed-point semantics, repetition can only be determinate if it is the entire tranformation that is repeated, and whether to do so would seem to be a legitimate execution option. Therefore QVT should either not specify repetition at all, leaving it to the invoking engine, or specify it as an invocation option for a RelationCallExp.

If an in-place transformation does perform fixed-point repetition at the transformation level, it would seem that the whole repetition should still be a single atomic transaction so that outputs are never observable in an inconsistent partially transformed state between iterations. The engine must therefore iterate over candidate outputs rather than actual outputs.

Resolution:

Fixed point semantics can be achieved by a has-it-changed loop.

Revised Text:

In 7.7 In-place Transformations replace

> A transformation may be considered in-place when its source and target candidate models are both bound to the same model at runtime. The following additional comments apply to the enforcement semantics of an in-place transformation:
>
> • A relation is re-evaluated after each enforcement-induced modification to a target pattern instance of the model.
>
> • A relation’s evaluation stops when all the pattern instances satisfy the relationship.

by

> A transformation may be considered in-place when one or more source models are bound to one or more target models at runtime. Execution proceeds as if the source and target models are distinct with an atomic update of non-distinct models occurring on completion of the transformation. This implies that an implementation that operates in-place must take copies of the old state to avoid confusion with updated new state.
>
> Execution of a transformation should return a Boolean status to indicate whether any changes were made to target models. This status enables transformation applications to repeat execution until no changes occur where that is appropriate for the application.

Disposition: **Resolved**

Issue 19021: Inconsistent description about constructor names.

Source:

Open Canarias, SL (Mr. Adolfo Sanchez-Barbudo Herrera, adolfosbh(at)opencanarias.com)

Summary:

Problem:

Specification first says in the Constructor concept description: "The name of the constructor is usually the name of the class to be instantiated. However this is not mandatory. Giving distinct names allows having more than one constructor."

Later on in the Constructor notation: "The name of the constructor is necessarily the name of the context type"

This is inconsistent.

Discussion:

Indeed, the notation section statement seems to be correct since:

1. Looks like other programming languages, like Java.

2. More importantly, the instantiation expression would not be so obvious when constructing new objects, and would required to be changed.

Example:

If we have the following constructors:

constructor MyClass::MyClassConstructor(name : String) { name := name }

constructor MyClass::MyClass(name : String) { name := name + "v2" }

How can the instantiation expression refer the different constructor ?

- new MyClass("abc")

- new MyClassConstructor("abc")

- new MyClass::Constructor("abc")

The referred class in a InstantiationExp would not be enough. Changing instantiation expression to provide different name constructor doesn't seem sensible.

Proposed solution:

In section 8.2.1.13

Replace:

"A constructor does not declare result parameters. The name of the constructor is usually the name of the class to be

instantiated. However this is not mandatory. Giving distinct names allows having more than one constructor."

by

"A constructor does not declare result parameters and its name must be the name of the class to be instantiated."

Resolution:

This was discussed on https://bugs.eclipse.org/bugs/show_bug.cgi?id=421621.

Unless we abandon constructor diversity completely, the current AS imposes a needless implementation difficulty by requiring dynamic resolution of a statically known constructor. This can be

avoided by augmenting InstantiationExp.instantiatedClass with InstantiationExp.referredConstructor, which can refer to any statically determined constructor. We can therefore relax the contradictory restrictions on constructor name spelling.

Revised Text:

In 8.2.1.13 Constructor notation change

The name of the constructor is necessarily the name of the context type

to

The name of the constructor is usually the name of the context type

In 8.2.2.23 InstantiationExp add

`referredConstructor : Constructor [0..1]`

The constructor for the object to be created. The constructor may be omitted when implicit construction occurs with no arguments.

In Figure 8.5 add

Constructor
unidirectional reference from InstantiationExp to Constructor
-- forward role referredConstructor [0..1]
-- reverse role instantiationExp [*]

Disposition: **Resolved**

Issue 19096: Resolve expressions without source variable.

Source:

University of Paderborn (Christopher Gerking, christopher.gerking(at)upb.de)

Summary:

In contrast to 8.2.1.23 ResolveInExp, the prior section 8.2.1.22 ResolveExp does not explicitly state whether the source variable is optional for resolve expressions. For Eclipse QVTo, this leads to the situation the source variable is assumed to be mandatory. Consequently, the resolve expression in following snippet is invalid in Eclipse:

var p : EPackage = resolveone(EPackage);

I don't think that this restriction is desirable from the specification viewpoint.

Eclipse bugzilla: https://bugs.eclipse.org/bugs/show_bug.cgi?id=392156

Resolution:

The resolve needs a domain in which to search for target objects. For ResolveInExp this can be the traces of a designated mapping. For ResolveExp it would seem that some source objects must be supplied. However there seems no reason to prohibit a search everywhere and this could be achieved by specifying Element.allInstances(). Therefore an omitted ResolveExp sources can mean search all traces.

Revised Text:

In 8.2.1.22 ResolveExp after the first paragraph add

> The source object is optional. When no source object is provided, this expression inspects all the targets created or updated by all mapping operations irrespective of the source objects.

Disposition: **Resolved**

Issue 19121: Imprecise result types of resolveIn expressions.

Source:

University of Paderborn (Christopher Gerking, christopher.gerking(at)upb.de)

Summary:

Section 8.2.1.22 ResolveExp states the following about the result type of a resolve expression:

"If no target variable is provided, the type is either Object (the type representing all types, see Section 8.3.1) either a Sequence of Objects - depending on the multiplicity."

On top of that, Section 8.2.1.23 ResolveInExp states:

"The type of a ResolveInExp expression is computed using the same rules as for the type of a ResolveExp."

In case of a ResolveInExp, why can't we obtain the result type from the respective mapping?

Consider the following example

mapping EClass :: EClass2EPackage() : EPackage

The result of any resolveIn expression for that mapping is necessarily a subtype of EPackage. No need to cast this up to Object.

Resolution:

OclAny rather Object is of course the top type.

Revised Text:

In 8.2.1.23 ResolveInExp replace

The type of a *ResolveInExp* expression is computed using the same rules as for the type of a *ResolveExp*.

by

Type of a resolveIn expression

The type of a *ResolveInExp* expression depends on the type of the ‘target’ variable, the 'inMapping' operation and on the multiplicity indication (the ‘one’ property). The overall returned type is specified in terms of an intermediate resolved type.

If a 'target' variable is provided, the resolved-type is the type of the 'target' variable

Otherwise if an 'inMapping' is provided, .the resolved-type is the type of the 'inMapping'.

Otherwise the resolved-type is *Object* (the type representing all types, see Section 8.3.1).

If ‘one’ is true, the returned type is the resolved-type. Otherwise, the returned type is a Sequence of the resolved-type.

Disposition: Resolved

Issue 19146: Specify List::reject and other iterations.

Source:

Nomos Software (Dr. Edward Willink, ed(at)willink.me.uk)

Summary:

The current specification of List has a vague claim that all OCL Collection operations are available.

Are iterations available?

Are Sequence operations and iterations not available?

Are return types adjusted to List?

The specification needs to provide a clear model defining all the operations and iterations.

Resolution:

Specifying the operations is straightforward, but highlights some nasty inconsistencies such as two versions of insertAt.

The OCL 2.4 operations are included.

From Issue 13223 clarify wording of insertAt:

From Issue 13251 add remove operations.

A model must wait till OCL 2.5 provides an extensible library model.

Revised Text:

Replace 8.3.8 Operations On List

All operations of the OCL Collection type are available. In addition the following are available.

add

```
List(T)::add(T) : Void
```

Adds a value at the end of the mutable list. Synonym: **append**

prepend

```
List(T)::prepend(T) : Void
```

Adds a value at the beginning of the mutable list.

insertAt

```
List(T)::insertAt(T,int) : Void
```

Adds a value at the given position. The index starts at zero (in compliance with OCL convention).

joinfields

```
List(T)::joinfields(sep:String,begin:String,end:String) :String
```

Creates a string separated by sep and delimited with begin and end strings.

asList

```
Set(T)::asList() : List(T)
OrderedSet(T)::asList(T) : List(T)
Sequence(T)::asList(T) : List(T)
Bag(T)::asList(T) : List(T)
```

Converts a collection into the equivalent mutable list.

by

The following operations can be invoked on any list. A list type is a parameterized type. The symbol T denotes the type of the values. The operations include all those of the OCL Sequence type.

=

```
List(T)::=(s : List(T)) : Boolean
```

True if self contains the same elements as s in the same order.

```
post: result = (size() = s->size()) and
  Sequence{1..size()}->forAll(i | at(i) = s->at(i))
```

<>

```
List(T)::<>(c : List(T)) : Boolean
```

True if c is not equal to self.

```
post: result = not (self = c)
```

add

```
List(T)::add(T) : Void
```

Adds a value at the end of the mutable list.

```
post: size() = size@pre() + 1
post: Sequence{1..size@pre()}->forAll(i | at(i) = at@pre(i))
post: at(size()) = object
```

append

```
List(T)::append(object: T) : List(T)
```

Returns a new list of elements, consisting of all elements of self, followed by object.

```
post: result->size() = size() + 1
post: Sequence{1..size()}->forAll(i | result->at(i) = at(i))
post: result->at(result->size()) = object
```

asBag

```
List(T)::asBag() : Bag(T)
```

Returns a Bag containing all the elements from self, including duplicates. The element order is indeterminate.

```
post: result->forAll(elem | self->count(elem) = result->count(elem))
post: self->forAll(elem | self->count(elem) = result->count(elem))
```

asList

```
List(T)::asList(T) : List(T)
```

Returns a new list that is a shallow clone of self.

```
post: Sequence{1..size()}->forAll(i | result->at(i) = self->at(i))
```

asOrderedSet

```
List(T)::asOrderedSet() : OrderedSet(T)
```

Returns a OrderedSet that contains all the elements from self, in the same order, with duplicates removed.

```
post: result->forAll(elem | self ->includes(elem))
post: self->forAll(elem | result->count(elem) = 1)
post: self->forAll(elem1, elem2 | self->indexOf(elem1) < self->indexOf(elem2)
    implies result->indexOf(elem1) < result->indexOf(elem2) )
```

asSequence

```
List(T)::asSequence() : Sequence(T)
```

Returns a Sequence that contains all the elements from self, in the same order.

```
post: result->size() = size()
post: Sequence{1..size()}->forAll(i | result->at(i) = self->at(i))
```

asSet

```
List(T)::asSet() : Set(T)
```

Returns a Set containing all the elements from self, with duplicates removed. The element order is indeterminate.

```
post: result->forAll(elem | self->includes(elem))
post: self->forAll(elem | result->includes(elem))
```

at

```
List(T)::at(i : Integer) : T
```

Returns the element of the list at the i-th one-based index.

```
pre : 1 <= i and i <= size()
```

clone

```
List(T)::clone(T) : List(T)
```

Returns a new list that is a shallow clone of self.

```
post: Sequence{1..size()}->forAll(i | result->at(i) = self->at(i))
```

count

```
List(T)::count(object : T) : Integer
```

Returns the number of occurrences of object in self.

deepclone

```
List(T)::deepclone(T) : List(T)
```

Returns a new list that is a deep clone of self; that is a new list in which each element is in turn a deepclone of the corresponding element of self.

excludes

```
List(T)::excludes(object : T) : Boolean
```

True if *object* is not an element of *self*, false otherwise.

```
post: result = (self->count(object) = 0)
```

excludesAll

```
List(T)::excludesAll(c2 : Collection(T)) : Boolean
```

Does *self* contain none of the elements of *c2* ?

```
post: result = c2->forAll(elem | self->excludes(elem))
```

excludesAll

```
List(T)::excludesAll(c2 : List(T)) : Boolean
```

Does *self* contain none of the elements of *c2* ?

```
post: result = c2->forAll(elem | self->excludes(elem))
```

excluding

```
List(T)::excluding(object : T) : List(T)
```

Returns a new list containing all elements of self apart from all occurrences of object. The order of the remaining elements is not changed.

```
post:result->includes(object) = false
post: result->size() = self->size()@pre - self->count(object)@pre
post: result = self->iterate(elem; acc : List(T) = List{} |
    if elem = object then acc else acc->append(elem) endif )
```

first

```
List(T)::first() : T
```

Returns the first element in self.

```
post: result = at(1)
```

flatten

```
List(T)::flatten() : List(T2)
```

Returns a new list containing the recursively flattened contents of the old list. The order of the elements is partial.

```
post: result = self->iterate(c; acc : List(T2) = List{} |
    if c.oclType().elementType.oclIsKindOf(CollectionType)
    then acc->union(c->flatten()->asList())
    else acc->union(c)
    endif)
```

includes

```
List(T)::includes(object : T) : Boolean
```

True if *object* is an element of *self*, false otherwise.

```
post: result = (self->count(object) > 0)
```

includesAll

```
List(T)::includesAll(c2 : Collection(T)) : Boolean
```

Does *self* contain all the elements of *c2* ?

```
post: result = c2->forAll(elem | self->includes(elem))
```

includesAll

```
List(T)::includesAll(c2 : List(T)) : Boolean
```

Does *self* contain all the elements of *c2* ?

```
post: result = c2->forAll(elem | self->includes(elem))
```

including

```
List(T)::including(object : T) : List(T)
```

Returns a new list containing all elements of self plus object added as the last element.

```
post: result = append(object)
```

indexOf

```
List(T)::indexOf(obj : T) : Integer
```

The one-based index of object obj in the list.

```
pre : includes(obj)
post : at(result) = obj
```

insertAt

```
List(T)::insertAt(index : Integer, object : T) : List(T)
```

Returns a new list consisting of self with object inserted at the one-based position index.

```
pre : 1 <= index and index <= size()
post: result->size() = size() + 1
post: Sequence{1..(index - 1)}->forAll(i | result->at(i) = at(i))
post: result->at(index) = object
post: Sequence{(index + 1)..size()}->forAll(i | result->at(i + 1) = at(i))
```

insertAt

```
List(T)::insertAt(object : T, index : Integer) : Void
```

The list is modifed to consist of self with object inserted at the one-based position index.

```
pre : 1 <= index and index <= size()
post: size() = size@pre() + 1
post: Sequence{1..(index - 1)}->forAll(i | at(i) = at@pre(i))
post: at(index) = object
post: Sequence{(index + 1)..size()}->forAll(i | at(i) = at@pre(i - 1))
```

isEmpty

```
List(T)::isEmpty() : Boolean
```

Is *self* an empty list?

```
post: result = (self->size() = 0)
```

joinfields

```
List(T)::joinfields(sep:String,begin:String,end:String) :String
```

Creates a string separated by sep and delimited with begin and end strings.

```
post: result = begin + Sequence{1..size()}->iterate(i; acc : String = '' |
    acc + if i = 1 then '' else sep endif + at(i).toString())
  + end
```

last

```
List(T)::last() : T
```

Returns the last element in self.

```
post: result = at(size())
```

max

```
List(T)::max() : T
```

The element with the maximum value of all elements in self. Elements must be of a type supporting the max operation. The max operation - supported by the elements - must take one parameter of type T. Integer and Real fulfill this condition.

```
post: result = self->iterate(elem; acc : T = self->any(true) | acc.max(elem))
```

min

```
List(T)::min() : T
```

The element with the minimum value of all elements in self. Elements must be of a type supporting the min operation. The min operation - supported by the elements - must take one parameter of type T. Integer and Real fulfill this condition.

```
post: result = self->iterate(elem; acc : T = self->any(true) | acc.min(elem))
```

notEmpty

```
List(T)::notEmpty() : Boolean
```

Is *self* not an empty list?

```
post: result = (self->size() <> 0)
```

prepend

```
List(T)::prepend(object : T) : List(T)
```

Returns a new list consisting of object, followed by all elements in self.

```
post: result->size = size() + 1
post: result->at(1) = object
post: Sequence{1..size()}->forAll(i | result->at(i + 1) = at(i))
```

product

```
List(T)::product(c2: Collection(T2)) : Set(Tuple(first: T, second: T2))
```

The cartesian product operation of *self* and *c2*.

```
post: result = self->iterate(e1; acc: Set(Tuple(first: T, second: T2)) = Set{} |
        c2->iterate(e2; acc2: Set(Tuple(first: T, second: T2)) = acc |
             acc2->including(Tuple{first = e1, second = e2})))
```

remove

```
List(T)::remove(element : T) : Void
```

Removes .all elements from self equal to element.

```
post: result = self@pre->reject(e = element)
```

removeAll

```
List(T)::removeAll(elements : Collection(T)) : Void
```

Removes .all elements from self equal to any of elements.

```
post: result = self@pre->reject(e | elements->includes(e))
```

removeAll

```
List(T)::removeAll(elements : List(T)) : Void
```

Removes .all elements from self equal to any of elements.

```
post: result = self@pre->reject(e | elements->includes(e))
```

removeAt

```
List(T)::removeAt(index : Integer) : T
```

Removes .and returns .the list element at index. Returns invalid for an invalid index.

```
pre: 1 <= index and index <= size()
post: size() = size@pre() - 1
post: Sequence{1..index}->forAll(i | at(i) = at@pre(i))
post: Sequence{(index+1)..size()}->forAll(i | at(i) = at@pre(i+1))
post: result = at@pre(index)
```

removeFirst

```
List(T)::removeFirst() : T
```

Removes .and returns .the first list element. Returns invalid for an empty list.

```
pre: 1 <= size()
post: size() = size@pre() - 1
post: Sequence{1..size()}->forAll(i | at(i) = at@pre(i+1))
post: result = at@pre(1)
```

removeLast

```
List(T)::removeLast() : T
```

Removes .and returns .the last list element. Returns invalid for an empty list.

```
pre: 1 <= size()
post: size() = size()@pre - 1
post: Sequence{1..size()}->forAll(i | at(i) = at@pre(i))
post: result = at@pre(size@pre())
```

reverse

```
List(T)::reverse() : List(T)
```

Returns a new list containing the same elements but with the opposite order.

```
post: result->size() = self->size()
post: Sequence{1..size()}->forAll(i | result->at(i) = at(size() - (i-1)))
```

selectByKind

```
List(T)::selectByKind(type : Classifier) : List(T1)
```

Returns a new list containing the non-null elements of self whose type is *type* or a subtype of *type*. The returned list element type T1 is the type specified as *type*.

```
post: result = self
    ->collect(if oclIsKindOf(type) then oclAsType(type) else null endif)
    ->excluding(null)
```

selectByType

```
List(T)::selectByType(type : Classifier) : List(T1)
```

Returns a new list containing the non-null elements of self whose type is *type* but which are not a subtype of *type*. The returned list element type T1 is the type specified as *type*.

```
post: result = self
    ->collect(if oclIsTypeOf(type) then oclAsType(type) else null endif)
    ->excluding(null)
```

size

```
List(T)::size() : Integer
```

The number of elements in the collection *self*.

```
post: result = self->iterate(elem; acc : Integer = 0 | acc + 1)
```

subSequence

```
List(T)::subSequence(lower : Integer, upper : Integer) : Sequence(T)
```

Returns a new sub-List of self starting at number lower, up to and including element number upper.

```
pre : 1 <= lower and lower <= upper and upper <= size()
post: result->size() = upper -lower + 1
post: Sequence{lower..upper}->forAll(i | result->at(i - lower + 1) = at(i))
```

sum

```
List(T)::sum() : T
```

The addition of all elements in *self*. Elements must be of a type supporting the + operation. The + operation must take one parameter of type T. It does not need to be commutative or associative since the iteration order over a list is well-defined. Integer and Real fulfill this condition.

```
post: result = self->iterate(elem; acc : T = 0 | acc + elem)
```

union

```
List(T)::union (s : List(T)) : List(T)
```

Returns a new list consisting of all elements in self, followed by all elements in s.

```
post: result->size() = size() + s->size()
post: Sequence{1..size()}->forAll(i | result->at(i) = at(i))
post: Sequence{1..s->size()}->forAll(i | result->at(i + size()) = s->at(i)))
```

<New sub-sub-section number> Iterations on Lists

There are no iterations defined for Lists since Lists are mutable and iteration domains are immutable. However the iterations defined for Sequences may be used without explicitly converting the List to a Sequence.

Invocation of one the following list iterations returning non-Lists

```
aList->iteration(...)
```

is therefore shorthand for

```
aList->asSequence()->iteration(...)
```

```
List(T)::any(i : T[?]) : T[?]
List(T)::collect(i : T[?]) : Collection(T1)
List(T)::collectNested(i : T[?]) : Collection(T1)
List(T)::exists(i : T[?]) : Boolean[?]
List(T)::exists(i : T[?], j : T[?]) : Boolean[?]
List(T)::forAll(i : T[?]) : Boolean[?]
List(T)::forAll(i : T[?], j : T[?]) : Boolean[?]
List(T)::isUnique(i : T[?]) : Boolean
List(T)::iterate(i : T[?]; acc : T2[?]) : T2[?]
List(T)::one(i : T[?]) : Boolean
```

Invocation of one the following list iterations returning Lists

```
aList->iteration(...)
```

is shorthand for

```
aList->asSequence()->iteration(...)->asList()
```

```
List(T)::reject(i : T[?]) : List(T)
List(T)::select(i : T[?]) : List(T)
List(T)::sortedBy(i : T[?]) : List(T)
```

<New sub-sub-section number> Operations on Collections

The following operations are added to the standard OCL collections

Collection::asList

```
Collection(T)::asList(T) : List(T)
```

Returns a new list containing all the elements of a collection. Whether the order is determinate depends on the derived collection type..

```
post: result->size() = size()
post: self->asSet()->forAll(e | result->count(e) = self->count(e))
```

Collection::clone

```
Collection(T)::clone(T) : Collection(T)
```

Collections are immutable so a clone returns self.

```
post: result = self
```

Collection::deepclone

```
Collection(T)::deepclone(T) : Collection(T)
```

Collections are immutable so a deep clone returns self.

```
post: result = self
```

<New sub-sub-section number> Operations on Bags

The following operations are added to the standard OCL bags

Bag::asList

```
Bag(T)::asList(T) : List(T)
```

Returns a new list containing all the elements of a bag in an indeterminate order.

Bag:: clone

```
Bag(T)::clone(T) : Bag(T)
```

Bags are immutable so a clone returns self.

Bag:: deepclone

```
Bag(T)::deepclone(T) : Bag(T)
```

Bags are immutable so a deep clone returns self.

<New sub-sub-section number> Operations on OrderedSets

The following operations are added to the standard OCL ordered sets

OrderedSet::asList

```
OrderedSet(T)::asList(T) : List(T)
```

Returns a new list that contains all the elements an ordered set in the same order.

```
post: Sequence{1..size()}->forAll(i | result->at(i) = self->at(i))
```

OrderedSet:: clone

```
OrderedSet(T)::clone(T) : OrderedSet(T)
```

OrderedSets are immutable so a clone returns self.

OrderedSet:: deepclone

```
OrderedSet(T)::deepclone(T) : OrderedSet(T)
```

OrderedSets are immutable so a deep clone returns self.

<New sub-sub-section number> Operations on Sequences

The following operations are added to the standard OCL sequences

Sequence::asList

```
Sequence(T)::asList(T) : List(T)
```

Returns a new list that contains all the elements a sequence in the same order.

```
post: Sequence{1..size()}->forAll(i | result->at(i) = self->at(i))
```

Sequence:: clone

```
Sequence(T)::clone(T) : Sequence(T)
```

Sequences are immutable so a clone returns self.

Sequence:: deepclone

```
Sequence(T)::deepclone(T) : Sequence(T)
```

Sequences are immutable so a deep clone returns self.

<New sub-sub-section number> Operations on Sets

The following operations are added to the standard OCL sets

Set::asList

```
Set(T)::asList() : List(T)
```

Returns a new list containing all the elements of a set in an indeterminate order.

Set:: clone

```
Set(T)::clone() : Set(T)
```

Sets are immutable so a clone returns self.

Set:: deepclone

```
Set(T)::deepclone() : Set(T)
```

Sets are immutable so a deep clone returns self.

Disposition: Resolved

Issue 19178: What happens when an exception is thrown by an exception handler.

Source:

Nomos Software (Dr. Edward Willink, ed(at)willink.me.uk)

Summary:

QVTo provides no guidance as to how a nested exception should be handled.

Suggest that like C++, a nested exception is ill-formed and the transformation terminates.

Resolution:

The QVTo exception mechanism is lightweight.

Throwing an exception while handling another makes it very difficult to guarantee that the handler for the first executes correctly.

Therefore declare nested exceptions as ill-formed.

Revised Text:

In 8.2.2.13 TryExp add

> The selected exceptClause completes before the TryExp completes. Consequently if an exception is raised during execution of the exceptClause, the execution of the exceptClause cannot complete and so execution of the transformation terminates without any further changes occurring. The trace records may be examined to determine what actions the transformation performed prior to termination.

Disposition: **Resolved**

Disposition: Duplicate / Merged

Issue 13223: explanation of the operation: 'List(T)::insertAt(T,int).

Source:

Open Canarias, SL (Mr. E. Victor Sanchez, vsanchez(at)opencanarias.com)

Summary:

The explanation of the operation: 'List(T)::insertAt(T,int) : Void' in Section 8.3.8.3 has a mistake concerning the index associated with the first element for OCL collections. The index starts at one, '1', not zero, '0'. Suggestion: Substitute the word 'zero' by the word 'one', so that the text reads: "The index starts at one (in compliance with OCL convention)."

Resolution:

Yes. Improved wording in Issue 19146.

Disposition: **See issue 19146 for disposition**

Issue 13228: Missing operations on Lists.

Source:

Open Canarias, SL (Mr. E. Victor Sanchez, vsanchez(at)opencanarias.com)

Summary:

For 'removeAt' I specify 'one' as the starting index value. Suggestion: - List(T)::remove(T) : T Removes a value from the list. - List(T)::removeAt(int) : T Removes a value from the mutable list at the given position. The index starts at one (in compliance with OCL convention). The return value is the value removed from the mutable list. - List(T)::clear() : Void Removes all values in the mutable list.

Resolution:

Yes. Revised wording in 132 is explicitly one-based.

Disposition: **See issue 19146 for disposition**

Issue 13251: add the following operations to mutable lists.

Source:

Open Canarias, SL (Mr. E. Victor Sanchez, vsanchez(at)opencanarias.com)

Summary:

In the spirit of issue #13228, and in conversation with Mariano Belaunde, we think it is worth considering also adding the following operations to mutable lists. As specific usages of 'removeAt()': 'removeFirst()' and 'removeLast()'. And also: 'removeAll(Collection(T))'. Notice that this one includes the case of removing values specified inside mutable lists as long as ListType inherits CollectionType. Suggestion: Add the following text to section 8.3.8: - List(T)::removeFirst() : T Removes the first value of the mutable list. The return value is the value removed. - List(T)::removeLast() : T Removes the last value of the mutable list. The return value is the value removed. - List(T)::removeAll(Collection(T)) : Void Removes from the mutable list all the values that are also present in the specified collection.

Resolution:

And remove() from Issue 13228. A List is not a Collection so we need two removeAll's.

Additional operations in Issue 19146.

Disposition: **See issue 19146 for disposition**

Issue 13987: Minor typographical error in ImperativeIterateExp notation.

Source:

Open Canarias, SL (Mr. E. Victor Sanchez, vsanchez(at)opencanarias.com)

Summary:

Section 8.2.2.7, page 95:

The text "list[condition]; // same as list->xselect(i; condition)" should read "list[condition]; // same as list->xselect(i | condition)". Notation of imperative iterate expressions state that the condition section must be separated from prior artifacts (iterators or target initializations), where present, by a "|" character, not a semicolon ";".

Resolution:

Duplicates in part Issue 13282.

Disposition: **See issue 13282 for disposition**

Issue 15524: Rule Overriding in QVTr.

Source:

Nomos Software (Dr. Edward Willink, ed(at)willink.me.uk)

Summary:

You have reached the edge of the specification as written.

1: Yes

2: Yes

3: Yes

4: Yes

I gave some consideration to this for UMLX.

I felt that an abstract 'rule' could define a 'subrule' obligation, which would require an identical match signature, since if the override was narrower it would not fulfill the obligation and if it was wider the additional width would not be permitted by the invocation of the abstract 'rule'.

I felt that all concrete rules should always be matched to ensure that addition of extended functionality did not change previous behaviour. This complies with UMLX's all maximal matches philosophy. Keys in QVTr, Evolution Ids in UMLX can ensure that derived rules reuse inherited matches.

I think a transformation being both a package and a class introduces some difficult compatibility issues to be studied.

Transformation extension is also poorly defined giving additional imprecision when considering the combination of transformation extension and rule override.

My ideas for UMLX were not complete but I think that they may be sounder than QVTr's.

Resolution:

This issue was inadvertently raised from Issue 15417 correspondence. Close it as merged even though Issue 15417 needs further work.

Disposition: **See issue 15417 for disposition**

Issue 19095: Not possible to remove from a mutable List.

Source:

University of Paderborn (Christopher Gerking, christopher.gerking(at)upb.de)

Summary:

A List is a mutable type. However, there is only an interface for adding to a List, not for removing from a List. Analogously to the operation

List(T)::add(T) : Void

there should be something like:

List(T)::remove(T) : Void

Resolution:

Duplicates in part Issue 13251.

Disposition: **See issue 19146 for disposition**

Issue 19174: List does not support asList().

Source:

Nomos Software (Dr. Edward Willink, ed(at)willink.me.uk)

Summary:

The asList() operation is defined for all concrete collection types. Consequently, it should be defined on List itself (returning self).

Suggestion: specify the operation for the Collection type:

Collection(T)::asList() : List(T)

Add another redefinition for the List type:

List(T)::asList() : List(T)

Resolution:

The additional operation is shown in the Issue 19146 resolution..

Disposition: **See issue 19146 for disposition**