QVT Declarative Documentation

Edward Willink and contributors

Chapter 1. Overview and Getting Started

For a quick demonstration of QVTc or QVTr editing and execution go to Getting Started.

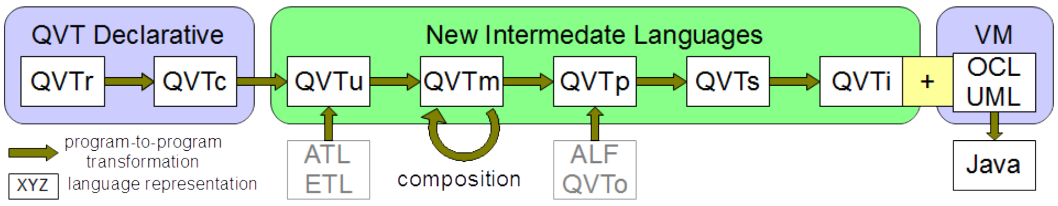
A PDF version of this documentation is available at QVTd 0.13.0 Documentation.

1.1. What is *QVT*?

The Query/View/Transformation language is the model transformation language specified by the *OMG* (Object Management Group). Or rather it is three specified languages to which Eclipse QVTd adds five intermediate languages.

- QVTo – Operation Mappings is an imperative transformation language supported by the Eclipse QVTo project.
- QVTr – Relations is a rich declarative transformation language.
- QVTc – Core is a simple core declarative transformation language.

QVTc and QVTr, generically referred to as QVTd, are supported the Eclipse QVTd project (QVT Declarative) and documented here.


QVT Declarative
QVTr
QVTc
New Intermediate Languages
QVTu
QVTm
QVTp
QVTs
QVTi
+
VM
OCL
UML
Java
program-to-program transformation
XYZ language representation
ATL
ETL
composition
ALF
QVTo

The Eclipse QVTd support involves a transformation chain from QVTr via QVTc to executable form. The stages in this chain are:

- QVTu – a unidirectional declarative transformation language – a simplification of QVTc.
- QVTm – a minimal declarative transformation language – a further simplification of QVTc.
- QVTp – a partitioned declarative transformation language – a further simplification of QVTc.
- QVTs – a graphical declarative transformation language suitable for schedule determination.
- QVTi – an executable imperative transformation language – a variant of QVTc.
- Java – the ultimate executable form

Note that QVTu, QVTm, QVTp, QVTs and QVTi are not defined by the current OMG specification. It is possible that they may contribute to a future specification.

1.1.1. Modeling Layers

- *EMF is Modeled Structure*

The Eclipse Modeling Framework (*EMF*) supports the definition of structural meta-models and the subsequent use of models conforming to these meta-models. EMF also supports generating of Java code to represent the meta-models. Additional Java code can be provided to add behavior to the structural meta-models.

- *OCL is EMF and Modeled Behavior*

OCL provides a modeling language that allows the behavior to be embedded within the structural meta-models or provided as a complement to those meta-models. As a modeling language, OCL understands the models and so OCL code is much more compact than the equivalent Java. OCL code can be statically checked, whereas the corresponding Java code often uses reflection and so cannot be checked.

- *QVT is Modeled Model Execution*

Eclipse QVTd is an implementation of the OMG QVT 1.3 specification for use with Ecore and UML meta-models.

Eclipse QVTd exploits the extensibility of the Eclipse OCL Pivot model.

1.2. How Does It Work?

QVTr and QVTc are programming languages, so you will want to edit them, execute them and debug them.

1.2.1. Editing

Eclipse QVTd provides QVTr, QVTc and QVTi editors developed using Xtext. The QVTc editor may be used to view or maintain the intermediate QVTu, QVTm and QVTp models.

The editor may be opened conventionally by double clicking a *.qvtr, *.qvtc, or *.qvti text file name in an Explorer view. This displays the source text with syntax highlighting and an outline of the Abstract Syntax.

The QVTc and QVTi editors may also be used to view *.qvtcas and *.qvtias XMI files as source text.

1.2.1.1. Status

These editors have been available since QVTd 0.9.0. They are useful, but currently have only limited well-formedness validation.

Editing *.qvtcas and *.qvtias files should be possible but is not recommended in the current release.

Hopefully 1.0.0 (Oxygen) will offer UMLX as a graphical alternative to the QVTr textual exposition of a transformation.

1.2.2. Execution

The 0.13.0 release provides one QVTr and one QVTc example project that be installed and executed. However execution functionality is very new and not suitable for more than experimental use.

1.2.2.1. Status

QVTi execution has been available since 0.11.0 (Luna). It demonstrates the extensibility of the Eclipse OCL interpreter, Java code generator and debugger. QVTi is a low level intermediate; it is not intended as a primary programming language.

A very preliminary form of QVTc execution was available in 0.12.0 (Mars) using an Epsilon prototype of the transformation chain.

0.13.0 (Neon) introduces a Java re-implementation of the full transformation chain so that QVTr and QVTc transformations can be executed. It is only suitable for researchers.

The current execution supports only creation of output models. Checking, updated, incremental, in-place execution and views are work in progress.

Hopefully 1.0.0 (Oxygen) will have more substantial functionality and will be used internally to replace some of the manual Java transformations by QVTr / UMLX transformations.

1.2.3. Debugger

The QVTi debugger extends the OCL debugger.

1.2.3.1. Status

The further extension to provide QVTc and QVTr debugging is work in progress.

1.3. Who is Behind Eclipse QVTd?

Eclipse QVTd is an Open Source project. All code has been developed under the auspices of Eclipse.

Eclipse QVTd is a largely one man development by Ed Willink who has been the OMG QVT *RTF* (Revision Task Force) chair since QVT 1.2. Expect further revisions of the QVT specification to exploit feedback from the Eclipse QVTo and QVTd projects.

There is now a significant personnel and corporate overlap between the Eclipse QVTd committers and the OMG QVT RTF and so Eclipse OCL is pioneering solutions to many of the under-specification problems in the OCL specification.

Ed Willink is also project lead of the Eclipse OCL where the new pivot-based implementation prototypes solutions to many problems with the OMG OCL specification for which Ed Willink has been the RTF chair since OCL 2.4.

The many aspects of OCL and QVTd are converging; help welcome.

1.4. Getting Started

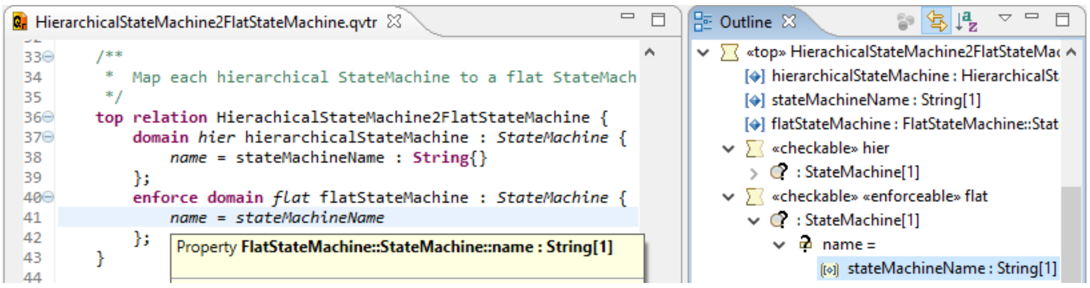
Warning: Eclipse QVTd 0.13.0 execution is not sufficiently mature for more than experimental/ research usage.

For a very quick demonstration of QVTc or QVTr you may install the appropriate example project.

1.4.1. QVTr Example Project

Invoke **File->New->Example...** then select **Examples** then **QVT (Query/View/Transformation Language) Projects** then select either **QVTr HSTM2FSTM Project** then **Finish** to create a small example project called **org.eclipse.qvtd.examples.qvtrelation.hstm2fstm**.

A QVTr or QVTc editor opens to show the transformation.



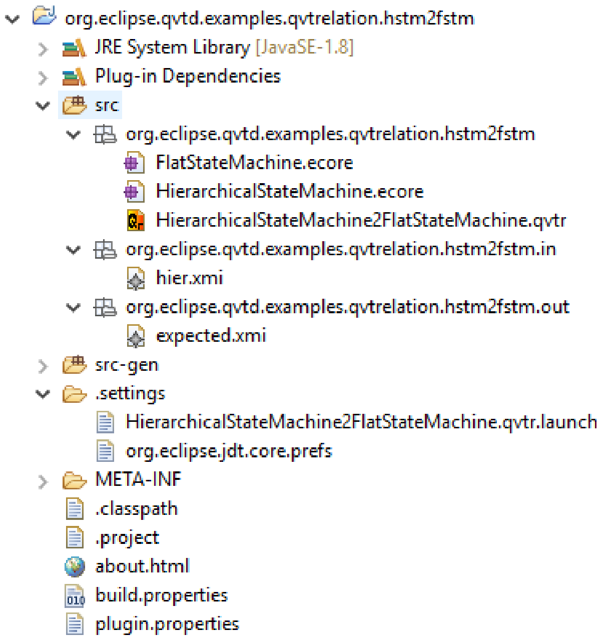
```
    /**
     *  Map each hierarchical StateMachine to a flat StateMach
     */
    top relation HierarchicalStateMachine2FlatStateMachine {
        domain hier hierarchicalStateMachine : StateMachine {
            name = stateMachineName : String{}
        };
        enforce domain flat flatStateMachine : StateMachine {
            name = stateMachineName
        };    Property FlatStateMachine::StateMachine::name : String[1]
    }
```

```
«top» HierarchicalStateMachine2FlatStateMac
    hierarchicalStateMachine : HierarchicalSt
    stateMachineName : String[1]
    flatStateMachine : FlatStateMachine::Stat
    «checkable» hier
        : StateMachine[1]
    «checkable» «enforceable» flat
        : StateMachine[1]
            name =
                stateMachineName : String[1]
```

The image shows one of the mappings with hovertext elaborating the declaration referenced by **name**.

The QVT editors extend the OCL editor and so the OCL context menu provides useful functionality such as saving the transformation in Abstyract Syntax form.

A QVTr project is currently just a normal Java project. (A QVTd nature may be added in the future to support auto-building.)



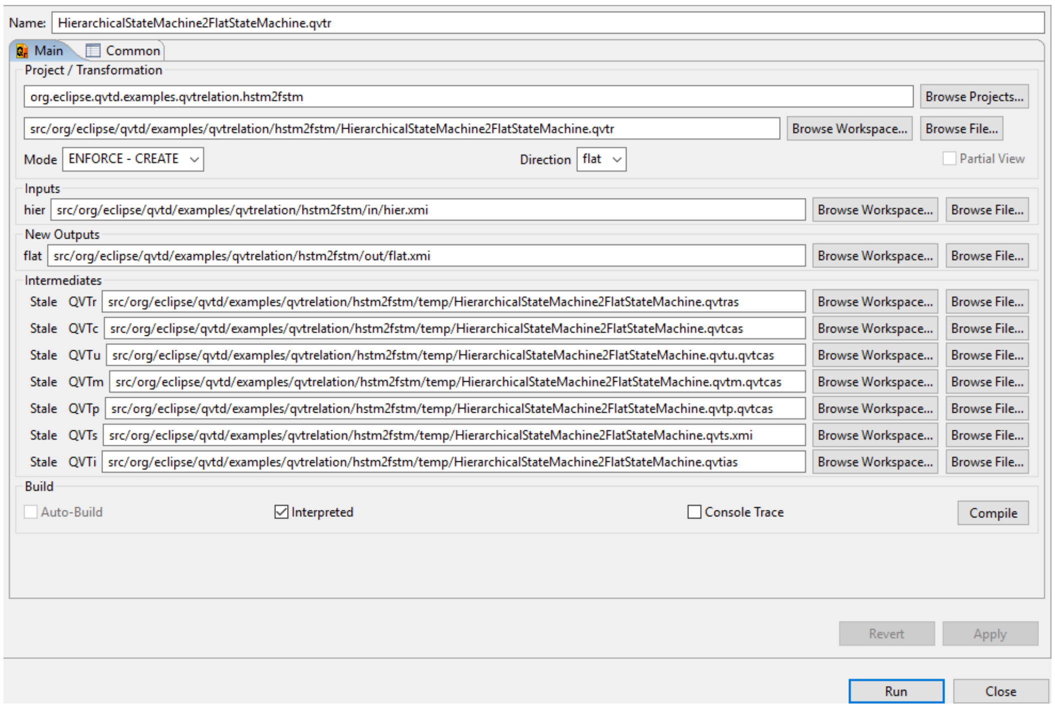
```
org.eclipse.qvtd.examples.qvtrelation.hstm2fstm
    JRE System Library [JavaSE-1.8]
    Plug-in Dependencies
    src
        org.eclipse.qvtd.examples.qvtrelation.hstm2fstm
            FlatStateMachine.ecore
            HierarchicalStateMachine.ecore
            HierarchicalStateMachine2FlatStateMachine.qvtr
        org.eclipse.qvtd.examples.qvtrelation.hstm2fstm.in
            hier.xmi
        org.eclipse.qvtd.examples.qvtrelation.hstm2fstm.out
            expected.xmi
    src-gen
    .settings
        HierarchicalStateMachine2FlatStateMachine.qvtr.launch
        org.eclipse.jdt.core.prefs
    META-INF
    .classpath
    .project
    about.html
    build.properties
    plugin.properties
```

The QVTr Hierarchical to Flat State Machine example contains

- **HierarchicalStateMachine2FlatStateMachine.qvtr** - the transformation
- **HierarchicalStateMachine.ecore** - the source metamodel
- **FlatStateMachine.ecore** - the target metamodel

- **hier.xmi** - an example model to exercise the execution
- **expected.xmi** - the expected execution result
- **HierarchicalStateMachine2FlatStateMachine.qvtr.launch** - a launch configuration for execution

You may execute the transformation on the **hier.xmi** input by invoking **Run->Run Configurations...** then **QVTr (Relations) transformation** then **HierarchicalStateMachine2FlatStateMachine.qvtr**.



Name: HierarchicalStateMachine2FlatStateMachine.qvtr

Main | Common

Project / Transformation

org.eclipse.qvtd.examples.qvtrelation.hstm2fstm

src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/HierarchicalStateMachine2FlatStateMachine.qvtr

Mode: ENFORCE - CREATE   Direction: flat   Partial View

Inputs

hier: src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/in/hier.xmi

New Outputs

flat: src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/out/flat.xmi

Intermediates

Stale	QVTr	src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/temp/HierarchicalStateMachine2FlatStateMachine.qvtras
Stale	QVTc	src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/temp/HierarchicalStateMachine2FlatStateMachine.qvtcas
Stale	QVTu	src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/temp/HierarchicalStateMachine2FlatStateMachine.qvtu.qvtcas
Stale	QVTm	src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/temp/HierarchicalStateMachine2FlatStateMachine.qvtm.qvtcas
Stale	QVTp	src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/temp/HierarchicalStateMachine2FlatStateMachine.qvtp.qvtcas
Stale	QVTs	src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/temp/HierarchicalStateMachine2FlatStateMachine.qvts.xmi
Stale	QVTi	src/org/eclipse/qvtd/examples/qvtrelation/hstm2fstm/temp/HierarchicalStateMachine2FlatStateMachine.qvtias

Build

[ ] Auto-Build   [✓] Interpreted   [ ] Console Trace

The launch is currently cluttered by an excess of development information and not yet implemented facilities. The important fields are:

1.4.1.1. Project / Transformation

The project name provides a default name against which other filenames are resolved to reduce screen clutter.

The transformation name is the transformation to be executed.

The mode can only be enforce creation/overwrite of the output model at present.

The direction selects the output direction when executing a multi-directional transformation.

1.4.1.2. Inputs

The file bound to each input domain must be specified.

1.4.1.3. New outputs

The file bound to each output domain must be specified.

1.4.1.4. Intermediates

The location of each intermediate file in the transformation chain is identified. Their default location is a **temp** subfolder of the transformation's parent folder. You can change them.

If you click **Compile** you can see the compilation progress as the grey **Stale** texts change to green **Ready** text.

1.4.1.5. Build

The **Interpreted** check box selects between interpreted execution (fast start, slow run) or code generated execution (slow start, fast run).

The interpreted compilation synthesizes a QVTc middle metamodel and a genmodel.

- **HierarchicalStateMachine2FlatStateMachine.ecore**
- **HierarchicalStateMachine2FlatStateMachine.genmodel**

The code generated compilation additionally synthesizes a Java class to implement the transformation and the Java classes for the QVTc middle model.

- **HierarchicalStateMachine2FlatStateMachine.java**

The generated files are generated to a distinct **src-gen** tree which you may need to refresh to see all files. Java problem markers come and go during the synthesis and build. The errors should all be gone before execution proceeds. Elimination of warnings is a work in progress.

Run

Clicking on **Run** will **Compile** automatically if you have not already done so.

The result should be a **flat.xmi** file that is similar to the **expected.xmi** file.

1.4.2. QVTc Example Project

The OMG QVTc variant of the traditional UML to RDBMS transformation may be installed by:

Invoke **File->New->Example...** then select **Examples** then **QVT (Query/View/Transformation Language) Projects** then select either **QVTc UML2RDBMS Project** then **Finish** to create a small example project called **org.eclipse.qvtd.examples.qvtcore.uml2rdbms**.

The QVTc editor opens to show the transformation.

The QVTc tooling is very similar to the QVTr tooling. The main difference is that for QVTc the middle model and its genmodel are designed by the user, whereas the QVTr middle model and genmodel are synthesized automatically. If code generated Java execution is required, using the genmodel to generate Java code for the middle model is again a manual user responsibility.

1.5. Extensions

1.5.1. Import

The Eclipse support for OCL, QVTc and QVTr adds an import statement to define the location of the metamodels.

```
import SimpleUML : 'SimpleUml.ecore'::umlMM;
```

This specifies that the **SimpleUml.ecore** metamodel is loaded. The metamodel root prefixes a navigation to select an imported element. In the example above the **umlMM** package is selected and made available with a **SimpleUML** alias. If alias-name and colon separator are omitted, the imported element is accessible by its own name.

The quoted name may be importing-file-relative file name as above, or a an absolute path such as **platform:/resource/org.eclipse.emf.examples.library/model/extlibrary.ecore** or a registered namespace URI such as **http://www.eclipse.org/emf/2002/Ecore**.

1.5.2. Packages

The Eclipse support for QVTc and QVTr supports declaration of transformations within a package hierarchy. A Transformation therefore extends just a Class, not a Class and a Package as specified by OMG. The Transformation is declared nested within one of more Packages by qualifying the transformation name with one of more package names. For compatibility, if no Package is declared, the Transformation is contained by a Package with a blank name.

The package hierarchy declared for the Transformation is re-used to define the Java package hierarchy when Java code is generated from the Transformation.

```
transformation org::eclipse::qvtd::xtext::qvtrelation::tests::hstm2fstm::HierarchicalStateM
{
    ...
}
```

Alternatively a package declaration may be used. This may also optionally define the package URI and/or namespace prefix.

```
package org::eclipse::qvtd::xtext::qvtrelation::tests::hstm2fstm : nsPrefix = 'nsURI'
{
    transformation HierarchicalStateMachine2FlatStateMachine(hier:hierMM, flat:flatMM)
    {
        ...
    }
}
```

1.5.3. Contextual Operations and Properties

A **package** declaration may contain nested packages, classes or transformations. Within class declarations operations and properties may be declared analoguously to QVTo’s contextual operatuions and properties, or to additional declarations from an imported Complete OCL document. The syntax is the same as OCLinECore which emulates typical OMG specification source text.

```
package MyPackage
{
    class MyClass
    {
        operation op(p : String[1]) : String[1]
        {
            body: p.toUpper();
        }
        property name : String[1];
    }
    transformation MyTx(...)
    {
        ...
    }
}
```

1.5.4. QVTc Middle Model

The QVTc part of the specification is unclear as to how the middle metamodel is defined.

Eclipse QVTc imports the middle metamodel in the same way as any other metamodel. This is then used as a used-package of an unnamed domain.

```
import SimpleUML : 'SimpleUml.ecore'::umlMM;
import SimpleUMLtoRDBMS : 'SimpleUMLtoRDBMS.ecore'::uml2rdbms;
import SimpleRDBMS : 'SimpleRdbms.ecore'::rdbmsMM;
```

```
transformation umlRdbms {
 uml imports SimpleUML;
 rdbms imports SimpleRDBMS;
 imports SimpleUMLtoRDBMS;
}
```

1.5.5. QVTr Middle Model

The QVTr part of the specification appears to specify how the middle metamodel is synthesized. The specification however ignores details such as Models, Packages and Collections. There is also a lack of clarity as to whether the trace is the minimum required to support non-re-invocation of mappings or whether it should contain everything necessary for minimal incremental re-execution. The Eclipse QVTd implementation is work-in-progress.

1.5.6. QVTr Collection Templates

The QVTr part of the specification omits almost all detail of the semantics of Collections and in particular Collection Templates. The implementation in Eclipse QVTd is therefore language design work-in-progress rather than implementation work-in-progress.

1.5.7. OCL/EMOF Metamodels

The QVT specification provides layered metamodels for QVTrelation, QVTtemplate, QVTcore and QVTbase layered on top of EssentialOCL and EMOF. The EssentialOCL and EMOF metamodels are very strongly influenced by OCL and MOF specifications, but are ot formally compliant.

Eclipse QVTd provides layered metamodels for QVTrelation, QVTtemplate, QVTcore and QVTbase layered on top of Pivot which is derived from the UML 2.5 metamodel and work towards a future OCL specification. (QVTimperative shares some QVTcore functionality through a QVTcoreBase abstraction.)

A consequence of extending the Eclipse OCL Pivot is that Eclipse QVTd has preliminary support for templated types, extensible libraries and safe navigation.

1.5.8. this

QVTo defines **this** as a reserved variable for the instance of the executing transformation.

Eclipse QVTd provides **this** for QVTc and QVTr and thereby solves an OCL compatibility problem with respect to the source of an operation call of a query. In Eclipse QVTd, queries, using the Function class, are operations of the Transformation class and are invoked with an implicit **this**. An explicit **this** can be used, and is used when viewing the equivalent source text for the Abstract Syntax model.

Chapter 2. pivot

2.1. OCLExpression

Associations

AddStatement : qvtimperative::AddStatement[?]

BufferStatement : qvtimperative::BufferStatement[?]

CheckStatement : qvtimperative::CheckStatement[?]

DeclareStatement : qvtimperative::DeclareStatement[?]

MappingLoop : qvtimperative::MappingLoop[?]

NewStatement : qvtimperative::NewStatement[?]

SetStatement : qvtimperative::SetStatement[?]

SimpleParameterBinding : qvtimperative::SimpleParameterBinding[?]

2.2. Property

Attributes

ObservableStatement : Bag(qvtimperative::ObservableStatement)

SetStatement : Bag(qvtimperative::SetStatement)

2.3. VariableDeclaration

Attributes

SetStatement : Bag(qvtimperative::SetStatement)

Chapter 3. `qvtimperative`

The Package for an Imperative QVT transformation.

An Imperative QVT trabsformation is expected to be created by an autogenerator that observes the following run-time restrictions:

A mapping that my be re-invoked must have MappingCall.isInfinite set for every possible invocation.

A mapping that reads object slots before they are guaranteed to have been assigned must declare the slots property in a corresponding ImperativeArea.checkedProperties entry.

A mapping that assigns an object slot that any mapping may access before assignment is guaranteed must declare declare the slots property in a corresponding ImperativeArea.enforcedProperties entry.

All reads by Functions/Queries must be guaranteed to succeed; i.e. the invoking mapping must check readiness before calling the query.

All writes to multi-valued properties must be guaranteed to occur before any read of the property.

3.1. `AddStatement`

An AddStatement appends the value of an expression to a connection.

syntax: `add connection := expression;`

conformsTo `MappingStatement` , `ObservableStatement`

Attributes

`isEnforcedUnique : Boolean[?]`

True if the append is suppressed after a check for uniqueness of the value.

Associations

`ownedExpression : OCLExpression[1]`

The expression whose evaluation yields the value to append.

`targetVariable : ::ConnectionVariable[1]`

The connection to be appended.

3.2. `AppendParameter`

An AppendParameter of a Mapping defines an output connection to which values may be appended.

syntax: `append name : type;`

conformsTo `ConnectionVariable` , `MappingParameter`

3.3. `AppendParameterBinding`

An AppendParameterBinding binds an AppendParameter of an invoked Mapping to a connection of the invoker. Execution of the mapping may append to the connection.

syntax: `formalName appendsTo connection;`

conformsTo `MappingParameterBinding`

Associations

`value : ::ConnectionVariable[1]`

The value or collection of values to bind to boundVariable

3.4. `BufferStatement`

A BufferStatement declares a connection buffer and optionally assigns initial content.

syntax: `buffer name : type := expression;`

`type` or `expression` but not both may be omitted. An omitted type is deduced from the initial expression values.

conformsTo `ConnectionVariable` , `VariableStatement` , `ObservableStatement`

Associations

`ownedExpression : OCLExpression[?]`

The optional expression computing initial content values.

3.5. CheckStatement

A CheckStatement evaluates a predicate. If the evaluation is false, the mapping execution terminates fails and does nothing.

syntax: `check expression;`

conformsTo `ObservableStatement`

Associations

`ownedExpression : OCLExpression[1]`

3.6. ConnectionVariable

A ConnectionVariable identifes a variable used as a connection buffer.

conformsTo `VariableDeclaration`

Attributes

`AddStatement : Bag(qvtimperative::AddStatement)`

`AppendParameterBinding : Bag(qvtimperative::AppendParameterBinding)`

`GuardParameterBinding : Bag(qvtimperative::GuardParameterBinding)`

3.7. DeclareStatement

A DeclareStatement declares a variable and initial value.

syntax: `check var name : type := expression;`

`type` or `expression` but not both may be omitted. An omitted type is deduced from the initial expression values.

`check` may be omitted when the expression type is necessarily conformant.

conformsTo `VariableStatement` , `ObservableStatement`

Attributes

`isCheck : Boolean[?]`

True if the initial expression's type must be checked for conformance with the variable's type. A non-conforming vlaue is a predicate failure causing the mapping to fail without doing anything. This is a derivation of not ownedInit.type.conformsTo(self.type).

Associations

`ownedExpression : OCLExpression[1]`

The expression computing the variable's value.

3.8. GuardParameter

A GuardParameter of a Mapping defines a input at which a value is consumed from a connection.

syntax: `guard:typedModel name : type;`

conformsTo `MappingParameter`

Associations

`referredTypedModel : ::ImperativeTypedModel[1]`

The TypedModel that contains the passed value.

3.9. GuardParameterBinding

A GuardParameterBinding binds a guard parameter of an invoked Mapping to a value of a connection. Execution of the mapping may use the value. A distinct Mapping invocation occurs for each value in the connection.

syntax: `formalName consumes expression;`

conformsTo `MappingParameterBinding`

Attributes

`isCheck : Boolean[?]`

True if each consumed value must be checked for conformance with the variable's type. A non-conforming vlaue is a predicate failure causing the mapping invocation to fail without doing anything. This is a derivation of not ownedInit.type.conformsTo(self.type).

Associations

`value : ::ConnectionVariable[1]`

The connection providing the invocation values.

3.10. ImperativeModel

The Model of an Imperative QVT transformation.

conformsTo `BaseModel`

3.11. ImperativeTransformation

An ImperativeTransfornmation distinguishes a QVTi transformation from other transformations.

conformsTo `Transformation`

3.12. ImperativeTypedModel

An ImperativeTypedModel defines an input,middle or output modek for the transformation.

conformsTo `TypedModel`

Attributes

`GuardParameter : Bag(qvtimperative::GuardParameter)`

`NewStatement : Bag(qvtimperative::NewStatement)`

`SimpleParameter : Bag(qvtimperative::SimpleParameter)`

`isChecked : Boolean[?]`

True for an input model.

`isEnforced : Boolean[?]`

True for an output model.

3.13. LoopParameterBinding

A LoopParameterBinding binds a guard parameter of an invoked Mapping to the value of a loop variable in the invoker. Execution of the mapping may use the value.

syntax: `formalName iterates expression;`

Deprecated – WIP for a consuming stream

conformsTo `MappingParameterBinding`

Attributes

`isCheck : Boolean[?]`

Whether the variable initialization needs to be checked as a predicate. This is a derivation of not ownedInit.type.conformsTo(self.type).

Associations

`value : ::LoopVariable[1]`

The value or collection of values to bind to boundVariable

3.14. LoopVariable

A LoopVariable defines the iterator of a MappingLoop.

conformsTo `VariableDeclaration`

Attributes

`LoopParameterBinding : Bag(qvtimperative::LoopParameterBinding)`

Associations

`owningMappingLoop : ::MappingLoop[1]`

3.15. Mapping

An Imperative Mapping extends the abstract declarative mapping to support explicit nested invocation of mappings with bindings for the invoked mapping’s bound variables.

conformsTo `Rule`

Attributes

`MappingCall : Bag(qvtimperative::MappingCall)`

`ownedParameters : Set(qvtimperative::MappingParameter)[*|1]`

`ownedStatements : OrderedSet(qvtimperative::Statement)[*|1]`

3.16. MappingCall

A MappingCall specifies the invocation of a referredMapping with a set of bindings. An installed mapping is invoked asynchronously whenever suitable values are available on consumed connections. An invoked mapping is invoked synchronously with values provided by the caller.

conformsTo `MappingStatement`, `ReferringElement`

Attributes

`binding : OrderedSet(qvtimperative::MappingParameterBinding)[*|1]`

The Set of bindings of variables or expressions to forma parameters.

`isInfinite : Boolean[?]`

An infinite MappingCall requires re-invocation of the called mapping to be suppressed to avoid an infinite loop.

Deprecated ?? not needed once install works.

`isInstall : Boolean[?]`

An install MappingCall declares a MappingCall that consumes one or more connections and appends to zero or more connections. Invocations of the mapping are driven by the availability of values in the connection.

`isInvoke : Boolean[?]`

An invoke MappingCall invokes a Mapping that uses one or more values and appends to zero or more connections. Invocations of the mapping is requested by the caller.

Associations

`referredMapping : ::Mapping[1]`

The Mapping invoked by the MappingCall.

3.17. MappingLoop

A MappingLoop supports an iteration of mapping invocations.

syntax: `for name : type in expression {...}`

conformsTo `MappingStatement`, `ObservableStatement`

Attributes

`ownedIterators : OrderedSet(qvtimperative::LoopVariable)[*|1]`

The iterator loop variable.

`ownedMappingStatements : OrderedSet(qvtimperative::MappingStatement)[*|1]`

The statements to be iterated, typically a single MappingCall.

Associations

`ownedExpression : OCLExpression[1]`

The expression whose values provide the iterator values.

3.18. MappingParameter

A MappingParameter defines a formal parameter of a mapping. The parameter is bound by the mapping invocation to satisfy the requirements of the derived parameter class.

conformsTo `VariableDeclaration`

Attributes

`MappingParameterBinding : Bag(qvtimperative::MappingParameterBinding)`

Associations

`Mapping : ::Mapping[?]`

3.19. MappingParameterBinding

A MappingCallBinding specifies the binding of a single variable or value to the formal parameter of a mapping as part of its inviocatuon or installation.

conformsTo `Element`

Associations

`boundVariable : ::MappingParameter[1]`

The formal parameter bound by the call.

`mappingCall : ::MappingCall[?]`

The containing MappingCall.

3.20. MappingStatement

A MappingCall specifies the invocation of a referredMapping with a set of bindings of the bound variables of the referredMapping to values provided in the invocation. Where Collections of values are provided for isLoop bindings, a distinct invocation is performed for each distinct permutation of Collection elements.

conformsTo `Statement`

Associations

`MappingLoop : ::MappingLoop[?]`

3.21. NewStatement

A NewStatement creates an instance of a class and binds a name to it.

syntax: `new:typedModel name : type := expression;`

If expression is omitted, a new instance if the tyope is created. If expression is provided, it computes the ‘new’ object, typically a singlton supervisor fpr a QVTr key.

conformsTo `VariableStatement`, `ObservableStatement`

Associations

`ownedExpression : OCLExpression[?]`

Optional expression that constructs the new object.

`referredTypedModel : ::ImperativeTypedModel[1]`

The TypedModel to which the new object is added.

3.22. ObservableStatement

An ObservableStatement may involve evaluation of an expression that accesses object properties whose values may not be available. If not ready,the mapping execution is suspended until the required value is made available by a notifying SetStatement.

syntax: `observe class::property ...`

conformsTo [Statement]

Attributes

`observedProperties : Set(Property)[*|1]`

The properties whose accesses must be checked for readiness.

3.23. SetStatement

A SetStatement sets an object property to a computed value.

syntax: `notify set name : type := expression;`

If `notify` is specified, execution defines the property as ready enabling mappings whose ObservableStatements are waiting for the value to resume.

conformsTo [ObservableStatement]

Attributes

`isNotify : Boolean[?]`

`isOpposite : Boolean[?]`

Associations

`ownedExpression : OCLExpression[1]`

`targetProperty : Property[1]`

`targetVariable : VariableDeclaration[1]`

3.24. SimpleParameter

A SimpleParameter of a Mapping defines an input at which a value is passed to the mapping.

syntax: `in:typedModel name : type;`

conformsTo [MappingParameter]

Associations

`referredTypedModel : ::ImperativeTypedModel[1]`

The TypedModel that contains the passed value.

3.25. SimpleParameterBinding

A SimpleParameterBinding binds a simple parameter of an invoked Mapping to the value of an expression computed by the invoker. Execution of the mapping may use the value.

syntax: `formalName uses expression;`

conformsTo [MappingParameterBinding]

Attributes

`isCheck : Boolean[?]`

Whether the variable initialization needs to be checked as a predicate. This is a derivation of not ownedInit.type.conformsTo(self.type).

Associations

`value : OCLExpression[1]`

The value or collection of values to bind to boundVariable

3.26. Statement

A Statement is the basis for all execution by a Mapping.

conformsTo `NamedElement`

Associations

`Mapping : ::Mapping[?]`

3.27. VariableStatement

A VariableStatement is the basis for a Mapping execution that makes a name available to subsequent starements.

conformsTo `VariableDeclaration`, `Statement`

Appendix A. Glossary

EMF	Eclipse Modeling Framework
OCL	Object Constraint Language
OMG	Object Management Group
QVT	Query/View/Transformation
RTF	Revision Task Force