SINTEF

SINTEF REPORT

SINTEF ICT

Address: NO-7465 Trondheim, NORWAY
Location: Forskningsveien 1
Telephone: +47 22 06 73 00
Fax: +47 22 06 73 50

Enterprise No.: NO 948 007 029 MVA

TITLE

Evaluation of the QVT Merge Language Proposal

AUTHOR(S)

Roy Grønmo, Jan Aagedal, Arnor Solberg (SINTEF), Mariano Belaunde (France Telecom), Peter Rosenthal (INRIA), Madeleine Faugere (Thales), Tom Ritter, Marc Born (Fraunhofer)

CLIENT(S)

EU Commission

REPORT NO. STF90 A05045	CLASSIFICATION Open	CLIENTS REF. IST-511731	
CLASS. THIS PAGE Open	ISBN 82-14-03659-3	PROJECT NO. 90B209	NO. OF PAGES/APPENDICES 85
ELECTRONIC FILE CODE Document2		PROJECT MANAGER (NAME, SIGN.) Jan Øyvind Aagedal [illegible]	CHECKED BY (NAME, SIGN.) Jon Oldevik Jon Oldevik
FILE CODE	DATE 2005-03-31	APPROVED BY (NAME, POSITION, SIGN.) Bjørn Skjellaug, Research Director	

ABSTRACT

This report has identified 29 weighted evaluation criteria representing desired properties of a model to model transformation language. These criteria have then been used to evaluate the current QVT Merge specification. We have so far only been able to evaluate 21 of these criteria, mainly due to missing tool support. Some of the criteria are considered absolute in the sense that missing to fulfill such a criterion is considered a failure. The 21 evaluated criteria give a score of 59 out of a maximum possible score of 68 (language-based + example-based testing). We have also compared the QVT-Merge submission with the QVT-Compuware/Sun submission and at the time being the QVT-Merge seems to be the preferred one due to more support on the absolute criteria and better easy-to-use score.

Eight transformation examples for solving six different transformation tasks have given a lot insight on the ease of use criteria for both simple and complex transformations.

When defining transformations using QVT Merge we believe that a lot of effort may be required in order to define the source and target metamodels.

The evaluation in this report could be improved by using the reference examples with alternative approaches published in the literature. An available QVT-Merge tool is necessary in order to provide evaluations of all the suggested criteria. In order to further investigate the usability of the graphical notation, we need to define more of the transformation examples graphically. Only one of the examples has been specified graphically in this version. The current evaluation has been done by a single evaluator who has only reviewed the transformation code that was written by somebody else. The evaluation will be further improved by incorporating input from other evaluators as well as evaluation from those who wrote the transformation code.

KEYWORDS	ENGLISH	NORWEGIAN
GROUP 1	ICT	IKT
GROUP 2	Information systems	Informasjonssystemer
SELECTED BY AUTHOR	Model transformation	Modelltransformasjon
	Evaluation	Evaluering

MODELWARE

IST Project 511731

MODELling solution for softWARE systems

Evaluation of the QVT Merge Language Proposal

Document Reference: Modelware/I1.2.1

Document Version:	1.0	
Document Date:	Preparation	31st of March 2005

Dissemination level: PU ☒ PP ☐ RE ☐ CO ☐

Author(s): Roy Grønmo, Jan Aagedal, Arnor Solberg (SINTEF), Mariano Belaunde (France Telecom), Peter Rosenthal (INRIA), Madeleine Faugere (Thales), Tom Ritter, Marc Born (Fraunhofer)

Project funded by the European Community under the "Information Society Technologies" Sixth Framework Programme

Table of contents

1. Introduction

An overall objective of MODELWARE is to improve productivity in software development. This objective will be pursued by realising the vision of model-driven software development. To this end, model transformation is viewed as a crucial technology. Model transformation makes it possible to derive models from other models in a controlled and automatable manner. It also simplifies the way one relate models, for instance to ensure consistency. OMG is currently finalising a standard that defines how one should specify model to model transformations of MOF models. In this report we evaluate this forthcoming standard.

The purpose of this evaluation is to investigate to what extent the current QVT Merge approach [1] – which is likely to become an OMG adopted standard in 2005 - is able to fulfil the requirements criteria and expectations on model to model transformations expressed by the MODELWARE partners. In addition, this evaluation may provide valuable feedback for any needed improvement to the QVT submission team. This report also includes a quick evaluation of the competing QVT Compuware/Sun approach [2].

This report selects a number of evaluation criteria, defines the evaluation method and then performs the evaluation on concrete examples. We have chosen examples that other approaches claim to handle well and reformulate them according to the QVT Merge proposal, in order to see whether QVT Merge is able to handle what other approaches claim they can. The summary section provides an overall view of the results of the evaluation, and provides some recommendations.

MODELWARE is a project co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2002-2006). Information included in this document reflects only the author's views. The European Community is not liable for any use that may be made of the information contained herein.

2. Deriving evaluation criteria

In this section a set of requirements for model-transformations are identified. These are analyzed in order to derive our evaluation criteria, which is described in the next section. The Modelware WP description (what we aim to achieve in this WP) has been our guide to the requirements elicitation. In order to come of with relevant requirements we have conducted a survey of relevant literature (in particular [3-5]).

The identified requirements serves as the rationale for the selected evaluation criteria and its given weight value. Each evaluation criteria may be traced to one or more of the identified requirements

Traditional use case techniques are used for requirements elicitation. Thus the requirements identified are user-driven and is thereby specified at the operational level (the level where model-transformations are performed). Following the use case approach a set of actors are identified and their needs are derived. These are depicted in Figure 1 and Figure 2.

Three main actors have been identified:

- QVT actors specifying the model transformation assets,
- QVT actors implementing the model transformation assets,
- QVT actors using the model transformation asset to create and update models

2.1. QVT actors realising the model transformation assets

Actor	Role
QVT Designer	Its role is to design QVT rules of a determined set of user requirements. QVT Designer tasks are: • Elaborate the source and target metamodels, • Define source to target mapping ❑ Reuse already defined rules if necessary ❑ Specialize existing rules ❑ Create nested rules ❑ Apply transformation patterns ❑ Integrate non-functional constraints properties • Define package organisation for model transformation rules in order to improve reuse and business knowledge capitalization • Precise rule properties like constraints or order • Test / check rule design completeness • Document the rule design to ease maintenance

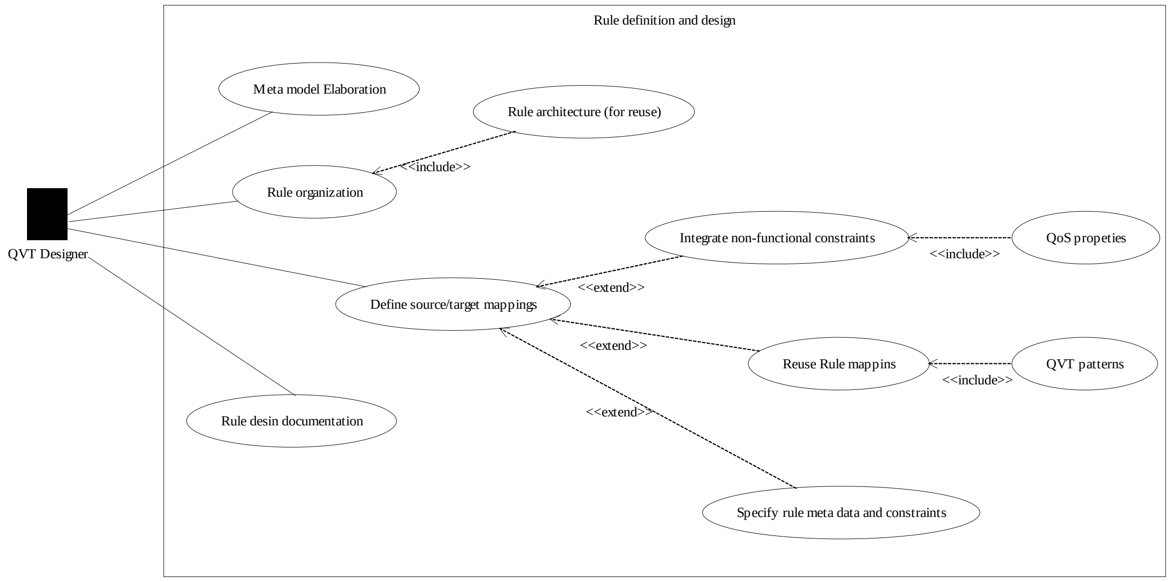


Figure 1 Rule definition and design

For this activity, the transformation language shall be easy to learn and intuitive use. Rules design can be described in a textual or graphical form, and like traditional software design must provide reuse, specialization, and capitalization capabilities and shall cover all model aspect (static, dynamic, non functional aspect…). Transformation rules can require several input models, and can produce several target models.

2.2. QVT actors implementing the model transformation assets

Actor	Role
QVT Developers	Its role is to code QVT rules. He has to • Implement QVT rules according rule design ❑ Use existing libraries and patterns ❑ Use software programming techniques • Define rule implementation architecture and modules • Test the implementation according the requirements • Create the model transformation modules (artefacts) if necessary

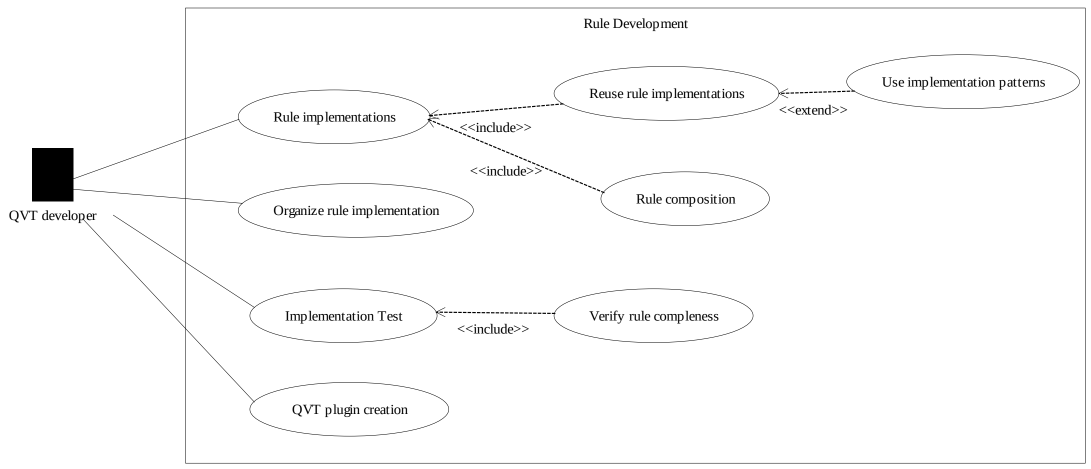
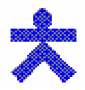


Figure 2 Rule development

2.3. QVT actors applying model transformation assets

Actor	Role
 **QVT User**	Its role is to apply QVT transformation assets on models. He has to • Creates models • Apply transformation ❑ make choices • Modify models and propagates modification over different abstraction layers.

3. Evaluation Criteria

Each criterion is defined using these properties:

- "Name" uniquely identifies the criterion.
- "Description" describes what the criterion is.
- "Absolute" defines whether the criteria must be present or whether it is optional. A criterion is satisfying the absolute requirement if there is some level of support, such as the lowest level. An approach that does not meet one of its absolute criteria is considered useless. Notice that an absolute criterion which has a scale with only two outcomes (No support / support) has irrelevant weight as all the useful approaches will get the same score.
- "Scale" defines the measurement scale (Examples: Support / No support or {0,1,2,3} where 0 means no support, 1 means…). Increasing values are always better unless stated otherwise. Thus the optimal scale measurement is the highest achievable value.
- "Weight" defines how important we consider the criteria to be (1 = lowest importance, 6 = highest importance).
- "Weight and Absolute judgment" describes why we have assigned the weight and choice of absolute vs. optional.

The aim is to ensure that each criterion has a proper rationale, that there are no overlapping criteria and that they are easy to measure. Furthermore the criteria should be such that it will differentiate different tools/languages. The desired criterion is poorly specified if all tools and languages automatically will support the property.

The criteria are sorted in three categories:

- **Tool-dependant**. These criteria can only be evaluated if the transformation language is implemented in a specific tool.
- **Inherent language properties.** These criteria can be evaluated entirely based on the definition of the language. The outcome of the evaluation will not vary with the kind of source or transformation examples used.
- **Example-dependant.** The evaluation of these criteria depends on the kind of source or transformation examples used.

The categorization between tool-dependant and inherent language properties may be difficult since in most cases a tool may add more support than is built into the language. We suggest that the property in such cases should belong primarily to the inherent language properties as this will make the approach less tool-dependant. For some of the properties it is also specified that they belong to another category as a secondary criterion. All the criteria are expressed as desired properties. The measurement on the scale determines to what extent the desired property is fulfilled. Tool-dependant properties are not measured as there are no current tool implementations of the latest QVT-Merge specification. The inherent language properties are measured by manual inspection of the specification, while the example-dependant properties are measured by manual inspection of proposed QVT transformation code for specific examples.

3.1. Tool-Dependant Properties

3.1.1. Incomplete transformations completed by human intervention

Rationale: In some cases it will be desirable to parameterize the transformation with user input. This is an additional need to the parameterization of the reusability property due to convenience or due to knowledge that is not present in design time.

Scale details: 0 = No support. 1 = Human input is possible, but it is not stored, re-applied in repeated transformations or modifiable in consecutive transformations. 2 = Full support for human input. It is stored, re-applied or modified depending on the user's choice in consecutive transformations.

Weight and Absolute judgment: The weight is low since it will not be needed in most use cases. It is absolute since it needs to be supported in those cases where it is needed.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Incomplete transformations completed by human intervention	Be able to execute incomplete/in-deterministic transformations that may require external input to process the transformation. The re-applying of the transformation will take the human input. Human input shall be stored. Human input shall be modifiable.	Yes	{0,1,2}	2

3.1.2. Transactional transformation

Rationale: This is needed in transformation compositions where the failure of one transformation implies that all the other state prior to all the transformations should be recovered. This is equivalent to transactions and recovery in databases.

Scale details: 0=No support, 1 = ACID transactions, 2 = distributed transactions, 3= nested distributed transactions.

Weight and Absolute judgment: The weight is low since there are many typical use cases where transactional transformation is not needed. It is absolute since it in some use cases is critical.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Transactional transformation	Nested transaction (useful by rule composition) shall be possible.	Yes	{0,1,2,3}	2

3.1.3. Conservative transformation

Rationale: This will enable model lifecycle support so that the model may be generated, manual changes may be made and then at a later stage a re-generation will not override the manual changes unless the user gives permission to do so.

Scale details: 0 = No support. 1 = Some kind of manual marking is done to ensure that certain parts of the target model shall not be overwritten in a re-generation. 2 = All manual changes to the target model will be discovered and not overwritten. 3 = All manual changes are discovered and the user is consulted about which parts to keep and which to overwrite. 4 = Level 3 support + the automatic support of updating the source model if the user chooses this feature.

Weight and Absolute judgment: The weight is low since there are many typical use cases where it is not needed to do manual changes. It is absolute since it is critical in cases where one needs to re-generate after manual changes.

Name	Description	Absolute	Scale	Weight

		(Yes/No)		(1..6)
Conservative transformation	Be able to re-apply a transformation without loosing manually target model upgrade.	Yes	{0,1,2,3,4}	2

3.1.4. Performance/Scalability

Rationale: This property is given a low weight since most transformations will probably be executed prior to run-time meaning that time is not very critical. But it is still desirable that the large transformations are executed within an acceptable time period.

Scale details: The scale is not determined for this property. One needs to define some reference examples with transformation definitions , source models and corresponding acceptable execution time.

Weight and Absolute judgment: The weight is medium because a transformation execution is not a time-critical where one desires optimal performance. Acceptable performance is good enough. The weight is not low since it must be able to handle complex transformations in reasonable time. It is not absolute since the boundary of acceptable waiting time is unclear.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Performance/Scalability	Be able to specify huge transformations and perform them for large models in reasonable time.	No		3

3.1.5. Control of execution process

Rationale: It may be more convenient for the transformation user to be able to specify all the user information at once, so that it is not needed to watch the transformation in its complete execution period.

Scale details: Support / No support

Weight and Absolute judgment: The weight is very low since this is only a convenience property and which is not even applicable in many typical use cases.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Control of execution process	During a transformation, avoid spread user information request	No	Support / No support	1

3.1.6. Ability to debug the transformation

Rationale: This is desired so that the transformation architect more easily can track down errors or ensure that the transformation does what it is supposed to do.

Scale details: Support / No support

Weight and Absolute judgment: The weight is high since debugging facilities can greatly improve the working conditions for the transformation architect.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)

Ability to debug the transformation	The language is associated with an environment to debug rule transformation	Yes	Support / No support	5

3.1.7. Ability to check rule consistency

Rationale: This is desired so that a transformation can be validated prior to being executed. Thus it will be easier to capture errors at an early stage.

Scale details: Support / No support

Weight and Absolute judgment: The weight is high since this ability is expected to greatly decrease the time needed to define and maintain transformations. It is not absolute since many other desired properties can fulfill some of the needs.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Ability to check rule consistency	The language is associated with an environment to check rule consistency	No	Support / No support	5

3.1.8. Secondary properties of tool-dependant properties

These properties are defined as belonging to a different category, but do also have some secondary relevance in this category: <none>.

3.2. Inherent Language Properties

3.2.1. Traceability

Rationale: This property will make it easier for the transformation architect to understand how changes in the source will affect the target. It is very useful when managing operation on models like impact analysis, for instance if an element is deleted then other depending elements may need to be deleted and this could be reported by the traceability mechanism. It is also useful when undesired target results are produced as the tracing back to the source element will be of important help in order to correct the source model or definition of the transformation.

Scale details: 0 = No support, 1 = Manual support. The user must explicitly express the elements to be traced. 2 = Automatic support. The tool automatically provides traceability of all the elements.

Weight and Absolute judgment: The weight is high and absolute since this property is essential in order to understand and maintain the transformations.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Traceability	Be able to relate modelling elements between source and target. (A) Be able to retrieve without ambiguity the elements created in the target model. (B)	Yes	{0,1,2}	5

3.2.2. Bidirectionality

Rationale: It will be easier for the transformation architect to define one bidirectional transformation than to define two separate transformations for this purpose. The maintenance of a single transformation definition will also be easier to maintain and it reduces the risk of errors.

Scale details: Support / No support.

Weight and Absolute judgment: The weight is low since it is possible to achieve the same transformation with two separate transformations and many transformations will not be bidirectional.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Bidirectionality	Whether the transformation rules are executable in both directions (from source and target).	No	Support / No support	2

3.2.3. QoS Mapping

Rationale: QoS specifications should be preserved in transformations. This means that source model elements with QoS specifications linked to them should produce source model elements with corresponding QoS specifications associated. Such a transformation could either see to it that the QoS specifications are carried forward through the transformation or that they are refined into other QoS specifications.

Scale details: Support / No support.

Weight and Absolute judgment: The weight is medium since many transformations will not care about QoS mapping. It is absolute since it is critical for those transformations where it is needed.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
QoS Mapping	Be able to preserve QoS specifications in transformations.	Yes	Support / No support	3

3.2.4. Resolution of QoS properties

Rationale: QoS requirements at one level of abstraction may correspond to a functional requirement/solution at a lower level of abstraction. It should be possible to specify and trace such a transformation.

Scale details: Support / No support.

Weight and Absolute judgment: The weight is low since many transformations will not care about QoS mapping and it is partly overlapping with traceability. It is absolute since it is critical for those transformations where it is needed.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Resolution of QoS properties	Be able to transform QoS properties into behaviour and trace these transformation	Yes	Support / No support	2

3.2.5. Composition of transformations

Rationale: This is desired in order to reuse several basic transformations to accomplish a more complex task.

Scale details: 0 = No support. 1 = Sequence only. 2 = Supporting the five basic control flow patterns (http://tmitwww.tm.tue.nl/research/patterns/patterns.htm)

Weight and Absolute judgment: The weight is medium since there are many typical cases where composition is not needed. It is absolute since leaving it to the user to handle the transformation composition execution will be too error-prone and tedious.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Composition of transformation	Be able to compose transformations (in sequence or parallel). Create transformation rules by sequencing of already defined transformation rules (example Query rule composition)	Yes	{0,1,2}	3

3.2.6. Constraints between rules

Rationale: This is desired in order to capture dependencies between rules. These dependencies rules can be realized by rule pre/post condition definition or rule ordering.

Scale details: Support / No support

Weight and Absolute judgment: The weight is very low since this capability is partly overlapping with the restricting conditions/pre-conditions property.

Name	Description	Absolute (Y/N)	Scale	Weight (1..6)
Constraints between rules	Be able to specify constraints between rules.	No	Support / No support	1

3.2.7. Multiple source models

Rationale: This is important since the input from more than one source model may be necessary in order to produce the target.

Scale details: Support / No support

Weight and Absolute judgment: The weight is low since many typical use cases do not need this property. For those where it is needed it can probably be achieved by consecutive transformations taking one source model at a time.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Multiple source models	Be able to have more than one source model	No	Support / No support	2

3.2.8. Multiple target models

Rationale: This is useful since there may be cases where it is desired to produce more than one target model.

Scale details: Support / No support

Weight and Absolute judgment: The weight is very low since many typical use cases do not need this property. If not desired it may still be achieved by defining several transformations, each operating on the source model(s) and producing one target each.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Multiple target models	Be able to have more than one source model with a single transformation definition	No	Support / No support	1

3.2.9. Updating source model(s)

Rationale: In some cases it is desired to update/complete an existing model instead of producing a new model (source and target models are the same).

Scale details: Support / No support

Weight and Absolute judgment: The weight is low since this is not needed in many typical use cases. It is absolute since it is critical in the cases where it is needed.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Updating source model(s)	Be able to define transformation rules to update the source model(s)	Yes	Support / No support	2

3.2.10. Reusability

Rationale: It is desirable to define transformations that capture common transformation rules that can be reused by other more specialized or parameterized transformations. This will improve the ability to share common knowledge, the ability to faster make new transformations and the ability to maintain the transformations.

Scale details: 0 = No support. 1 point for each of these that are satisfied: a) can import transformation library b) can specialize transformations. Maximum score is 2. To avoid overlapping with another desired property (3.2.11) specialization does not include parameterization. The specialization part property is overlapping with the inheritance part property of Object orientation, but it still seems relevant as there may also be other ways of specializing a transformation. If transformation inheritance is supported then one point will be given both in the Reusability and the Object orientation criteria.

Weight and Absolute judgment: The weight is high since this property is believed to improve faster development and maintenance and it is applicable to most transformations except for the simplest ones. It is not absolute since it is overlapping with other properties and thus many of the needs may be covered by them.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)

Reusability	Be able to reuse transformations by: • Specialization of transformations, • Importing of transformation library.	No	{0,1,2}	5

3.2.11. Incomplete transformations completed with pattern parameters

Rationale: This is a powerful construction to reuse large parts of a transformation that otherwise needs to be copied into several transformations.

Scale details: Support / No support

Weight and Absolute judgment: The weight is low since this is just one of many properties that can enable reuse of code.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Incomplete transformations completed with pattern parameters	This is the ability to use transformation patterns that can be parameterised/instantiated into complete transformations.	Yes	Support / No support	2

3.2.12. Repetitiveness

Rationale: This ensures that the transformations are defined precisely and unambiguously. There will be one and only one target result for a given source.

Scale details: Support / No support. It is not supported if and only if the transformation language allows non-deterministic or ambiguous constructions.

Weight and Absolute judgment: The weight is the highest since this is a fundamental property of the language in order to be used as an unambiguous transformation language.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Repetitiveness	Whether it is possible to ensure that same transformation on same source gives same result.	Yes	Support / No support	6

3.2.13. Restricting conditions/pre-conditions

Rationale: This is useful to ensure that the source model(s) provided to the transformation follows the restrictions set by the transformation. It prevents the transformation from being used incorrectly and provides the opportunity to give critical feedback to the transformation user. It is partly overlapping with the constraints between rules criterion.

Scale details: Support / No support.

Weight and Absolute judgment: The weight is high since this can detect errors in the usage of a transformation and it is needed in every transformation. It is optional since the validity check of the source model(s) also can be checked in external tools such as a modeling tool.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Restricting conditions/pre-conditions	Whether it is possible to restrict the applicability of a rule depending on conditions. For instance that the source model follows some-UML-profile	No	Support / No support	4

3.2.14. Black-box interoperability

Rationale: This enables the reuse of any existing codes or scripts that otherwise would need to be rewritten in the QVT language.

Scale details: Support / No support. Support requires that it is possible to specify references to external code within a QVT transformation.

Weight and Absolute judgment: The weight is high since this will make it faster to develop new transformations by reusing parts from legacy code, and reduces errors since we do not need to redefine existing transformation code that is tested to be OK.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Black-box interoperability	Whether it is possible, within a QVT transformation, to make usage of a transformation component that has been specified or implemented using another technology.	Yes	Support / No support	4

3.2.15. Unidirectionality

Rationale: When we never need to apply the reverse transformation it will be easier to concentrate only on the transformation one-way.

Scale details: Support / No support.

Weight and Absolute judgment: The weight is high since it is expected that more than 50% of the transformation use cases are unidirectional. It is absolute since it will make unidirectional transformation specifications unnecessarily complex if not satisfied.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Unidirectionality	Whether it is possible to concentrate on the transformation in one direction without any need to solve any issue regarding the potential opposite direction.	Yes	Support / No support	4

3.2.16. Modularity

Rationale: This will ease the comprehension and development of transformations.

Scale details: Support / No support. Support for this includes the possibility to split a transformation into several files, structure the code in separate UML package, provide separate transformation rules or to group methods inside classes, thus achieving fine grain modularity.

Weight and Absolute judgment: The weight is the highest since this will enable one to define structured and maintainable code. It is not absolute since it is overlapping with some of the other properties.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Modularity	Whether it is possible to structure the transformation rules in distinct pieces to ease comprehensibility of the specification.	No	Support / No support	6

3.2.17. Object orientation

Rationale: The principles of object orientation (OO) will improve the reuse, maintenance and comprehension of transformations.

Scale details: 0 = No support. 1 point for each of these four OO principles that are satisfied: a) inheritance b) encapsulation c) identity/ instantiation d) late binding/ polymorphism.

Weight and Absolute judgment: The weight is medium since there is a chance that other programming paradigms are equally or better suited for defining transformation specifications.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Object orientation	Whether common object oriented design techniques are applicable to the design of a transformation	No	{0,1,2,3, 4}	3

3.2.18. Availability of complete textual notation

Rationale: Textual notation enables users to define transformations without a graphical tool. Textual notations are also often preferred for defining large, complex transformations since graphical approaches are hard to scale.

Scale details: Support / No support

Weight and Absolute judgment: The weight is high and it is absolute because we assume that a textual notation is essential to properly handle large and complex transformations.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Availability of complete textual syntax	Ability to specify a transformation completely using a textual syntax	Yes	Support / No support	4

3.2.19. Presentation using graphical notation

Rationale: Graphical notations provide a higher-level view on the transformation and can more easily be communicated than a pure lexical alternative.

Scale details: 0 = No support. 1 = Only parts of a transformation can be graphical. 2 = A single transformation can be fully defined graphically. 3 = Compositions of transformations (see separate property) as well as single transformations can be fully defined graphically.

Weight and Absolute judgment: The weight is low since we doubt that a graphical notation scales well enough to handle complex transformations. It is absolute since it provides a higher level "transformation model" view of the transformation than the lower level textual notation.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Presentation using graphical notation	Can transformations or parts of them visualized in a graphical format? For example by connecting elements of the participating meta-models?	Yes	{0,1,2,3}	2

3.2.20. Learning Curve

Rationale: This property is desired since it increases the chance of becoming widely adopted.

Scale details: Measured as an answer to the question: Is the transformation language easy to learn?

0 = Strongly disagree. 1 = Disagree. 2 = Neither. 3 = Agree. 4 = Strongly agree

Weight and Absolute judgment: The weight is low, since it should not stop the introduction of a new way of programming style that has major advantages but that is unfamiliar to most people.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Learning Curve	How difficult is it to learn the language, How many skilled people are available?	No	{0,1,2,3,4}	2

3.2.21. Secondary properties of inherent language properties.

These properties are defined as belonging to a different category, but do also have some secondary relevance in this category:

- Transactional transformation. It may also be relevant to specify which transformations or parts of a transformation that should be executed as a nested transaction.
- Performance/Scalability. Is it possible to reason about the language having constructions that will enable it to be fast/slow?

3.3. Example-Dependant Properties

The criteria in this section require some case studies on reference transformation examples in order to be answered properly since language inspection or tool testing alone will not be able to provide a complete measurement.

3.3.1. Ease of use in simple transformations

Rationale: This property is highly desirable in order to increase productivity and adoptability of a transformation language.

Scale details: Measured as an answer to the question: Is the transformation language easy to use?

0 = Strongly disagree. 1 = Disagree. 2 = Neither. 3 = Agree. 4 = Strongly agree

Important sub-questions that are useful to answer the main question: Is the transformation language clear and understandable? It does not require a lot of mental effort to set up the transformation? It is easy to use the language to define transformations? It is not cumbersome to use? Is it frustrating to use? Is it controllable? Is it flexible? Is it easy to remember?

Weight and Absolute judgment: The weight is the highest since this is considered most crucial to the usability and adoption of the language in the transformation community. It is not absolute since there is not a clear border on what is an acceptable scale measurement.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Ease of use in simple transformations	How much effort is needed to solve a simple transformation problem? User friendly for the transformation engineer (e.g. Compact, easy to understand what the transformation is doing etc.). How easy it is to: to define rules, to manage rule consistency, to follow rule execution (for debug).	No	{0,1,2,3,4}	6

3.3.2. Ease of use in complex and large transformations

Rationale: This property is highly desirable in order to increase productivity and adoptability of a transformation language.

Scale details: Measured as an answer to the question: Is the transformation language easy to use?

0 = Strongly disagree. 1 = Disagree. 2 = Neither. 3 = Agree. 4 = Strongly agree

Important sub-questions that are useful to answer the main question: Is the transformation language clear and understandable? It does not require a lot of mental effort to set up the transformation? It is easy to use the language to define transformations? It is not cumbersome to use? Is it frustrating to use? Is it controllable? Is it flexible? Is it easy to remember?

Weight and Absolute judgment: The weight is the highest since this is considered most crucial to the usability and adoption of the language in the transformation community. It is not absolute since there is not a clear border on what is an acceptable scale measurement.

Name	Description	Absolute (Yes/No)	Scale	Weight (1..6)
Ease of use in complex and large transformations	How much effort is needed to solve a complex and large transformation problem? User friendly for the transformation engineer (e.g. Compact, easy to understand what the transformation is doing etc.). How easy it is to: to define rules, to manage rule consistency, to follow rule execution (for debug).	No	{0,1,2,3,4}	6

3.3.3. Secondary properties of example-dependant properties

These properties are defined as belonging to a different category, but do also have some secondary relevance in this category:

- Performance/Scalability. This criterion can only be tested against some reference examples with a given source model(s) and a given transformation.

4. Evaluation Method

The tool-dependant properties are not considered since there are no QVT tools yet. The language-dependant properties have been determined independently of the examples. This leaves us only with two important criteria that we need examples to evaluate: ease of use in simple/complex transformations. These two criteria are also given the maximum weight of 6 as they are particularly important for the wide-spread adoption and usage of QVT.

The evaluation method is basically driven by concrete examples. Some are very simple transformations while others are more complex transformations. Some of the examples are examples that were originally described using other transformation approaches. All the examples are then defined using QVTMerge to see if it is suitable for defining the transformations. One person defines the QVT transformation and the QVT expert, Mariano Belaunde, has reviewed the transformation to ensure that QVT is used in the best manner. A new person, not involved in defining the transformation will then inspect the transformation example and its QVT code and evaluate the ease of use criterion for this example. The evaluator is not a QVT expert, has not been involved in the QVT process or defined any QVT transformations, but has read the latest QVT Merge specification. The evaluator is an experienced programmer, but has only short experience with OCL expressions. Since none of the examples have been validated in a syntax parser or any other QVT-compliant tool, there may be errors in the code examples. It is decided if the example is considered simple or complex.

The examples cover both horizontal and vertical transformations, and cover both structural and behavioural models as shown in the table below:

Table 1 Categories of transformation examples

	Vertical	**Horizontal**
Structural	EJB/UML → EJB/Java AND UML → RDBMS	Spem UML Profile → Spem metamodel AND Book → Publication
Behavioural	EDOC → J2EE	XSLT → XQuery

The average score for the examples is used for the example-dependant criteria. All the scores of the criteria are used to compute an overall score by an algorithm described by the following pseudo-code:

1. Assign the *scale-value* 0 to "No support" values.
2. Assign the *scale-value* 1 to "Support" values.
3. If any of the absolute properties achieves a zero score then the total score is assigned to 0. If all the absolute properties are supported, then continue:
4. For each property assign

   property-value = (*scale-value* / (*max-scale-value* for this property)) * *property-weight*
5. *Total-score* = The sum of all *property-values*

5. QVT Merge Language-Based Evaluation

In the table below the M (M=measured-scale-level) column shows the level of support and the S (S=score) column shows the weighted score for the criterion. The values in parentheses show the maximum value. Note that the level of support is downscaled to a value between 0 and 1 (0= no support, 1 = full support) by dividing by maximum scale level, which ensures that the criteria are treated on equal scales before the weights are applied. A final score is computed by adding all the values in the S column.

The criteria that can be evaluated by manual inspection of the language itself and that does not depend on a tool or on observation in examples are presented in the following table.

Table 2 Evaluation of QVTMerge language-dependant properties

Criterion	How it is supported by QVTMerge	M	S
Traceability	Fully automatic traceability is achieved by the four resolve operations that can trace from any source object to any target object and vice versa.	2 (2)	5 (5)
Unidirectionality	The language in textual as well as graphical notation directly supports it.	1 (1)	4 (4)
Complete textual notation	Any transformation can be fully defined with the mappings part in textual notation.	1 (1)	4 (4)
Black-box interoperability	A query operation, a mapping rule and transformation module may be declared without a body definition. This means that the implementation will be provided externally - for instance using Java.	1 (1)	4 (4)
Composition of transformations	QVTMerge does not get maximum score of 2 due to the lack of possibility to specify parallel control flows.	1 (2)	1.5 (3)
QoS mapping	Source and target can be expressed as MOF models and we believe that QVT can be used to transform between any two pairs of MOF models.	1 (1)	3 (3)
Graphical notation	The maximum score of 3 is not achieved due to lack of graphically specifying compositions such as “parallel split” and “synchronization” which is not possible at all. It is assumed that single transformations can be defined fully graphically although the specification states that in some complex transformations OCL annotations are needed.	2 (3)	1.3 (2)
Updating source model(s)	The transformation signature allows input parameters which can be specified as `inout`.	1 (1)	2 (2)
Resolution of QoS properties	Source and target can be expressed as MOF models and we believe that QVT can be used to transform between any two pairs of MOF models.	1 (1)	2 (2)

Incomplete transformations completed with pattern parameters	QVTMerge/Mappings: A mapping may extend "abstract" incomplete mappings. QVTMerge/Relations: An abstract or checkable relation can be extended into executable transformations.	1 (1)	2 (2)
Modularity	The transformation may be grouped into several separate transformation rules.	1 (1)	6 (6)
Reusability	One point is given for the import module construction that enables one to import other libraries, and one point is given for the ability to specialize transformations by the extension mechanisms `extends`, `merges` and `inherits`.	2 (2)	5 (5)
Restricting conditions/pre-conditions	This is supported by associating the source model with a `modelType` with `complianceKind` = “strict”.	1 (1)	4 (4)
Object orientation	Inheritance is supported by the three extension mechanisms `extends`, `merges` and `inherits`. Polymorphism is supported for query and mapping operations (through the virtual call mechanism). No specific mechanism is defined for object identity or encapsulation.	2 (4)	1.5 (3)
Bidirectionality	The textual relations part or the graphical notation enables bidirectionality.	1 (1)	2 (2)
Multiple source models	The transformation signature allows any number of input parameters.	1 (1)	2 (2)
Learning Curve	One disadvantage is that there are many ways of doing the same thing, using relations, mappings, graphical or textual. It is however possible for a transformation writer to stick to a unique paradigm to minimize the learning effort. Another disadvantage is that there are many implicit constructions for shorthand notations that are hard to understand when you are a newcomer to this language. Advantages are that the textual language shares many similarities of both syntax and constructions with well-known object oriented languages such as Java and c#, c++. Furthermore the graphical notation is quite intuitive to understand.	2 (4)	1 (2)
Multiple target models	The transformation signature allows any number of output parameters.	1 (1)	1 (1)
Constraints between rules	Supported by the ability to specify rule ordering, pre- and post-conditions	1 (1)	1 (1)
Repetitiveness	Not been tested since this requires a lot of work on building a large proof based on the entire language specification.	-	-
	TOTAL	24 (30)	52 (57)

6. QVT Merge Example-Based Evaluation

In order to evaluate the ease of use for simple and complex transformations we provide several examples. The transformation code of each example is manually inspected by one or more persons not involved in the defining the code. Positive and negative feedback is given as text, the example is judged as either simple or complex and finally a score is provided.

The examples should as much as possible follow the following structure.

1) Informal Description of the example.

   What the transformation problem is. Define informally the transformation rules. Describe the metamodels involved in the transformation.

   State what criteria are exposed by this example.

2) Original definition of the transformation rules [OPTIONAL]

   If available, provide the definition of the transformation using any existing formalism or pseudo-code.

3) Definition of the transformation rules using Merge QVT submission

   More than one solution can be proposed.

4) Discussion.

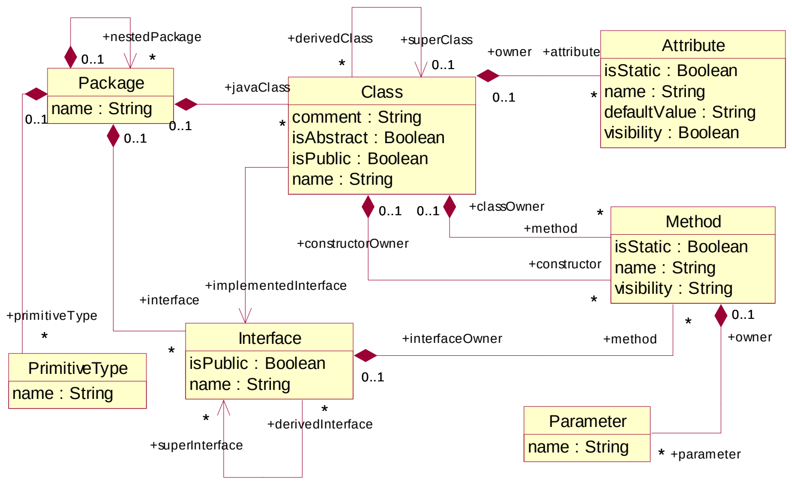
   Discuss how merge QVT solves this specific transformation problem. What specific problems where encountered? Discuss relevant criteria applied to this example.

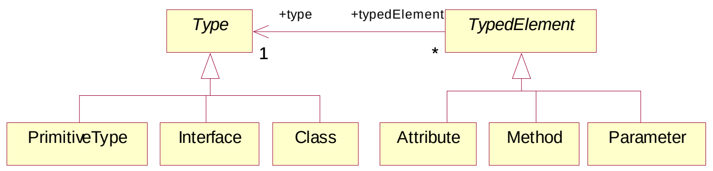
6.1. Example 1: EJB/UML → EJB/Java

A UML Class model defines a set of classes and interfaces in UML packages to represent corresponding Java classes and interfaces within Java packages. UML classes stereotyped <<EJBEntity>> represent EJB entities. The purpose of this transformation is to generate the corresponding Java instances.

6.1.1. Metamodels

The UML metamodel is the UML 1.4. We provide below the Java metamodel used in this example.





6.1.2. Rules Specification

UML packages are mapped as Java Packages, UML classes are mapped as Java classes and UML interfaces are mapped as Java interfaces. The UML classes stereotyped <<EJBEntity>> have a special treatment. The following rules apply:

- An EJBEntity maps into four entities: a Home and a Remote interface, an implementation class and a primary key class. Any reference to a UML EJBEntity – for instance in parameters - is treated as a reference to the Remote interface.
- The Home interface inherits of the predefined java.ejb.EJBHome interface. The remote interface inherits from the pre-defined java.ejb.EJBObject interface. The implementation class inherits from the pre-defined java.ejb.homeEntityBean. The primary key class inherits from the java.io.serializable interface.
- Persistent attributes in a UML EJBEntity (stereotyped <<EJBPersistent>>) are mapped as persistent attributes of the implementation class.
- Comparison attributes in a UML EJBEntity (stereotyped <<EJBCmpField>>) are mapped as attributes of the implementation class.
- Primary key fields in a UML EJBEntity (stereotyped <<EJBPrimaryKeyField>>) are mapped as an attribute in the implementation class and two utility methods – equals() and hashCode() - in the primary key class.
- Ordinary non-stereotyped operations in a UML EJBEntity are mapped as operations in the implementation class.
- Operations stereotyped <<EJBRemoteOperation>> are mapped as an operation in the implementation class and another in the remote interface.
- Operations stereotyped <<EJBCreateOperation>> are mapped as operations named "create" in the home interface.
- Operations stereotyped <<EJBFinderOperation>> are mapped as operations named "findByPrimaryKey" in the home interface.
- The "context" attribute of type java.ejb.EntityContext is added to the implementation class in addition of other predefined methods – not detailed here.

Remark: Not all the details of the mapping are provided here. For instance all primitive UML types are to be translated into Java primitive types.

6.1.3. Typical Test Example

A test model and the expected output model (optional).

6.1.4. Definition using MergeQVT

Version using QVT/Mappings

This solution uses two passes: first the Java types are built, then, in the second pass, the UML EJBEntity classes are converted. We use the "merge" extension facility – defined by the QVT Merge submission version 1.8 – to split in various rules the mapping of an EJBEntity.

Remark: this example has not been checked yet and executed using a tool, so it may contain errors.

```
module UmlEjbToJavaBean
  [in umlEjb:UML] (in javaLib:JAVA): javaBean:JAVA;
   -- UML and JAVA represent the imported model types. The variables umlEjb,
   -- javaLib and javaBean represent the extents (the models).


-- The 'getJavaClassByName' below is a utility query defined on a JAVA model type.
-- This method encapsulates the access to the pre-defined java classes
-- It accesses the top level JAVA::Package of the 'javaLib' extent and then
-- it navigates through the nested packages until reaching the class denoted
-- by the path parameter. The body is intentionally not provided to illustrate
-- the definition of black-box queries …
query JAVA::getJavaClassByName(in path:String) : JAVA::Object;

-- global accessible properties to factorize code
var javaBooleanType : JAVA::Type
  = javaLib.getJavaClassByName("java.lang.Boolean");
var javaIntegerType : JAVA::Type
  = javaLib.getJavaClassByName("java.lang.Integer");
var javaFloatType  : JAVA::Type
  =  javaLib.getJavaClassByName("java.lang.Real");
var javaStringType : JAVA::Type
  =  javaLib.getJavaClassByName("java.lang.String");

mapping main() {  -- the top level mapping
  -- using shorthands:  '[xxx]' means '->select(xxx)'
  --                     and '#MyType' means oclIsKindOf(MyType)
  var umlTopPack := umlEjb->objects()[#UML::Model]->first();
  umlTopPack.transformUmlPackagesAndTypes(); -- first pass
  umlTopPack.transformUmlEjbEntities();      -- seconfd pass
}

-- REMINDER on the mapping operation signature syntax:
--   mapping <name> [<ctxparam>] (<otherparam>,) : <outparams>

-- REMINDER on mapping invocation: the guard need to be satisfied
-- in order for a mapping to be invoked. The complete guard is made
-- of the constraints on the parameters and the condition appearing
-- after the guard keyword. If the guard is not satisfied, 'null'
-- (undefined) is returned.


-------------------------------------------------------------------
--- FIRST PASS: Convert Packages and UML types into JAVA types ---
-------------------------------------------------------------------

mapping transformUmlPackagesAndTypes [in UML::Package]() : JAVA::Package
{
  init {
    -- invokes the 'transformUmlType' for each UML::Classifier
    var javaTypes := self.ownedElement->transformUmlType();
  }
  -- population section below for the 'result' parameter
  name := self.name;
  primitiveType := javaTypes[#JAVA::PrimitiveDataType]; --
  javaClass := javaTypes[#JAVA::Class];
  interface := javaTypes[#JAVA::Interface];
  -- recursive call (for each owned element of type UML::Package)
  nestedPackage := self.ownedElement->transformUmlPackagesAndTypes();
}

mapping transformUmlType [in UML::Classifier](): JAVA::Type
  disjuncts
    -- one of three rules are invoked depending on guard evaluation
    -- ('null' is returned if all guards fail)
    transformUmlClassType,
    transformUmlEjbEntityType,
    transformUmlPrimitiveType;
    transformUmlInterfaceType;

mapping transformUmlClassType [in UML::Class]() : JAVA::Class
 guard not self.isStereotypedBy("EJBEntity") {}

mapping transformUmlEjbEntityType [in UML::Class]() : JAVA::Class
 guard self.isStereotypedBy("EJBEntity") {}

mapping transformUmlInterfaceType [in UML::Interface]() : JAVA::Interface {}

mapping transformUmlPrimitiveType [in UML::PrimitiveDataType]()
 : JAVA::PrimitiveDataType
{
  init {
    result := switch (
      self.name="boolean" ? javaBooleanType,
      self.name="string" ? javaStringType,
      self.name="integer" ? javaIntegerType,
      self.name="real" ? javaFloatType,
  }
}

-----------------------------------------------------------
--- Second PASS: Conversion of <<EJBEntity>> UML classes --
-----------------------------------------------------

-- Remark: the ":=" has additive semantics for multivalued properties
-- In addition 'null' values are skipped in multivalued assignments

mapping transformUmlEjbEntities[in UML::Package](): JAVA::Package
{
   init {
     -- the Java package is retrieved since it already created
     result := self.resolveone(JAVA::Package);
     var items:Sequence(
         Tuple {impl:JAVA::Class,home:JAVA::Interface,
                remote::JAVA::Interface,pkey:JAVA::Class})
       := self.ownedElement->transformUmlEjbEntity();
   }
   -- shorthand used here  x := {a;b;c;}
   -- concatenates the results of the evaluation of a, b and c
   javaClass := {items->i.impl;items->i.pkey;};
   interface := {items->i.home;items->i.remote;};
   nestedPackage := self.ownedElement->transformUmlEjbEntities();
}

mapping transformUmlEjbEntity [in UML::Class]()
 : impl:JAVA::Class, home:JAVA::Interface,
   remote:JAVA::Interface,pkey:JAVA::Class
 guard self.isStereotypedBy("EJBEntity")
{
  init {
    -- 'resolveoneByRule' retrieves the object created by 'transformUmlType'
    remote := self.resolveoneByRule(transformUmlType);
  }
  out impl: JAVA::Class {
    name := self.name + "_Bean";
    implementedInterface := javaLib.getJavaClassByName("java.ejb.EntityBean");
  }
  out home: JAVA::Interface {
    name := self.name + "_Home";
    superClass := javaLib.getJavaClassByName("java.ejb.EJBHome");
  }
  out remote: JAVA::Interface { -- this object is not re-created
    name := self.name;
    superClass := javaLib.getJavaClassByName("java.ejb.EJBObject");
  };
  out pkey : JAVA::Class {
    name := self.name + "_PK";
    implementedInterface := javaLib.getJavaClassByName("java.io.Serializable");
    };
  }
}

--

mapping transformPersistentAttribute [in UML::Class] ()
 merges transformUmlEjbEntity
{
   out impl : Class {
      -- '*EJBPersistent' is a shorthand for 'isStereotypedBy(EJBPersistent)'
      attribute := self.feature[#Attribute and *EJBPersistent]
         ->copyAttribute();
   }
}

mapping transformCmpField [in UML::Class] ()
 merges transformUmlEjbEntity
{
   out impl : Class {
      attribute := self.feature[#Attribute and *EJBCmpField]
         ->copyAttribute();
   }
}

mapping transformPrimaryKeyField [in UML::Class] ()
 merges transformUmlEjbEntity
{
   out impl : Class {
      attribute := self.feature[#Attribute and *EJBPrimaryKeyField]
         ->copyAttribute();
   }
   out pkey : Class {
      var javaObjectType := javaLib.getJavaClassByName("java.lang.Object")
      method := {
        -- the pre-defined 'tuple' operation creates an anonymous Tuple
        createMethod(
          "equals",Set{tuple("obj",javaObjectType)},javaBooleanType);
        createMethod("hashCode",Set{},javaIntegerType);
      };
     -- put here what to do in the primary key class
   }
}

mapping transformOrdinaryOperation [in UML::Class] ()
 merges transformUmlEjbEntity {
   init {var ops := self.feature[#Operation and hasEmptyStereotype()];}
   out impl : Class {
      method := ops->copyOperation();
   };
}

mapping transformRemoteOperation [in UML::Class] ()
 merges transformUmlEjbEntity
{
   init {var ops := self.feature[Operation and *EJBRemoteOperation];}
   out impl : Class {
      method := ops->copyOperation();
   };
   out remote : Class {
      method := ops->copyOperation();
   };
}

mapping transformCreateOperation [in UML::Class] ()
 merges transformUmlEjbEntity
 guard self.feature![#Operation and *EJBCreateMethod]<>null
{
   out impl : Class {
     method := createMethod("ejbCreate",Set{},pkey);
   };
   out home : Class {
     method := createMethod("create",Set{},remote);
   };
}

mapping transformFinderOperation [in UML::Class] ()
 merges transformUmlEjbEntity
 guard self.feature![#Operation and *EJBFinderMethod]
{
   out home : Class {
     method := findOps->createMethod("findByPrimaryKey",Set{},remote);
   };
}

mapping addPredefinedProperties [in UML::Class] ()
 merges transformUmlEjbEntity
{
   out impl : Class {
      attribute := createAttribute("context",
          javaLib.getJavaClassByName("java.ejb.EntityContext"));
      method := {
         -- add here all predefined operations of the Bean
      };
   };
}

mapping copyAttribute [in UML::Attribute]() : JAVA::Attribute {
   name := self.name;
   type := self.type.resolveoneByRule(transformUmlType);
}

mapping copyOperation [in UML::Operation] () : JAVA::Method {
   name := self.name;
   parameter := self.parameter->collect(i|
     out Parameter {
       name:=i.name;
       type:=i.type.resolveoneByRule(transformUmlType);
     });
   type := self.type.resolveoneByRule(transformUmlType);
}

-- The createAttribute is defined globally (no context parameter)
mapping createAttribute (in attrname:String,in attrtype:JAVA::Type)
 : JAVA::Attribute
{
  name := attrname;
  type := attrtype;
}

-- The createMethod is defined globally (no context parameter)
mapping createMethod (
  in opname:String,
  in inputs:Set(Tuple{name:String,type:String}),
  in resType:JAVA::Type)
  : JAVA::Method
{
  name := opname;
  parameter := inputs->collect(i|
    out Parameter{name:=i.name;type:=i.type;});
  type := resType;
}
```

-

6.1.5. Discussion

Table 3 Evaluation of example-dependant properties : EJB/UML → EJB/Java

Criteria	Scale Measure	Absolute	Weight	Comments	Score (normalized

			range = [1,6]		measure * weight)
Ease of use in simple transformations (MAPPINGS – TEXTUAL)	2 = Neither	No	6	The code has a proper structure of nicely separated mappings that are also ordered in an inheritance hierarchy which increases the reusability and maintenance. All of the mappings are also relatively concise so it is easy to grasp the main idea. The main drawback is that several of the single statements uses long and cryptic shorthand notations that require a lot of mental effort and are very difficult to interpret. This applies to collections, iterator variables, use of implicit and shorthand notations. Example1 : `var items : Sequence( Tuple{impl:JAVA::Class,home:JAVA::Interface, remote::JAVA::Interface,pkey:JAVA::Class}) := self.ownedElement->transformUmlEjbEntity();`	3

6.2. Example 2: XSLT2XQuery

The XSLT to XQuery example (originally described in [6]) describes a simplified transformation of XSLT code to XQuery code.

6.2.1. Metamodels

The source metamodel of XSLT has been modelled for this simplified transformation example. It is based on an XML metamodel, which it extends. Consequently, the XML metamodel has to be explained before the XSLT metamodel.

The XML metamodel presented (see Figure 3 XML) describes an XML document (Document) composed of one root node (RootNode). Node is an abstract class having two direct children, namely ElementNode and AttributeNode. ElementNode represents the tags, for example a tag named xml: <xml></xml>. ElementNodes can be composed of many Nodes. AttributeNode represents attributes, which can be found in a tag, for example the attr attribute: <xml attr="value of attr"/>. ElementNode has two sub classes, namely RootNode and TextNode. RootNode is the root element. The TextNode is a particular node, which does not look like a tag; it is only a string of characters.

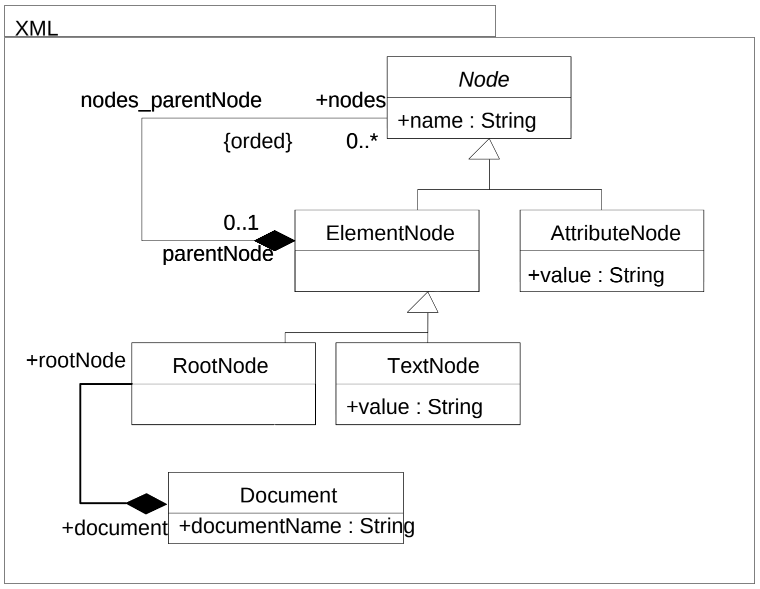


Figure 3 XML

The XSLT metamodel developed for this example (see Figure 4 XSLT) is an extension of the XML metamodel. The extension consists of classes represented in grey. The main class is called XSLTNode and inherits from ElementNode. The XSLTNode class has sub classes representing XSLT elements, namely xsl:apply-templates, xsl:template, xsl:if, xsl:value-of. For reasons of simplification, several features such as xsl:for-each, xsl:choose, xsl:sort, xsl:copy-of elements have been ignored; this is why these are neither in the metamodel nor in the transformation code.

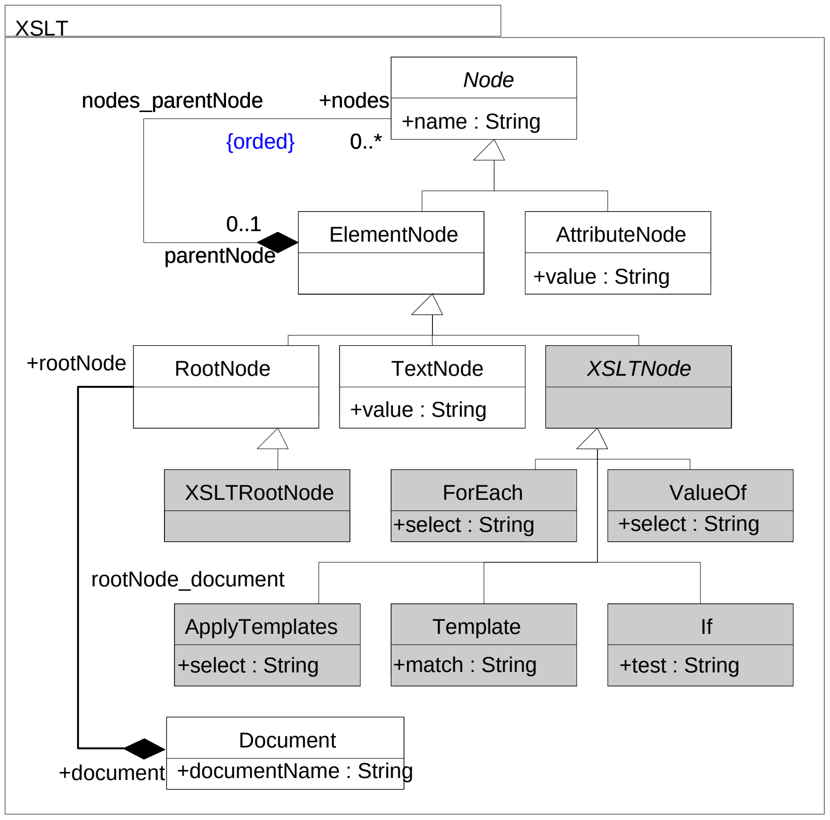


Figure 4 XSLT

The target metamodel of this example is XQuery (see Figure 5 XQuery). It contains also parts of the XML metamodel (Node, ElementNode, AttributeNode and TextNode). An XQueryProgram is composed of ExecutableExpressions which can be FLWOR expressions, function calls (FunctionCall) and function declarations (FunctionDeclaration).

The main class is FLWOR, it represents FLWOR expressions which are composed of For, Let, Where, Order by and Return statements. For is composed of an XPath expression representing the value stored by the variable defined by the var attribute. Let is also composed of an XPath expression representing the value stored by the variable defined by the var attribute. Where is composed of a boolean XPath expression used to do a selection on the variables of the For statements. OrderBy is composed of an XPath expression defining how to order the output. Return is composed of Expressions representing the output data. Expression is the superclass of ExecutableExpressions, (XML-) Nodes and ReturnXPath expressions. The Node class and its sub classes are the same as those of the XML metamodel. There are two different XPath classes. In the ReturnXPath class the corresponding String expression (value) has to be put between braces, in the XPath class the expression is without braces.

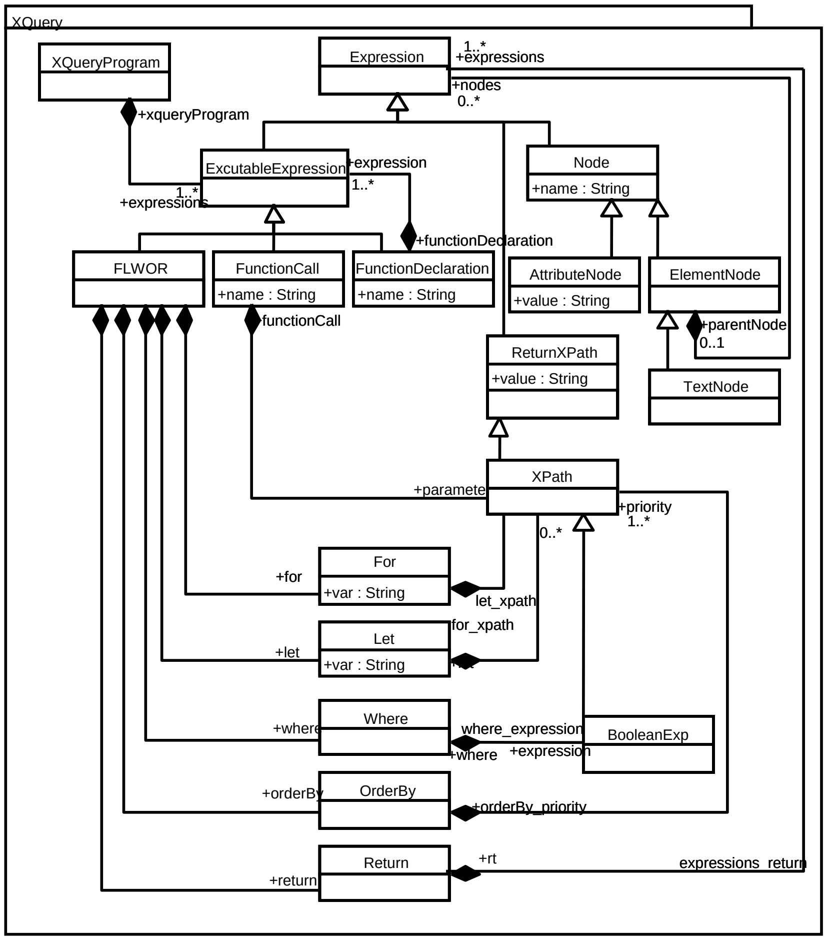


Figure 5 XQuery

6.2.2. Rules Specification

The transformation can be divided into three types of rules:

- the rule for the creation of an XQueryProgram from an XSLTRootNode instance,
- the rules for the transformation of XSLT elements into XQuery expressions and

- the rules for copying XML elements and XML attributes.

These are the rules to transform an XSLT model to an XQuery model:

- For the XSLTRootNode instance, an XQueryProgram instance has to be created. This involves follow-up instantiations and value attributions:
    - A new FLWOR instance has to be created and its references have to be set.
        - The xQueryProgram reference has to be set to the newly created XQueryProgram. The for and the return references have to point to the corresponding instances that will be described in the following.
    - A new For instance has to be created.
        - Its var attribute has to be set to '$var'.
        - Its expression reference has to point to the XPath instance that will be described in the following.
    - A new XPath instance has to be created.
        - Its value is set to 'document(\"xmlFile.xml\")'
    - A new Return instance has to be created.
        - The expressions reference set in Return has to contain all those grandchildren nodes (defined by the recursive use of the nodes reference in ElementNode) of which the children Template nodes of the XSLTRootNode have the match value '/'. In other words, select all instances of XSLTRootNode referenced in nodes and choose all those Templates having the match value '/'. Let the expressions references of this Return instance point to all those elements referenced by the elements that correspond to the nodes of the chosen Templates.

- For each XSLT Template instance, an XQuery FunctionDeclaration instance has to be created, if the match value is not '/'. This involves follow-up instantiations.
    - The new FunctionDeclaration instance has the following value and references:
        - The name of the FunctionDeclaration is 'fct' concatenated with the match String.
        - Its expression reference is a sequence of FLWOR instances that will be described below.
        - Its xQueryProgram reference points to the first XSLTRootNode instance.
    - A new FLWOR has to be created.
        - Its for and return references have to point to the corresponding instances described in the following.
    - A new For instance has to be created.
        - Its var value is '$var'.
        - Its forExpresson points to the corresponding XPath instance described in the following.
    - A new Path instance has to be created.
        - Its value attribute has to be set to '$paramVar'
    - A new Return instance has to be created.

        - Its expressions references corresponds the node references of Template.

- For each XSLT If instance, an XQuery FLWOR instance has to be created.
    - This involves also the instantiation of a Let, a Where and a Return variable which have to be referenced by the corresponding references (let, where and return) in this FLWOR instance.
    - A new Let instance has to be created.
        - The expression reference has to reference the new XPath instance described below.
        - The var attribute has to be set to '$var'.
    - A new XPath instance has to be created.
        - The value attribute has to be set to '$var'.
    - A new Where instance has to be created.
        - The expression reference has to reference the new BooleanExp instance described below.
    - A new BooleanExp has to be created.
        - Its value attribute has to be set to '$var' concatenated with the test attribute of the If instance.
    - A new Return instance has to be created.
        - Its expressions references have to point to the elements that correspond to the nodes references of the If instance.
- For each XSLT ApplyTemplate instance, an XQuery FunctionCall instance has to be created.
        - The name attribute has to be set to 'fct' concatenated with the select attribute of the ApplyTemplate instance.
        - Its parameters reference has to reference the XPath described below.
    - A new XPath instance has to be created.
        - Its value attribute has to be set to '$var' concatenated with the select attribute of the ApplyTemplate instance.
- For each XSLT ValueOf instance, an XQuery ReturnXPath instance has to be created.
    - Its value attribute has to be set to '$var' concatenated with the _valueOf attribute of the ValueOf instance.
- For each XSLT ElementNode instance that has a name different from xsl:otherwise, xsl:when, xsl:choose, xsl:copy-of, xsl:sort, xsl:foreach, xsl:if, xsl:apply-template, xsl:value-of, xsl:template and xsl:stylesheet, an XQuery ReturnXPath instance has to be created.
    - The name attributes of ElementNode and ReturnXPath have to correspond.
    - The nodes references of ElementNode and ReturnXPath have to correspond.
- For each XSLT AttributeNode instance, an XQuery AttributeNode instance has to be created with the same values.
    - The name attributes of ElementNode and ReturnXPath have to correspond.
    - The value attributes of ElementNode and ReturnXPath have to correspond.

The transformation described is simplified with the following constraints:

- All the template tags must be direct children of the root node. This constraint simplifies the behaviour of templates.
- The value of a select attribute of an apply-template must be a tag name, it must not be an XPath expression. This constraint hides the main difference between a template and a function call. An apply-template tag applies all available templates to a set of elements and each template treats only the elements that it is dedicated to. Whereas a function call applies a function to a set of elements; the test of type of the elements must be explicitly described in the function declaration.

The XSLT programmer has to write one template matching to '/'. It defines indirectly the starting point. This information is necessary with respect to the XQuery program. XQuery is partly an imperative language; it defines the order of the program execution.

6.2.3. Typical Test Example

The following example illustrates the transformation from XSLT to XQuery. It searches for all employees with a salary greater than 2000 and returns their name and their first name.

From the XSLT source code:
```
      <xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
            <xsl:template match="/">
                  <emps>
                    <xsl:apply-templates select="employee"/>
                  </emps>
            </xsl:template>
            <xsl:template match="employee">
              <xsl:if test="salary&gt;2000">
                <emp>
                  <xsl:value-of select="name"/>
                  <xsl:value-of select="firstname"/>
                </emp>
              </xsl:if>
            </xsl:template>
      </xsl:stylesheet>
```

the following XQuery code is produced:

```
      define function fctemployee($paramVar) {
            for $var in $paramVar
            return
                  let $var := $var
                  where $var/salary>2000
                  return
                        <emp>{$var/name}{$var/firstname}</emp>
      }
      for $var in document("xmlFile.xml")/*
      return
            <emps>{fctemployee($var/employee)}</emps>
```

6.2.4. Definition using MergeQVT

Version with QVT/Mappings

The QVT/Mapping code for the transformation of XSLT to XQuery is provided below:

```
modeltype XSLT, XQuery;
module XSLT2XQuery(inModel:XSLT) : outModel:XSLT;

-- renaming is needed to avoid clashes with keywords
renamed property FLWOR::_where = "where";
renamed property FLWOR::_var = "var";
renamed property FLWOR::_let = "let";

mapping P2P [in XSLTRootNode] () : res:XQueryProgram
-- 'self' refers to the input context parameter
-- Three objects are created here but a unique object is returned
{
  out res:XQueryProgram {
     expressions := out FLWOR {
        for := out For {
            _var := '$var';
            XPath := out XPath {
               value := 'document(\"xmlFile.xml\")/*';
            };
        };
     };
     return := out Return {
         expressions := self.nodes[#Template][t|t.match = '/']
               ->nodes->flatten()->NodeToExpression();
     };
  };
}

mapping NodeToExpression [in XSLT::ElementNode] () : XQuery::Expression
merges
   -- the following rules are potentially called by this mapping
   -- a mapping is called only if the signature and/or guard matches
   Template2FLOWR, Attribute2Attribute, ApplyTemplates2FunctionCall,
   ValueOf2ReturnXPath, ElementNode2ElementNode, If2FLOWR
{}


mapping Template2FLOWR [in Template] () : pFlwor:FLWOR
guard self.match <> '/'
{
   out pFdecl:FunctionDeclaration {
       name := 'fct' + self.match;
      expression := Sequence {pFlwor};
      xQueryProgram := outModel->objectsOfType(XSLTRootNode)->first();
   };
   out pFlwor : FLWOR {
      for := out For {
            XPath := out XPath { value := '$paramVar';};
            _var := '$var';
      };
      return := out Return {
         expressions := self.nodes->NodeToExpression();
      }
   };
}

mapping If2FLOWR [in If] () : pFlwor:FLWOR
{
  out pFlwor : FLWOR {
    _let   := out Let { XPath := pLetExpression; _var := '$var';};
    _where := out Where {expression := pWhereExpression;};
    return := out Return {expressions := self.nodes->NodeToExpression();};
  };
  -- this two object creations are not inlined simply for readability
  out pLetExpression : XPath {value := '$var';};
  out pWhereExpression : BooleanExp {value := '$var/' + self.test;};
}

mapping ApplyTemplates2FunctionCall [in ApplyTemplates] () {
  out functionCall : FunctionCall {
         name := 'fct' + self.select;
         parameters := out XPath {value := '$var/' + self.select;};
  };
}
```

```
-- we use here a notation shorthand
-- the body contains an implicit 'out result: ReturnXPath' block
mapping ValueOf2ReturnXPath [in ValueOf] () : ReturnXPath {
  value := '$var/' + self.select;
}

literal xslkeys = Sequence(String) {
 'xsl:otherwise', 'xsl:when', 'xsl:choose', 'xsl:copy-of',
 'xsl:sort', 'xsl:foreach', 'xsl:if', 'xsl:apply-templates',
 'xsl:value-of', 'xsl:template', 'xsl:stylesheet'
}

mapping ElementNode2ElementNode [in XSLT::ElementNode] ()
: XQuery::ElementNode
guard  not xslkeys->exists(self.name)
{
   name := self.name;
   nodes := self.nodes->ElementNode2ElementNode();
}

mapping Attribute2Attribute [in XSLT::AttributeNode] ()
: XQuery::AttributeNode
guard  not xslkeys->exists(self.name)
{
   name:=self.name;
   value:=self.value;
}
```

6.2.5. Discussion

Table 4 Evaluation of example-dependant properties : XSLT → XQuery

Criteria	Scale Measure	Absolute	Weight range = [1,6]	Comments	Score (normalized measure * weight)
Ease of use in complex transformations (MAPPINGS – TEXTUAL)	3 = Agree	No	6	Nicely separation into independent rules and the overall transformation is quite easy for this relatively complex task. The extensive use of out expressions is confusing. One would expect that they refer to separate out parameters which they don't. Furthermore the mapping signatures lack some out parameters (Not good if this is legal code) and there are a few single statements that are long and cryptic.	4,5

6.3. Example 3: UML SPEM profile → UML SPEM metamodel

The SPEM standard (Software Process Engineering Metamodel) is defined using a metamodel and a UML Profile. The profile is typically used within a UML case tool. Another tool may implement only the metamodel. The profile to metamodel transformation allow exchanging between these two kinds of tools. Note that the inverse transformation is not straightforward because there may be various ways to encode a single SPEM concepts using UML – this is the case for instance for work definitions which may be represented using

UseCases or using ActivityGraphs. In our example we assume that work definitions are represented using use-cases.

6.3.1. Metamodels

The used metamodels are the UML 1.4 metamodel [xxx] and the SPEM 1.0 metamodel [7]. The SPEM metamodel extends a sub-set of the UML metamodel with a list of process-specific concepts.

NOTE: The metamodels are too large to be described in this document.

6.3.2. Rules Specification

Below a partial definition of the mapping rules:

- A UML package is translated to a SPEM Package unless it represents a ProcessComponent or a Discipline.
- A UML Package stereotyped ProcessComponent is translated to a ProcessComponent
- A UML Package stereotyped Discipline is translated to a Discipline
- A UML UseCase stereotyped LifeCycle is translated to a LifeCycle.
- A UML UseCase stereotyped Phase is translated to a Phase.
- A UML UseCase stereotyped Iteration is translated to a Iteration.
- A UML UseCase stereotyped Activity is translated to a Activity.
- A UML UseCase stereotyped WorkDefinition is translated to a WorkDefinition.
- A UML Actor stereotyped ProcessRole is translated to a ProcessRole
- A UML Actor stereotyped ProcessPerformer is translated to ProcessPerformer
- A UML Constraint stereotyped "precondition" is translated to Precondition
- A UML Constraint stereotyped "goal" is translated to "Goal"
- The performer of a WorkDefinition is derived using the associations between the UseCase and the Actors stereotyped "perform". This applies in particular to all sub-classes of work definitions (Phase, Iteration, Activity and LifeCycle). If no performer is found the performer will be a ProcessPerformer unique instance defined for the entire modelled process. Note that a WorkDefinition can only have one performer.
- The assistants of the Activities are derived using the associations between the UseCase and the Actors stereotyped "assist".
- The work definition decomposition is derived using the UseCase dependencies stereotyped "includes".

6.3.3. Typical Test Example

A test model and the expected output model (optional).

6.3.4. Definition using MergeQVT

Version with QVT/Mappings

```
module SpemProfile2Metamodel[in umlmodel:UML] () : SPEM;

query UML::Classifier::getOppositeAends() : Set(AssociationEnd);
```

```
main () {
  -- first pass: create all the SPEM elements from UML elements
  umlmodel.objects[#Model]->createDefaultPackage();
  -- second pass: add the dependencies beyween SPEM elements
  umlmodel.objects[#UseCase]->addDependenciesInWorkDefinition();
}

mapping createDefaultPackage [in UML::Package] () : SPEM::Package {
  name := self.name;
  ownedElement := self.ownedElement->createModelElement();
}

mapping createProcessComponent [in UML::Package] () : ProcessComponent
  inherits createDefaultPackage
  guard self.isStereotypedBy("ProcessComponent")
  {}

mapping createDiscipline [in UML::Package] () : Discipline
  inherits createDefaultPackage
  guard self.isStereotypedBy("Discipline") {}


mapping createModelElement [in UML::ModelElement] () : SPEM::ModelElement
  disjuncts
    createProcessRole, createWorkDefinition,
    createProcessComponent, createDiscipline
  {}

mapping createWorkDefinition [in UseCase] () : WorkDefinition {
  disjuncts
    createLifeCycle, createPhase, createIteration,
    createActivity, createCompositeWorkDefinition
  {}
}

mapping createProcessRole [in Actor] () : ProcessRole
  guard self.isStereotypedBy("ProcessRole")
  {}

-- rule to vreate the default process performer singleton
mapping createOrRetrieveDefaultPerformer () : ProcessPerformer {
  init {
    result := resolveoneByRule(createOrRetrieveDefaultPerformer);
  }
}

abstract mapping createCommonWorkDefinition [in UseCase] () : WorkDefinition
{
   name := self.name;
   constraint := {
      self.constraint[*precondition]->createPrecondition();
      self.constraint[*goal]->createGoal();
   };
}

mapping createActivity [in UseCase] () : WorkDefinition
  inherits createCommonWorkDefinition
  guard self.isStereotypedBy(Activity)
  {}

mapping createPhase [in UseCase] () : Phase
  inherits createCommonWorkDefinition
  guard self.isStereotypedBy(Phase)
  {}

mapping createIteration [in UseCase] () : Iteration
  inherits createCommonWorkDefinition
  guard self.isStereotypedBy(Iteration)
  {}

mapping createLifeCycle [in UseCase] () : LifeCycle
  inherits createCommonWorkDefinition
  guard self.isStereotypedBy(LifeCycle)
  {}

mapping createCompositeWorkDefinition [in UseCase] () : WorkDefinition
  inherits createCommonWorkDefinition
```

```
    guard self.isStereotypedBy(WorkDefinition)
    {}

  mapping createPrecondition [in UML::Constraint] () : Precondition {
    body := self.body;
  }

  mapping createGoal [in UML::Constraint] () : Goal {
    body := self.body;
  }

  mapping addDependenciesInWorkDefinition [in UseCase] () : WorkDefinition {
    init {
      result := self.resolveone(WorkDefinition);
      var performers
        := self.getOppositeAends()[i|i.association[*perform]->notEmpty()];
      assert("A unique performer is allowed",self,
             not performers->size()>1)
    }
    subWork := self.clientDependency[*includes].supplier
       ->resolveone(WorkDefinition);
    performer := if performers then performers->first()
                 else createOrRetrieveDefaultPerformer() endif;
  }

  mapping addDependenciesInActivity [in UseCase] () : WorkDefinition
    merges addDependenciesInWorkDefinition
    guard self.isStereotypedBy("Activity")
    {
      assistant := self.getOppositeAends[i|i.association[*assist]->notEmpty()]->resolve();
    }
```

6.3.5. Discussion

Table 5 Evaluation of example-dependant properties : UML SPEM profile → UML SPEM metamodel

Criteria	**Scale Measure**	**Absolute**	**Weight** **range = [1,6]**	**Comments**	**Score (normalized measure * weight)**
Ease of use in complex transformations (MAPPINGS – TEXTUAL)	3 = Agree	No	6	Nicely separation into independent rules and good exploitation of the inheritance possibilities. A few single statements that are long and cryptic requires a lot of mental effort.	4,5

6.4. Example 4 UML → RDBMS

This transformation example illustrates the translation of a UML class-diagram like model into a relational data base. This example is directly taken from the MergeQVT submission version 1.8.

6.4.1. Metamodels

A simplified UML meta-model is shown in Figure A2-1. A class has attributes. An attribute's type can be either a primitive data type or another class (complex types). Classes are related to each other through Association objects. Only classes that are marked as persistent for the property kind are considered for mapping. Some attributes have the property kind set to Primary to indicate that they are the key attributes.

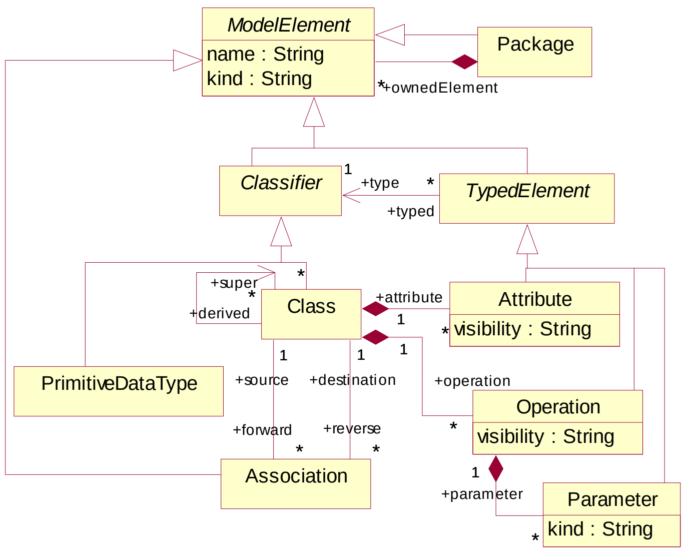


Figure A2-1 : A simple UML meta-model

A sample RDBMS meta-model is shown in Figure below. A table has columns. Every table has a mandatory primary key (Key). A table may optionally have foreign keys. A foreign key refers to a primary key of another associated table.

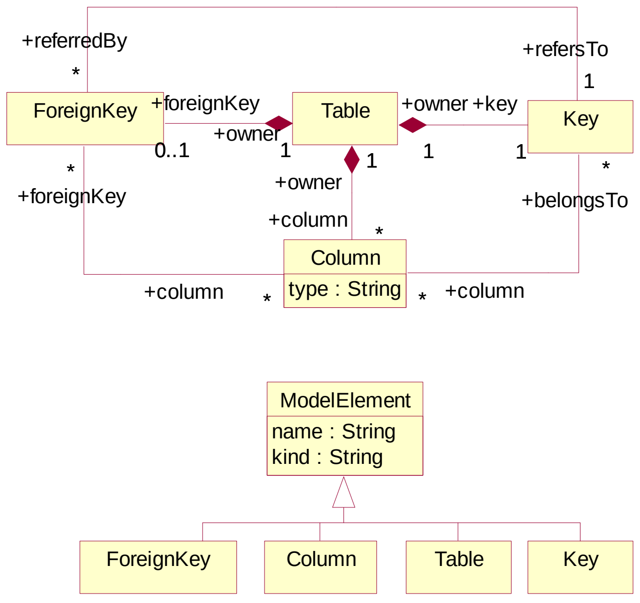


Figure A2-2 : A simple RDBMS meta-model

.

6.4.2. Rules Specification

"A class maps on to a single table. A class attribute of primitive type maps on to a column of the table. Attributes of a complex type are drilled down to the leaf-level primitive type attributes; each such primitive type attribute maps onto a column of the table. An association maps on to a foreign key of the table corresponding to the source of the association. The foreign key refers to the primary key of the table corresponding to the destination of the association."

6.4.3. Typical Test Example

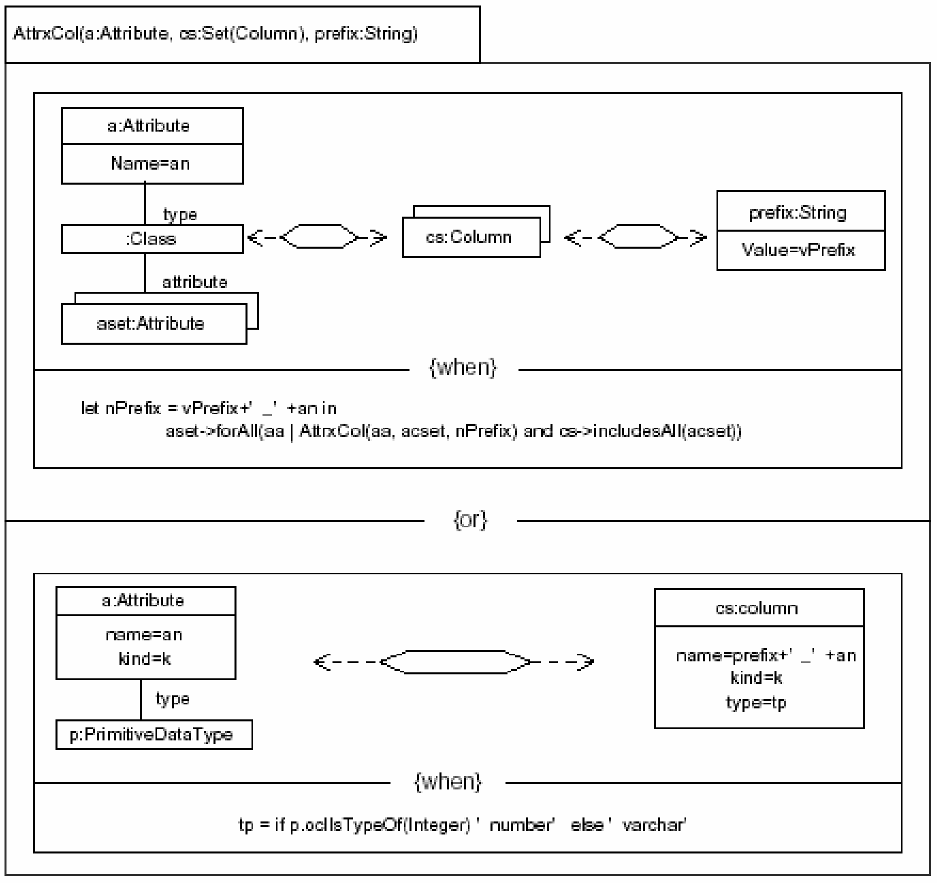
A test model and the expected output model (optional).

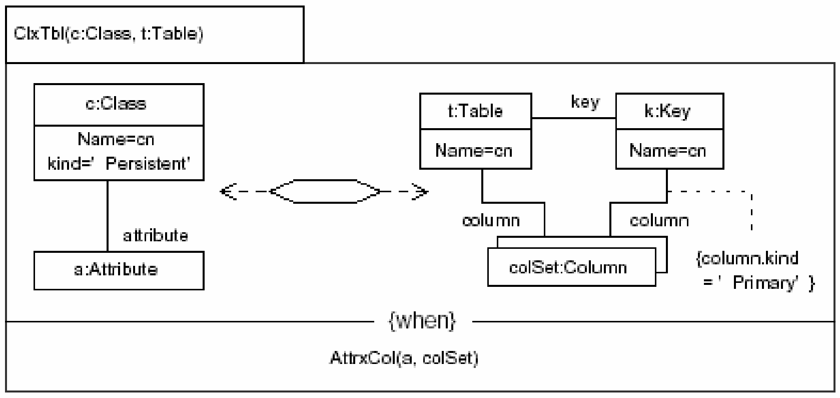
6.4.4. Definition using MergeQVT

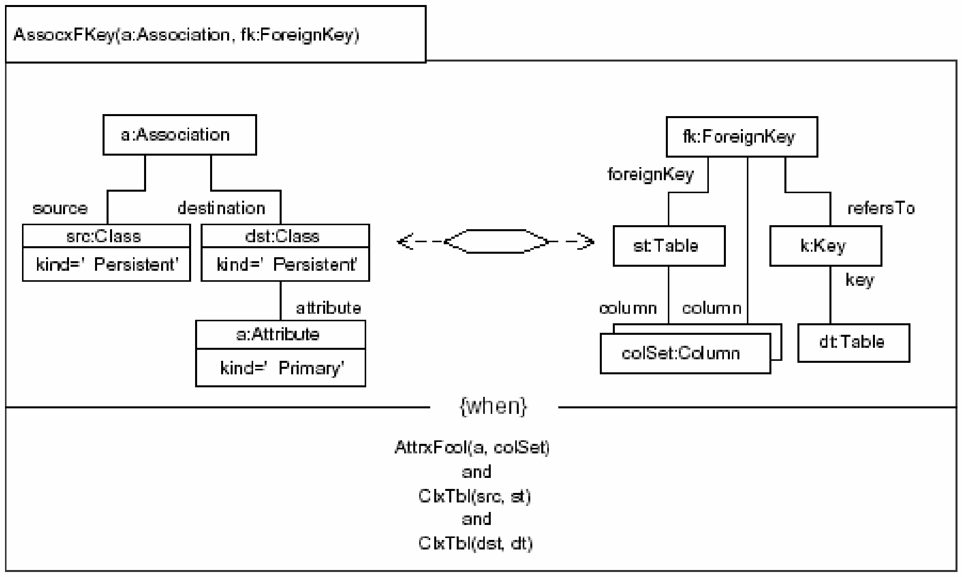
Version with QVT/Relations

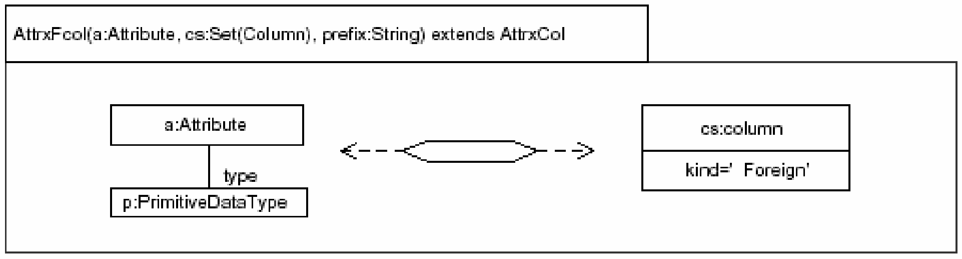
The solution is provided using the graphical notation.

[TODO: Provide explanatory text here].


AttrxCol(a:Attribute, cs:Set(Column), prefix:String)
a:Attribute
Name=an
type
:Class
attribute
aset:Attribute
cs:Column
prefix:String
Value=vPrefix
{when}
let nPrefix = vPrefix+' _' +an in
aset->forAll(aa | AttrxCol(aa, acset, nPrefix) and cs->includesAll(acset))
{or}
a:Attribute
name=an
kind=k
type
p:PrimitiveDataType
cs:column
name=prefix+' _' +an
kind=k
type=tp
{when}
tp = if p.oclIsTypeOf(Integer) ' number' else ' varchar'






AttrxFcol(a:Attribute, cs:Set(Column), prefix:String) extends AttrxCol
a:Attribute
type
p:PrimitiveDataType
cs:column
kind=' Foreign'

Version with QVT/Mappings

```
-- declaring the transformation module

module Uml2Rdb(in srcModel:UML) : RDBMS;

-- defining specific helpers and derived properties

metamodel UML {
  query Association.isPersistent() =
    (self.source.kind='persistent' and self.destination.kind='persistent');
  derived property Class.leafAttributes : Sequence(LeafAttribute);
}
```

```
-- defining intermediate data to reference leaf attributes that may
-- appear when struct data types are used

class LeafAttribute {name:String;kind:String;attr:Attribute;};

-- defining the default entry point for the module
-- first the tables are created from classes, then the tables are
-- updated with the foreign keys implied by the associations

main() {
  srcModel.objects()[#Class]->class2table(); -- first pass
  srcModel.objects()[#Association]->asso2table(); -- second pass
}

-- maps a class to a table, with a column per flattened leaf attribute

mapping class2table [in Class] () : Table
  guard self.kind='persistent' -- 'self' refers to the first parameter
{
  init { -- performs any needed intialization
    self.leafAttributes := self.attribute->attr2LeafAttrs();
  }
  -- population section for the table
  name := 't_' + self.name;
  column := self.leafAttributes->leafAttr2OrdinaryColumn();
  key := out Key {  -- nested population section for a 'Key'
           name := 'k_'+ self.name; column := t.column[kind='primary'];
         };
}

-- Mapping that creates the intermediate leaf attributes data.

mapping attr2LeafAttrs [in Attribute]
  (in prefix:String="",in pkind:String="")
: Sequence(LeafAttribute) {
  init {
    var k := if pkind="" then self.kind else pkind endif;
    result :=
       if self.type.isKindOf(PrimitiveDataType)
       then -- creates a sequence with a LeafAttribute instance
         {out LeafAttribute {attr:=self;name:=prefix+self.name;kind:=k;}}
       else self.type.attribute.attr2LeafAttrs(self.name+"_",k)
       endif;
  }
}

-- Mapping that creates an ordinary colum from a leaf attribute

mapping leafAttr2OrdinaryColumn [in LeafAttribute] (in prefix:String="")
: Column {
  name := prefix+self.name;
  kind := self.kind;
  type := if self.attr.type.name='int' then 'NUMBER' else 'VARCHAR' endif;
}

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

mapping asso2table[in Association] () : Table
  guard self.isPersistent()
{
  init { result := self.destination.resolveone(Table); }
  foreignKey := self.asso2ForeignKey();
  column := result.foreignKey.column;
}

-- mapping to build the foreign keys

mapping asso2ForeignKey [in Association] {
   name := 'f_' + name;
   refersTo := self.source.resolveone(Table).key;
   column := self.source.leafAttributes[kind='primary']
              .leafAttr2ForeignColumn(source.name+'_');
}

-- Mapping to create a Foreign key from a lef attributes
-- Inheriting of leafAttr2OrdinaryColumn has the effect to call the
-- inherited rule before entering the property population section
```

```
mapping leafAttr2ForeignColumn [in LeafAttribute] (in prefix:String) : Column
  inherits leafAttr2OrdinaryColumn {
     kind := "foreign";
}
```

6.4.5. Discussion

Table 6 Evaluation of example-dependant properties: UML → RDBMS

Criteria	Scale Measure	Absolute	Weight range = [1,6]	Comments	Score (normalized measure * weight)
Ease of use in simple transformations (RELATIONS – GRAPHICAL)	1 = Disagree	No	6	It is hard to understand the graphs involving sets of objects. The use of WHEN seems inappropriate when there is just an assignment. One would expect a boolean condition following such a keyword. The graphical definitions lacks associating comments that explains the non-trivial issues.	
Ease of use in simple transformations (MAPPINGS – TEXTUAL)	2 = Neither	No	6	It is confusing to have two parameter lists and it is not intuitive that the "self" reference refers to the first parameter. When to use "→" and when to use "." for invoking a mapping method associated with an object.	

6.5. The Example 5: Book → Publication

The Book to Publication example describes a very simple transformation task. In the metamodel Book the class Book contains an ordered set of Chapters. These Chapters hold the information of the number of pages of Chapters. The metamodel Publication is simpler; its class Publication contains a title and the total number of pages. For the transformation, all chapters of a Book have to be visited to calculate the total number of pages.

6.5.1. Metamodels

The source metamodel Book (see Figure 6 Book) consists of the class Book which contains a set of Chapters. Each Book has a title and each Chapter a title. The Chapter instances hold the information of the number of pages.

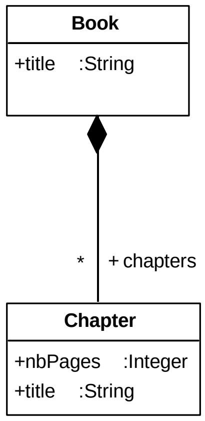


Figure 6 Book

The target metamodel Publication (see Figure 7 Publication) consists of the class Publication which holds a title and the number of pages.

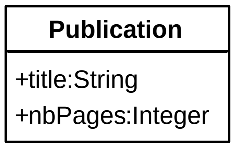


Figure 7 Publication

6.5.2. Rules Specification

These are the rules to transform a Book model to a Publication model:

- For each Book instance, a Publication instance has to be created. The attributes of the Publication instance are set as follows:
    - The title of a Publication has to be set with the title of a Book.
    - The total number of pages of a Publication is the sum of the pages of the Chapters of a Book.

6.5.3. Typical Test Example

A test model and the expected output model (optional).

6.5.4. Definition using MergeQVT

Version using QVT/relations

The code for the transformation of a Book to a Publication consists of one relation. In this transformation the sum of the number of pages of all Chapters corresponds to the number of pages of a publication.

```
relation Book_and_Publication {
    domain  b:Book {title = t};
    domain  Publication {title = t, nbPages = b.chapters.nbPages->sum()};
}
```

Version using QVT/Mappings

The code for the transformation of a Book to a Publication consists of one mapping.

```
mapping Book_to_Publication [in Book]() : Publication {
    title := self.title;
    nbPages := self.chapters.nbPages->sum();
}
```

6.5.5. Discussion

Table 7 Evaluation of example-dependant properties : Book → Publication

Criteria	Scale Measure (Min = 0, Max = 1)	Absolute	Weight range = [1,6]	Comments	Score (normalized measure * weight)
Ease of use in simple transformations (RELATIONS-TEXTUAL)	3 = Agree	No	6	It is a bit difficult to come up with and feel certain of the correctness of the expression b.chapters.nbPages->sum(). b.chapters is obviously a set, while b.chapters.nbPages is not so obviously a set.	
Ease of use in simple transformations (MAPPINGS-TEXTUAL)	3 = Agree	No	6	<<Same as above>>	

6.6. The Example 6: EDOC → J2EE

This example is a transformation of the EDOC metamodel to the J2EE metamodel. The transformation was originally used within the Fraunhofer transformation tool chain. This tool chain is based on a different approach and is not using QVT or a QVT like language to express the transformations. The transformations are directly implemented in C++ code.

The transformations described within this example are used in an industry related project that was aiming a special application domain. For this reason the used J2EE metamodel is not supposed to be a general purpose metamodel for J2EE that could be used for all J2EE based applications in general. But is a metamodel that sufficiently serves the purpose to support the transformation from EDOC models to the J2EE models and J2EE applications respectively. In that sense the transformation described here are also limited to the scope of the code generation for this particular scope of the application domain.

The next sections describe the source metamodel which is the EDOC metamodel, the target metamodel which is the Fraunhofer specific J2EE metamodel and the transformation rules expressed in different styles which allows the transformation from EDOC to J2EE.

6.6.1. Metamodels

For this example two metamodels are of importance. The first one is the EDOC metamodel. This metamodel is part of the EDOC specification [8]. The second metamodel is the J2EE metamodel. Since there is no standardised metamodel for J2EE a proprietary metamodel was designed. This metamodel does not cover all concepts of J2EE but it is sufficient to be used in a Fraunhofer tool chain that produces J2EE code skeletons.

6.6.1.1. EDOC Metamodel

The description of the EDOC metamodel is not done here explicitly since it is directly used from the EDOC specification [8]. The EDOC metamodel and more details about EDOC are within this specification. Parts of the EDOC metamodel are displayed later on for the overall description of the transformations.

6.6.1.2. The J2EE Metamodel

The J2EE metamodel that is described is based on a Java metamodel that covers parts of the Java language. This is done to allow the generation of Java classes implementing the abstract concepts of J2EE. This metamodel is not only used to describe the basic data types of the parameters and exceptions. Additionally it defines basic entities like Java interfaces and classes that are used as supertypes of J2EE specific artefacts. The following rules apply:

- A remote interface is (extends) a javax.ejb.EJBObject that itself is a java.rmi.Remote interface.
- A local interface is a javax.ejb.EJBLocalObject interface.
- A home interface is a javax.ejb.EJBHome that itself is a java.rmi.Remote interface.
- A local home interface is a javax.ejb.EJBLocalHome interface.
- A bean's implementation class implements either a javax.ejb.SessionBean or a javax.ejb.EntityBean or a javax.ejb.MessageDrivenBean interface, which are all javax.ejb.EnterpriseBean interfaces, which itself is a java.io.Serializable interface.

Thus the J2EE metamodel will be a superset of the Java metamodel.

6.6.1.2.1. Java Metamodel

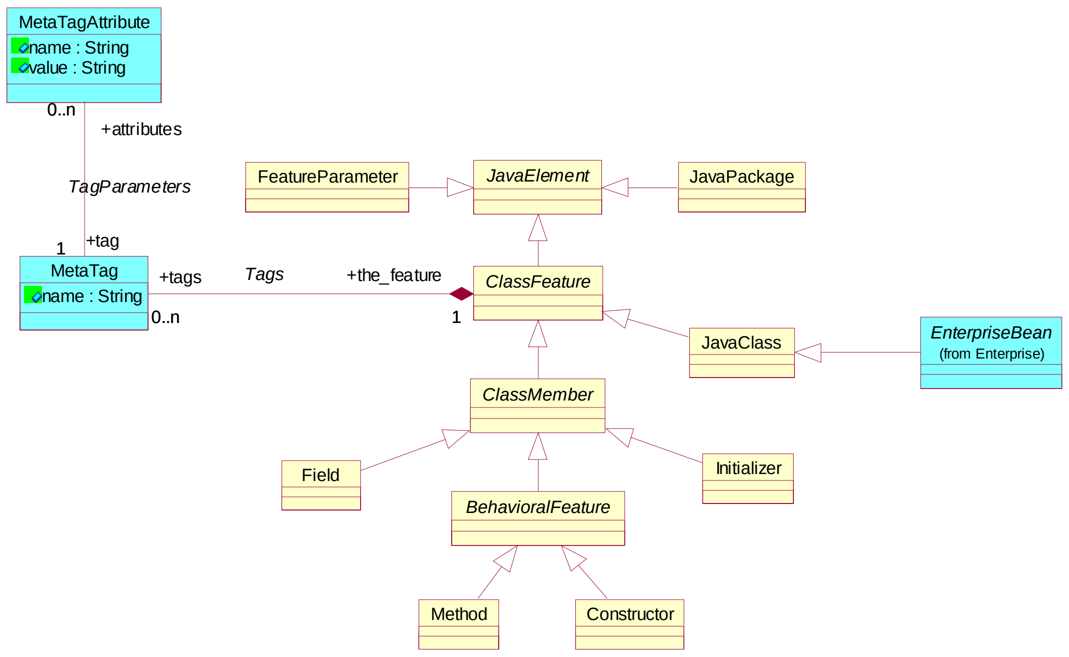


Figure 8 Main Elements of Java Metamodel

The most important parts of the Java metamodel and their inheritance hierarchy are shown in Figure 8. The elements *Method*, *JavaClass*, and *ClassFeature* are of special interest.

- A *JavaClass* represents Java classes and interfaces and will be used as a hook to connect the Java and J2EE metamodels.
- All methods that are used for the definition of the remote, local, and home interfaces as for the implementation class are instances of the model element *Method*.
- The *ClassFeature* will be used as a hook to add meta tags to Java elements.

Some other parts of the Java metamodel will be used as targets of the transformation process thus they will be introduced shortly.

JavaPackages and *JavaClasses* represent the corresponding artefacts in Java. Packages can be included in them selves to create a hierarchy of packages, a so called *package tree*. The same mechanism is available for classes, so called *nested classes*, see Figure 9.

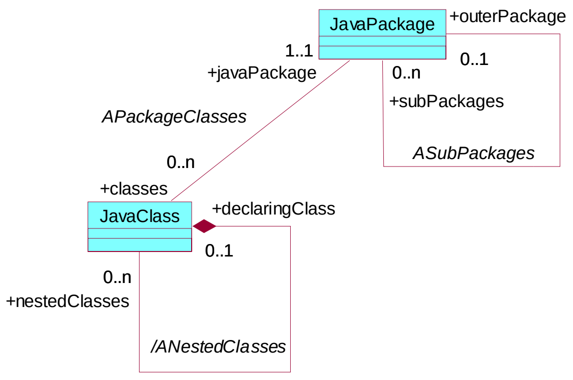


Figure 9 Java packages and classes

Adding content to a Java class is modelled with the following features.

- A *Method* represents a, possibly typed, Java method.
- A *Constructor* is a special, untyped method that is always called once while creating a new instance.
- A *Field* holds the state of a class explicitly for one instance or shared over all instances of a class.
- An *Initializer* is used for the initialization of class attributes.
- A *ClassDescriptor* describes the inheritance hierarchy and the implemented interfaces of the referenced Java class.
- A *FeatureParameter* specifies the parameters of constructors and methods. It holds relations to its type and is declared explicitly for each element.

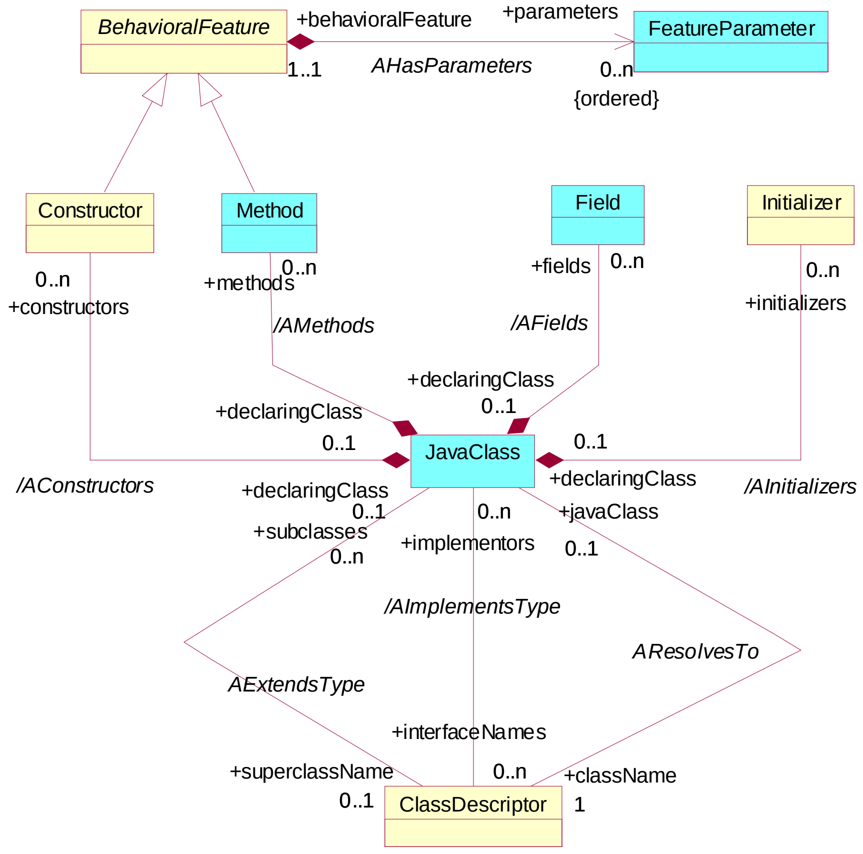


Figure 10 Features of a Java class

A Java primitive type is specified by the *PrimitiveTag* enumeration that is modelled as an attribute in the *PrimitiveType* class.

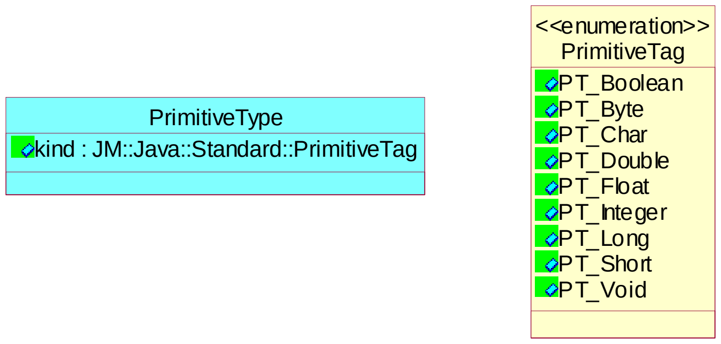


Figure 11 Java Primitive Types

6.6.1.2.2. EJB Metamodel

Three types of enterprise beans exist: session, entity, and message driven beans. Thus the part of the EJB metamodel that represents the *implementers view* looks like the diagram shown in Figure 12. The three types of enterprise beans are shown as subtypes of the generic *EnterpriseBean* that itself is a *JavaClass*. Additionally it is shown that entity beans have an attribute called *primary_key* that is of the type *ClassDescriptor*.

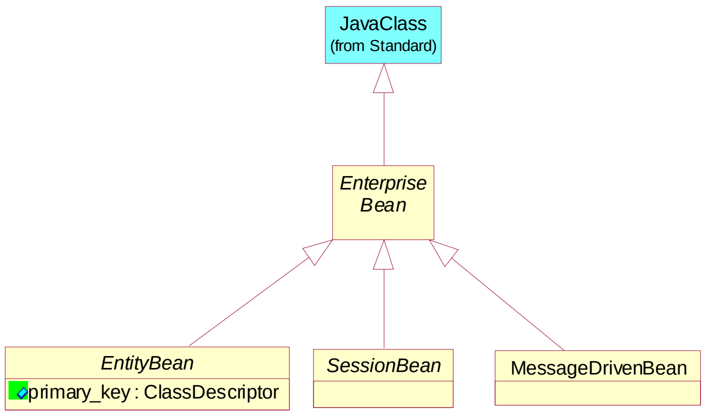


Figure 12 Implementers view of enterprise beans

Complementary the client view on enterprise beans is shown in Figure 13. This part of the metamodel expresses, that the four possible interface types of an enterprise beans are all *EJBInterfaces* that itself is a *JavaClass*. Additionally it is shown that every enterprise bean has an association to one or zero of these interfaces.

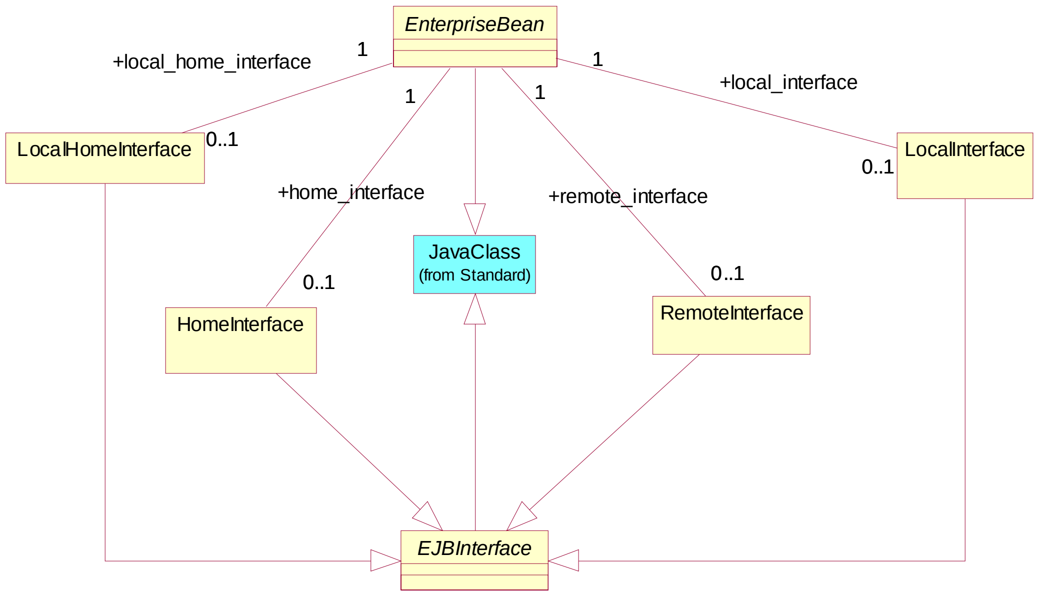


Figure 13 Client view of enterprise beans

Figure 12shows only the upper part of the hierarchy of enterprise beans. The whole tree is shown in Figure 14. Additionally the relationships between entity beans are modelled as an association called *cmr*.

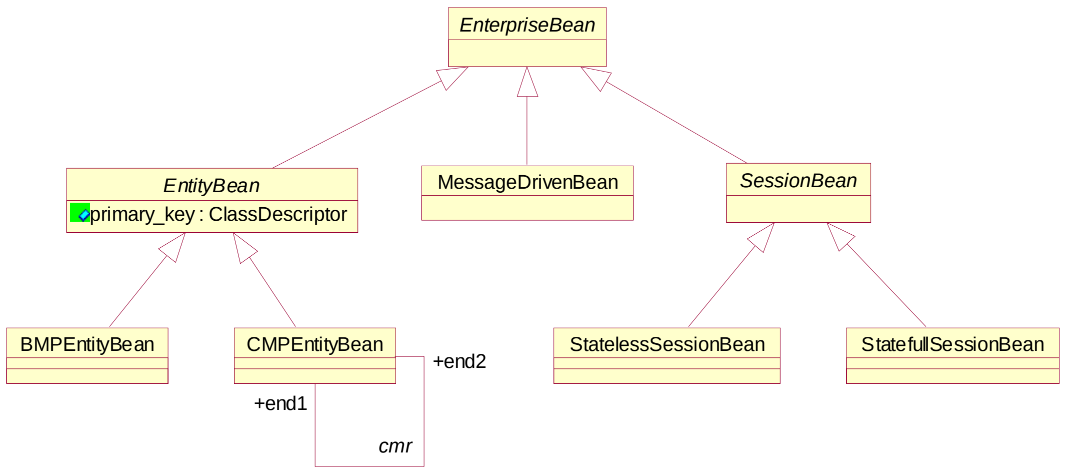
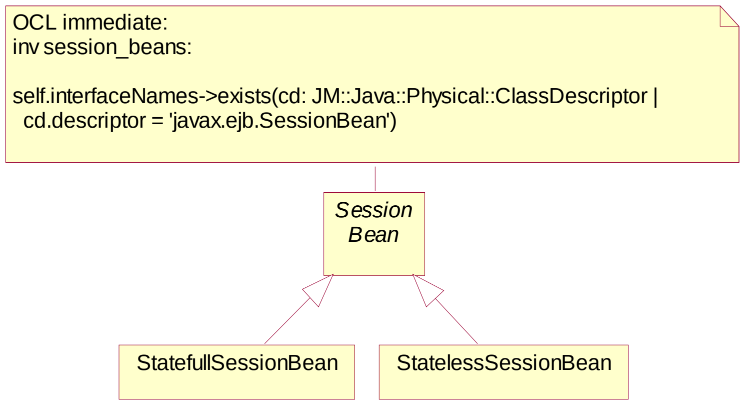


Figure 14 Complete hierarchy of enterprise beans

In the next step the specific characteristics of the enterprise beans have to be expressed. The Object Constrained Language OCL is used for this purpose. Figure 15 shows that a session bean has to implement the javax.ejb.SessionBean class.



```
OCL immediate:
inv session_beans:

self.interfaceNames->exists(cd: JM::Java::Physical::ClassDescriptor |
  cd.descriptor = 'javax.ejb.SessionBean')
```

Figure 15 Definition of a session bean

Similar definitions can be given for entity beans, see Figure 16, and for message driven beans.

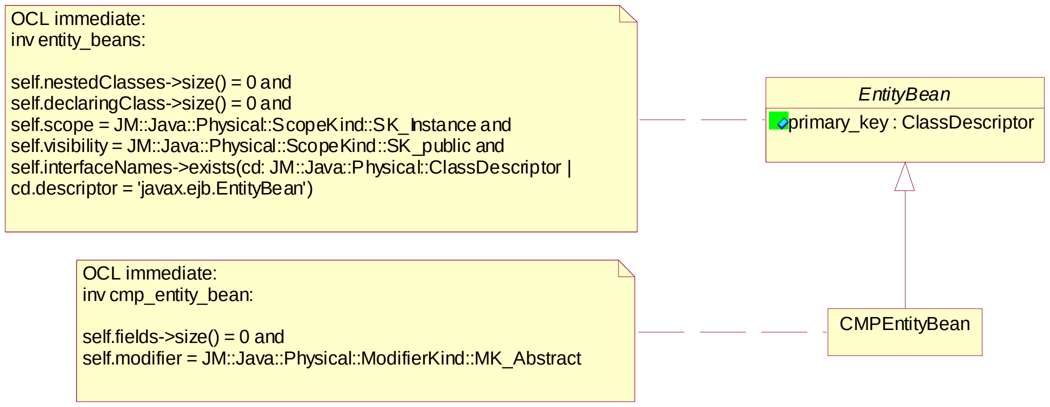

OCL immediate:
inv entity_beans:
self.nestedClasses->size() = 0 and
self.declaringClass->size() = 0 and
self.scope = JM::Java::Physical::ScopeKind::SK_Instance and
self.visibility = JM::Java::Physical::ScopeKind::SK_public and
self.interfaceNames->exists(cd: JM::Java::Physical::ClassDescriptor |
cd.descriptor = 'javax.ejb.EntityBean')
EntityBean
primary_key : ClassDescriptor
OCL immediate:
inv cmp_entity_bean:
self.fields->size() = 0 and
self.modifier = JM::Java::Physical::ModifierKind::MK_Abstract
CMPEntityBean

Figure 16 Definition of an entity bean

The specification of the J2EE artefacts that occur in the client view can be refined accordingly, as is shown in Figure 17.

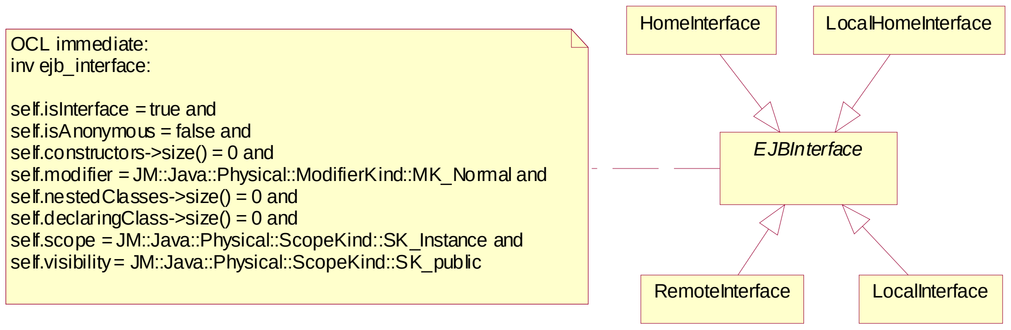

HomeInterface
LocalHomeInterface
OCL immediate:
inv ejb_interface:
self.isInterface = true and
self.isAnonymous = false and
self.constructors->size() = 0 and
self.modifier = JM::Java::Physical::ModifierKind::MK_Normal and
self.nestedClasses->size() = 0 and
self.declaringClass->size() = 0 and
self.scope = JM::Java::Physical::ScopeKind::SK_Instance and
self.visibility = JM::Java::Physical::ScopeKind::SK_public
EJBInterface
RemoteInterface
LocalInterface

Figure 17 Definition of EJB interfaces

6.6.2. Rules Specification

The mapping of EDOC to J2EE follows these principles:

- Entities become entity beans and the entity data are used to define the enterprise bean's view of data in the database as well as the abstract persistence schemas in the deployment descriptor.
- The process components become session beans or message driven beans.
- A data manager is mapped to a POJO Java class.
- The ports are mapped to the interface and methods of the enterprise bean.

For the transformation from EDOC modelling elements to J2EE modelling elements some general rules hold:

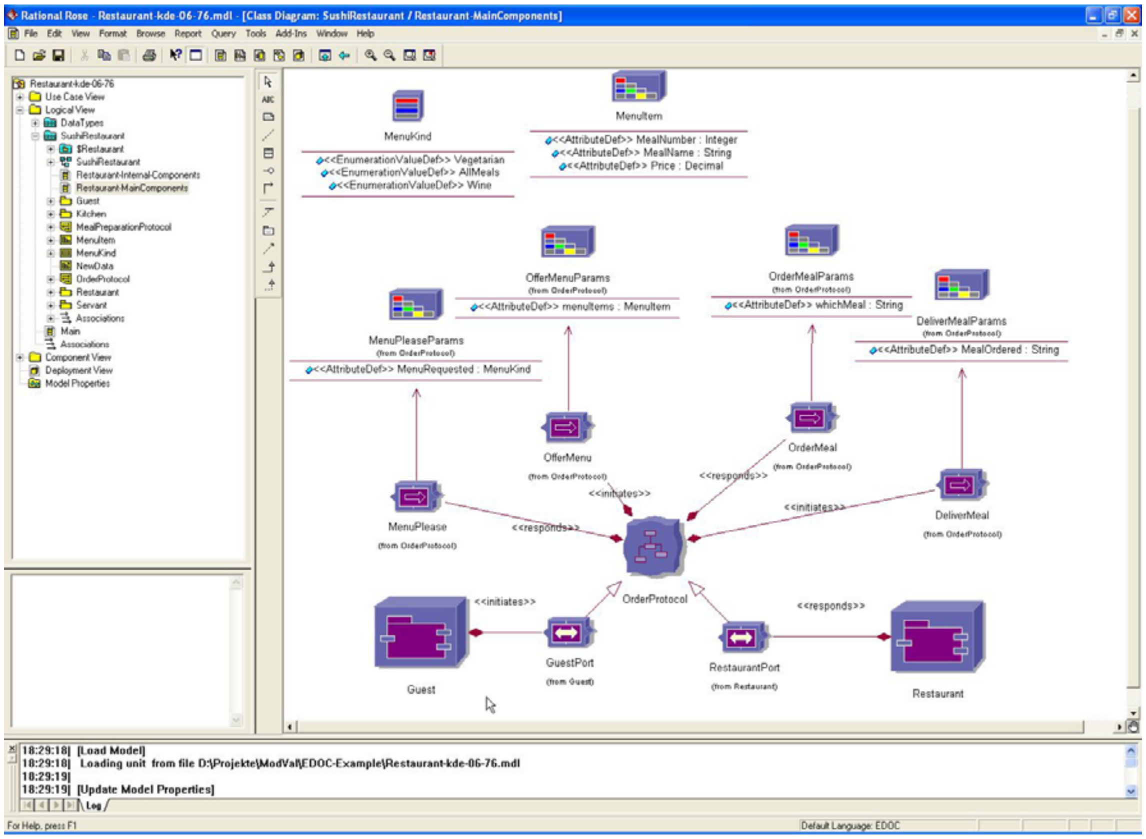
- Pure “client” components will be ignored. They are not transformed to enterprise beans.
- Each created enterprise bean has a home interface with the type specific default methods. Specific *create, finder,* and *select* methods are not created during this transformation. They will be created in the J2EE to Java transformation step.
- Flows (flow ports) are transformed to Java methods, message driven beans or JMS are currently not considered in the transformations.

- The transformation process starts with the most specialized EDOC modelling elements. For example an *Entity* that is a *DataManager* that is a *ProcessComponent* will be transformed according to the rules for *Entities*.

6.6.3. Typical Test Example

The rules specified here are currently not evaluated by a tool and are therefore not checked for completeness and correctness. But for illustrational purposes two screen shots are provided that demonstrate the source and the result of the transformation using the Fraunhofer tool chain.

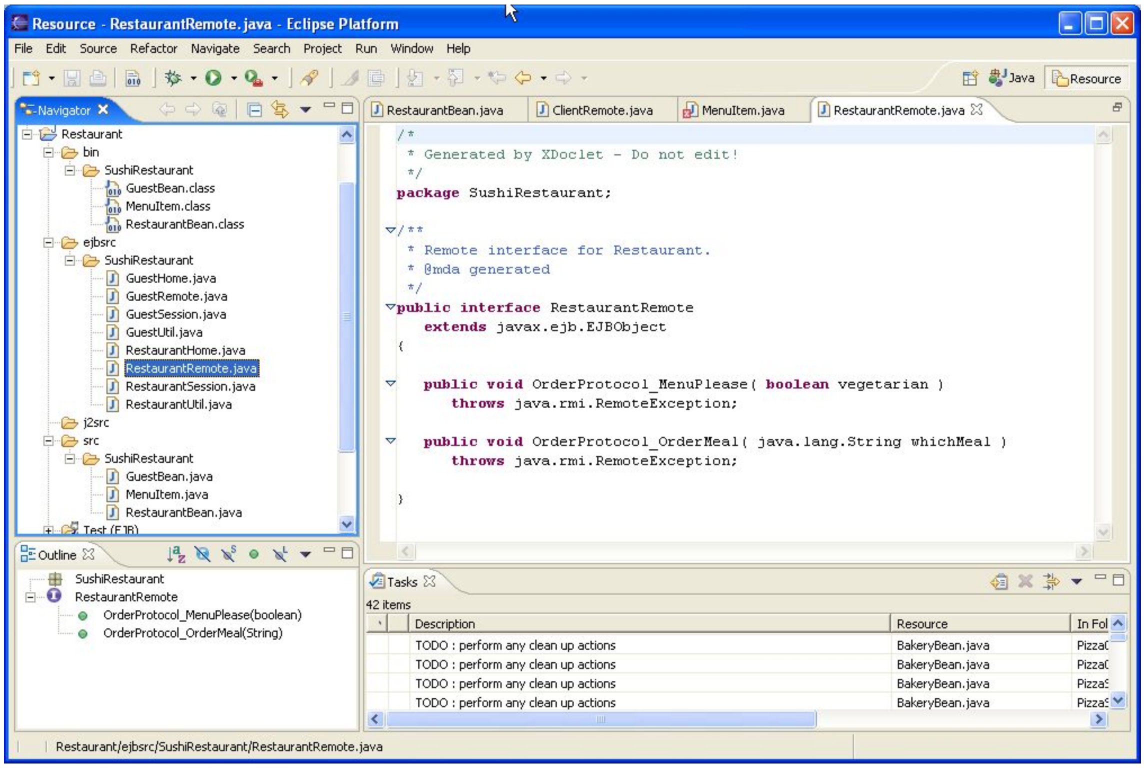
The first screen shot shows the EDOC Model that is used as source model. In the example a restaurant is modelled.


Rational Rose - Restaurant-kde-06-76.mdl - [Class Diagram: SushiRestaurant / Restaurant-MainComponents]
File Edit View Format Browse Report Query Tools Add-Ins Window Help
MenuKind
<<EnumerationValueDef>> Vegetarian
<<EnumerationValueDef>> AllMeals
<<EnumerationValueDef>> Wine
MenuItem
<<AttributeDef>> MealNumber : Integer
<<AttributeDef>> MealName : String
<<AttributeDef>> Price : Decimal
OfferMenuParams
<<AttributeDef>> menuItems : MenuItem
OrderMealParams
<<AttributeDef>> whichMeal : String
DeliverMealParams
<<AttributeDef>> MealOrdered : String
MenuPleaseParams
<<AttributeDef>> MenuRequested : MenuKind
OfferMenu
OrderMeal
MenuPlease
DeliverMeal
OrderProtocol
<<initiates>>
<<responds>>
Guest
GuestPort
RestaurantPort
Restaurant
For Help, press F1
Default Language: EDOC

```
18:29:18|  [Load Model]
18:29:18|   Loading unit  from file D:\Projekte\ModVal\EDOC-Example\Restaurant-kde-06-76.mdl
18:29:19|
18:29:19|  [Update Model Properties]
```

Figure 18 EDOC Model of the restaurant

The second screen shot illustrates the result of the transformation. It shows the ECLIPSE IDE with a project that includes the Java classes resulting from the transformation.



```
/*
 * Generated by XDoclet - Do not edit!
 */
package SushiRestaurant;

/**
 * Remote interface for Restaurant.
 * @mda generated
 */
public interface RestaurantRemote
   extends javax.ejb.EJBObject
{

   public void OrderProtocol_MenuPlease( boolean vegetarian )
      throws java.rmi.RemoteException;

   public void OrderProtocol_OrderMeal( java.lang.String whichMeal )
      throws java.rmi.RemoteException;

}
```

	Description	Resource	In Fol
	TODO : perform any clean up actions	BakeryBean.java	PizzaC
	TODO : perform any clean up actions	BakeryBean.java	PizzaC
	TODO : perform any clean up actions	BakeryBean.java	PizzaS
	TODO : perform any clean up actions	BakeryBean.java	PizzaS

Figure 19 Eclipse project with Java classes of the restaurant

6.6.4. Definition using MergeQVT

In the following the rules are defined by using MergeQVT/Mappings. To illustrate the rules they are headed by semi-formal textual notation.

6.6.4.1. Top level rule to invoke all rules

We provide here the transformation definition and the entry "main" mapping rule. Note that in QVT/Mapping rule invocation is explicit. Even if we define all the rules without no ordering assumption we need to write at least one top-level rule to execute all the other rules. In order for the "resolve" rule to work we need to create the target instances before retrieving them. To ensure this, the main mapping executes in two passes.

QVT:

```
module Edoc_To_J2EE (in edocModel:EDOC): j2eeModel:J2EE;
main () {
  edocModel.objects->firstPass();
  edocModel.objects->secondPass();
}
mapping firstPass(in EDOC::ModelElement) : JavaElement
  disjuncts Package_to_Package, EDOC_ProcessComponent_To_Java_Interface {}
mapping secondPass(in EDOC::ModelElement) : JavaElement
  disjuncts
    PackageContainement,
    FlowPort_To_Method,
    Protocol_FlowPort_To_Method,
    OperationPort_To_Method,
    Protocol_OperationPort_To_Method,
    InitiatingFlowPort_of_OperationPort_To_Field,
    SharedProcessComponent_To_RI_EnterpriseBean,
    OtherProcessComponent_To_ImplClass
{}
```

6.6.4.2. Package to Package

Semi-formal text:

For each EDOC Package create a Java package with the same name.

QVT:

```
mapping Package_to_Package [in EDOC::PackageDef] () : J2EE::JavaPackage {
      name := self.name;
}
```

6.6.4.3. Package to Package with containment

Note: This rule is not complete due to a lack of understanding. (Needs to by fixed with help of an QVT expert.)

Semi-formal text:

```
For each EDOC package P1
      Find the corresponding Java package J1
            For each ownedElement of P1 OEP1
                  Find the corresponding JavaPackage JO1
                  Set JO1.outerpackage to J1 and add JO1 to J1.subPackages
```

QVT:

```
mapping PackageContainement [in EDOC.PackageDef] () : J2EE.JavaPackage {
  init {
    var result := self.resolveone(J2EE.JavaPackage);
  }
  subPackages := self.ownedElement[EDOC::PackageDef]
     ->resolveone(J2EE.JavaPackage);
}
```

6.6.4.4. ProcessComponent to Java Interface

Semi-formal text:

```
For each EDOC.ProcessComponent PC
      Find JavaPkg jpkg corresponding to the PC.namespaceContainer
                  (remark: the EDOCPkg containing PC)
            Create a JavaClass JC and set
                  JC.name = PC.name
                  JC.isInterface = true
                  JC.javaPackage = jpkg
```

QVT:

```
mapping EDOC_ProcessComponent_To_Java_Interface
 [in EDOC.ProcessComponentDef] () : J2EE.JavaClass {
      name := self.name;
      isInterface := true;
      javaPackage := self.namespaceContained.resolveone(J2EE.JavaPackage);
}
```

6.6.4.5. ProcessComponent Flowport to Method

Semi-formal text:

```
For each Flowport fp with direction Responds and isSynchronous = true
    When owner is ProcessComponent or Protocol
        Find the Interface jc realizing the owner
        Create a method m with
            m.visibility = public
            m.name = fp.name
            m.declaringClass = jc
```

QVT:

```
mapping FlowPort_To_Method [in EDOC::FlowPortDef] () : J2EE.Method
  guard
    (self.direction = "Responds" and self.isSynchronous=true) and
     (self.the_owner.isKindOf(EDOC::ProcessComponent) or
      self.the_owner.isKindOf(EDOC::ProtocolDef))
{
  visibility := “public”;
  name := self.name;
  declaringClass := self.the_owner.resolveone(J2EE.JavaClass);
}
```

6.6.4.6. ProtocolPort FlowPort to Method

Semi-formal text:

```
For each ProtocolPortDef pp
    For each FlowPortDef  fp
        When (fp.the_owner = pp.the_protocol, fp.direction = pp.direction)
            Find j2ee.interface jc that corresponds to pp.the_owner
            Create j2ee.method m with
                m.visibility = public,
                m.declaringClass = jc,
                m.name = append(pp.name, append(“_”, fp.name)
```

QVT:

```
mapping Protocol_FlowPort_To_Method
  (in pp:EDOC::ProtocolPortDef, in fp:EDOC::FlowPortDef) : J2EE.Method
      guard match (fpo, fpd, ppp, ppd)
        fp:{the_owner = fpo, direction = fpd },
        pp:{the_protocol = ppp, direction = ppd}
        when { fpo = ppp, fpd = ppd}
{
  visibility := “Public”;
  name := pp.name.concat("_").concat(fp.name);
  declaringClass :=  pp.the_owner.resolveone(J2EE.JavaClass);
}
```

6.6.4.7. ProcessComponent OperationPort to Method

Semi-formal text:

```
For each OperationPortDef  op
      With op.direction = Responds
            Find j2ee.interface owning_jc corresponding to the_owner of op
            Create  j2ee.JavaClass ret_jc with
                  ret_jc.name = append(op.name, “Return”)
                  ret_jc.isInterface = false
            Create j2ee.Method m with
                  m.name = op.name
                  m.declaringClass =  owning_jc
```

QVT:

```
mapping OperationPort_To_Method [in EDOC::OperationPortDef] ()
 : jm : J2EE::Method, jc : J2EE::JavaClass
{
   out jc:J2EE::JavaClass {
     name := self.nameconcat(“Return”);
     isInterface := false;
   }
   out jm:J2EE::Method {
      jm.name := self.name;
      jm.declaringClass := resolveone(self.the_owner);
}
```

6.6.4.8. Protocol OperationPort to Method

Semi-formal text:

```
For each OperationPortDef op
      For each ProtocolPortDef pp
            For each ProcessComponentDef pc
                  when ( pc = pp.the_owner,
                        op.the_owner = pp.the_protocol and
                        op.direction = pp.direction )
                  Find JavaClass owning_jc corresponding to pc
                  create j2ee.JavaClass ret_jc with
                        ret_jc.name = append(op.name,”Return”)
                        ret_jc.isInterface = false
                  create j2ee.Method m with
                        m.name = append(pp.name,append(“_”,op.name)
                        m.declaringClass = owning_jc,
```

QVT:

```
mapping Protocol_OperationPort_To_Method
  (in op : EDOC::OperationPortDef,
   in pp : EDOC::ProtocolPortDef,
   in pc : EDOC::ProcessComponentDef)
  :jc : J2EE::JavaClass, jm : J2EE.Method
 guard match (ppo, opo, ppp, opd, ppd)
    pp:{the_owner = ppo, direction = ppd, the_protocol = ppp},
    op:{the_owner = opo, direction = opd]
    when { pc = ppo and opo = ppp and opd = ppd}
{
   out jc: J2EE::JavaClass {
     name := op.name.concat(”Return”);
     isInterface := false;
   };
   out jm : J2EE.Method {
     name := pp.name.concat(“_”).concat(op.name));
     declaringClass := pc.resolveone(J2EE.JavaClass);
   };
}
```

6.6.4.9. *Initiates Flowports owned by Operation Ports to Fields of method*

Semi-formal text:

```
For each FlowPortDef fp
      when (fp.direction = “Initiates” and type of fp.the_owner = OperationPortDef)
      Find the corresponding JavaClass jc of fp.the_owner
      Create J2EEField fld with
            fld.name = fp,name
            fld.declaringClass = jc
            fld.isFinal = true
            fld.isVolatile = true
```

QVT:

```
mapping InitiatingFlowPort_of_OperationPort_To_Field
  [in EDOC.FlowPortDef] () : J2EE.Field
 guard match (fpo) self:{the_owner = fpo}
       when { fpo.isKindOf(EDOC.OperationPortDef) }
{
   name := self.name;
   declaringClass := fpo.resolveone(J2EE.JavaClass);
   isFinal := true;
   isVolatile := true;
}
```

6.6.4.10. *“Shared” ProcessComponent to EnterpriseBean with Remote Interface*

Semi-formal text:

```
For each  ProcessComponentDef pc
      when (pc.granularity = “Shared” or pc.granularity = “Program”)
      Create J2EE.RemoteInterface ri
      Create StateLessSessionBean slsb
            with
                  slsb.name = append(pc.name,”Session”)
                  slsb.remote_interface = ri
```

QVT:

```
mapping SharedProcessComponent_To_RI_EnterpriseBean
  [in EDOC.ProcesscomponentDef]() :
    ri:J2EERemoteInterface,
    ssb:J2EE.StateLessSessionBean
guard match (pcg)
      self:EDOC.ProcessComponentDef { granularity = pcg }
      when { pcg = “Shared” or pcg = “Program”}
{
   out ssb: J2EE.StateLessSessionBean {
     name := append(self.name, “Session”);
     remote_interface := ri;
   };
}
```

6.6.4.11. “Other” ProcessComponents to implementing class

Semi-formal text:

```
For each ProcessComponentDef pc
      when (NOT (pc.granularity = “Shared”  OR pc.granularity =”Program”))
      Create J2EE.JavaClass class with
            class.name = append(pc.name,”Impl”)
            class.isInterface = false
```

QVT:

```
mapping OtherProcessComponent_To_ImplClass
 [in EDOC.ProcessComponentDef] () : J2EE.JavaClass
      match (pcg) self:EDOC.ProcessComponentDef { granularity = pcg}
      when { not (pcg = “shared” or “pcg = “Program”) }
{
  name := self.name.concat(“Impl”);
  isInterface := false;
}
```

6.6.5. Discussion

Table 8 Evaluation of example-dependant properties : EDOC → J2EE

Criteria	Scale Measure	Absolute	Weight range = [1,6]	Comments	Score (normalized measure * weight)

Ease of use in complex transformations (MAPPINGS – TEXTUAL)	3 = Agree	No	6	The code is nicely separated in different, understandable rules. At the time being this example is not fully defined so it is hard to evaluate this example fully.	4,5

At the current stage the rules defined with QVT for the transformation of EDOC to J2EE could not be checked by using a tool. Most likely the list of rules is incomplete or some rules are possibly incorrect.

Furthermore, the usage of the tool could help to improve the understanding of the nature of QVT and to define proper QVT rules. This would also help to learn the language more easily and to use it in an efficient way.

7. Summary of the QVT Merge Example-Based Evaluation

This section summarizes the evaluation of the QVT Merge language on the most important criteria, *ease of use*, which also is among the hardest to come up with an objective measurement of. The table below shows the average-based score for ease of use calculated from the examples. Based on the examples, the QVT Merge language scores a bit higher for complex than for simple transformations and vertical+structural transformations gets a lower score than the other categories of transformations. We need more discussion and more examples in order to show that these trends are valid in general. But the overall average ease of use is evaluated as approximately 2.5. This is half way between *neither easy to use* and *agree easy to use*. This is based on eight different transformation examples (two of the transformation examples have alternative definitions as graphical/textual and relations/mappings) and has thus quite strong reliability.

Table 9 Is the transformation language easy to use? (0= Strongly disagree, 1 = Disagree, 2 = Neither, 3 = Agree, 4 = Strongly agree)

Example	**simple** max=4	**complex** max = 4	**Vertical and structural** max = 4	**Vertical and behavioural** max = 4	**Horiz. and structural** max = 4	**Horiz. and behavioural** max = 4	**Score simple** max=6	**Score complex** max=6
Example 1	2							
Example 2		3						
Example 3		3						
Example 4	1,5							
Example 5	3							
Example 6		3						
Average	2,17	3	1,75	3	3	3	3,3	4,5

When reviewing the example transformations some negative findings were discovered that may be used to further improve the specification before it is finalized as an OMG adopted specification:

- It is confusing when to use arrow and when to use dot for referencing part attributes/associations, built-in functions, inherited OCL functions etc.
- There is a mixture of procedural style with object-oriented style when defining and invoking methods. Object method calls are object-oriented (`theXSLTRoot.P2P`), while the signature uses an input parameter to represent the object type on which we can invoke the method like in the code extract signature above. This makes it non-intuitive to understand the much used "self" keyword that refers to the context parameter.
- It is hard to discover calls to the mappings rules. When doing transformations it is crucial to easily see where calls are made recursively or to other mapping rules. These calls cannot easily be distinguished from other calls to built-in functions, attribute/association references or OCL functions. XSLT has a solution for this by letting all calls to other mapping rules happen with the apply-templates instructions.

In addition to the negative findings described above, some issues were controversial because there were different opinions in the review group if the issues are negative findings or not:

- **Long and cryptic expressions**. Single expressions are sometimes very long and cryptic to understand which requires a lot of mental effort. (Example: `return := **out** Return { expressions := self.nodes[#Template][t|t.match = '/']->nodes->flatten()->NodeToExpression();`) This is a heritage of OCL style and syntax. QVTMerge introduces additional short-hands to avoid excessive verbosity in single expressions – like the '#MyType' expression mapped as a call to the 'oclIsKindOf(MyType)' pre-defined operation . It is not clear yet whether these additional short-hands help on ease-of-use of the language. It is also possible for a transformation writer to split a computation in various lines using intermediate variables.

- **Two-pass**. Some of the transformations use a two-pass approach in order to ensure that some target instances are produced so that the `resolve()` methods will get the proper element in a different context. This is a consequence of the explicit execution strategy in QVTMerge/Mappings which might be perceived as an advantage or as a disadvantage depending on writer preferences. An interesting issue here is to know whether it is possible to handle automatically object resolutions - so that the language user does not need to worry about this – without loosing the advantages of the explicit execution strategy.

The review of all the code examples shows nice program code structure, inheritance, and modularity by separation into manageable mapping rules. We believe that reusability and maintenance will be positive side-effects when the transformation code is written as they were in the examples. The example-based ease-of-use evaluation of the QVTMerge language shows slightly higher scores for complex than for simple transformations and the combination of vertical and structural transformations gets a lower score than the other categories of transformations. We need more examples in order to show that these trends are valid in general. But the overall average ease-of-use is evaluated as approximately 2.5 on a scale from 0 to 4, where 4 is the goal. It should be stressed that the evaluation of ease-of-use are subjective judgments of the MODELWARE participants who performed the example-based testing.

8. Evaluation of QVT Compuware/Sun

Our Modelware evaluation framework has also been applied to the QVT Compuware/Sun submission to compare it with the results of the QVT Merge evaluation. It is important to note that the evaluation of the QVT Compuware/Sun is carried out on a single reference example only.

8.1. QVT Compuware/Sun Language-Based Evaluation

The criteria that can be evaluated by manual inspection of the language itself and that does not depend on a tool or on observation in examples are presented in the following table.

Table 10 Evaluation of QVT Compuware/Sun language-dependant properties

Criterion	How it is supported by QVT Compuware/Sun	M	S
Traceability	This is not part of the language, and thus it becomes a tool issue. A compliant tool may have no support for traceability. Note: Violation of an absolute criterion.	0 (2)	0 (5)
Unidirectionality	Supported.	1 (1)	4 (4)
Complete textual notation	Supported.	1 (1)	4 (4)
Black-box interoperability	Not supported. Note: Violation of an absolute criterion.	0 (1)	0 (4)
Composition of transformations	Not supported. Note: Violation of an absolute criterion.	0 (2)	0 (3)
QoS mapping	Source and target can be expressed as MOF models and we believe that QVT Compuware/Sun can be used to transform between any MOF models.	1 (1)	3 (3)
Graphical notation	There is no evidence in the submission that the graphical notation can be used to fully define any transformation that can be defined textually.	1 (3)	0.7 (2)
Updating source model(s)	Supported	1 (1)	2 (2)
Resolution of QoS properties	Source and target can be expressed as MOF models and we believe that QVT Compuware/Sun can be used to transform between any MOF models.	1 (1)	2 (2)

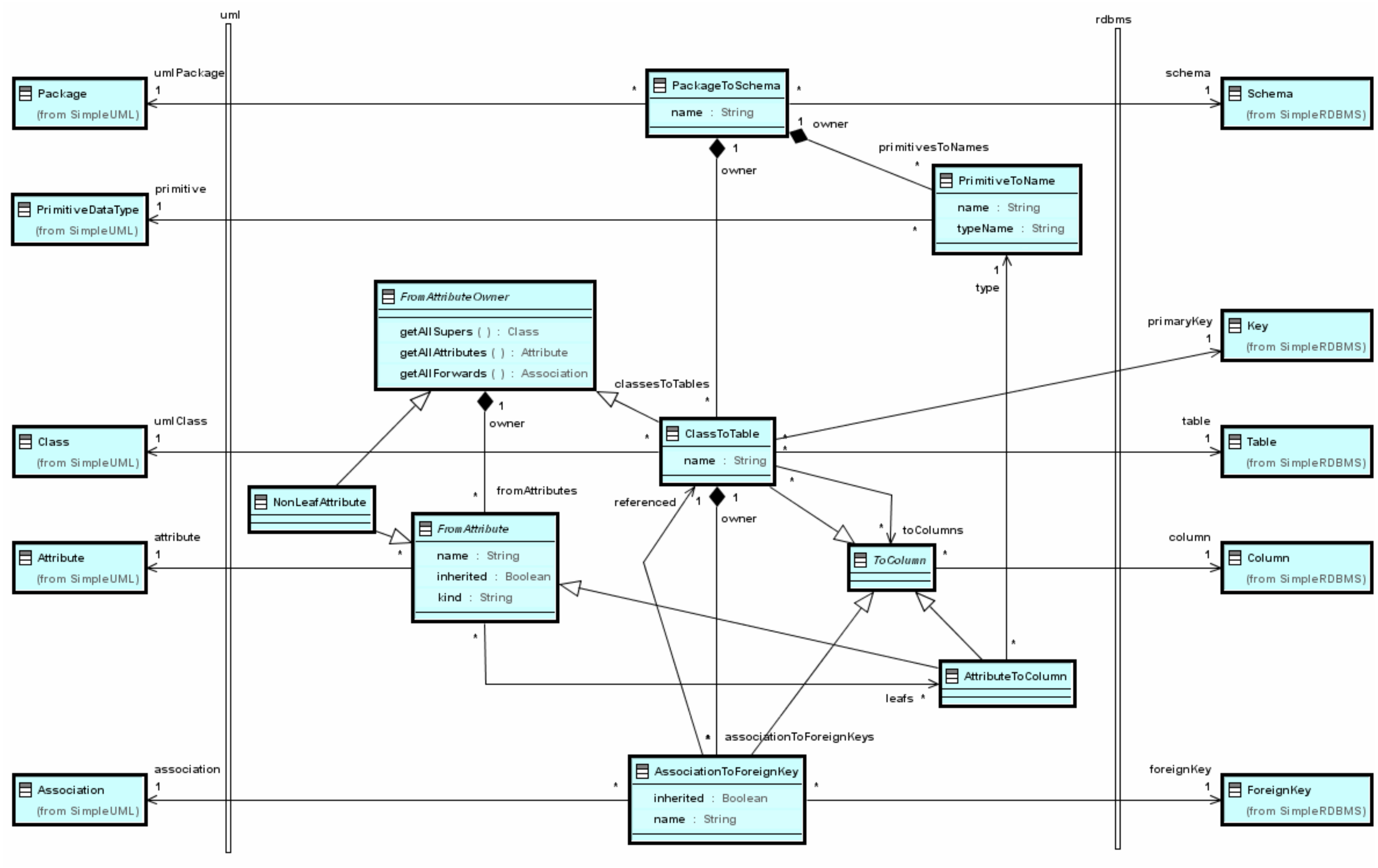
Incomplete transformations completed with pattern parameters	Not evaluated.	-	-
Modularity	Supported by grouping into UML packages	1 (1)	6 (6)
Reusability	Full support.	2 (2)	5 (5)
Restricting conditions/pre-conditions	OCL expressions can be used to restrict the source model(s)	1 (1)	4 (4)
Object orientation	Inheritance and encapsulation is supported. Identity and polymorphism is considered not supported.	2 (4)	1.5 (3)
Bidirectionality	Supported.	1 (1)	2 (2)
Multiple source models	Supported	1 (1)	2 (2)
Learning Curve	Measurement: 3 = Agree. The conciseness of the specification and the reuse of UML, MOF and OCL with very few extensions make it easy to learn this language. The disadvantage is the lack of examples and explaining of some of the syntax used.	3 (4)	1.5 (2)
Multiple target models	It seems likely that the user can textually define several target models. It is more unclear how this can be achieved graphically.	1 (1)	1 (1)
Constraints between rules	Rule ordering and pre-/postconditions can be specified.	1 (1)	1 (1)
Repetitiveness	Not been tested since this requires a lot of work on building a large proof based on the entire language specification.	-	-
	TOTAL	19 (29)	40 (55)

Note that since there are three violated absolute criteria the total score is assigned to 0, but the total summation is included so that it is easier to compare the evaluation of Compuware/Sun with QVTMerge.

8.2. QVT Compuware/Sun Example-Based Evaluation

The only example evaluated at this stage is the example provided by the current submission which is the UML to RDBMS example. The full description of this example with source and target metamodels will not be repeated here since it is also given in 6.4. In this section we will only present the transformation definition copied from the submission and the final evaluation table.

8.3. Definition using QVT Compuware/Sun



The diagram above is a class model of our object-relational transformation definition. It is a standard class model as any other UML-infrastructure based class model, with exception of the direction declarations in it. The directions are depicted according to the standard diagram-notation extension of EXMOF. You can easily see which properties are bundled in the *uml* direction, and which are bundled in the *rdbms* direction.

The class model defines the structural aspects of the transformation. The classes and properties do imply neither behavior nor side effects. However, they do define which mapping structures between simple UML and simple RDBMS are well formed.

For example, we can see that one package can be mapped to one schema and that one class can be mapped to one table, one (primary) key and one column (because ClassToTable specializes ToColumn).

Note that the above model does not imply that all instances of *class* are mapped to tables and primary keys. The actual rules that define the derivations can limit the amount of instances that are actually transformed.

The classes FromAttribute, FromAttributeOwner, AttributeToCollumn and NonLeafAttribute define the structure used to flatten the complex data types. These classes have a recursive structure to support the recursive flattening of complex data types containing attributes with complex data types.

All classes of UMLTORDBMS except PackageToSchema are derived classes. A transformation can be executed when an instance of PackageToSchema exists, whose properties *umlPackage* and *schema* have a value.

The attribute *typeName* should be given a value for each instance of PrimitiveToName to define the primitive-data-type marshaling.

The following EXMOF code defines the package UMLTORDBMS including all the EXMOF rules that define the derivation of our OO and relational language:

```
-- A Transformation definition from SimpleUML to SimpleRDBMS

package UMLTORDBMS imports SimpleUML, SimpleRDBMS {

  direction uml uses SimpleUML;
  direction rdbms uses SimpleRDBMS;

  -- Primitive data type marshaling
  class PrimitiveToName {
    owner : PackageToSchema opposites primitivesToNames;


    -- uml
    primitive : PrimitiveDataType to uml;


    -- rdbms
    typeName : String to rdbms;
  }

  class PackageToSchema {
    composite classesToTables : Set(ClassToTable) opposites owner;

    composite primitivesToNames : Set(PrimitiveToName) opposites                owner;

    name : String;
    -- uml of PackageToSchema
    umlPackage : Package to uml;

        name :=: umlPackage.name;

        map primitivesToNames[compose pn] :#
                umlPackage.elements[prim:PrimitiveDataType]
    {
      pn.primitive :== prim;
     }
    map classesToTables[compose c2t] :#:
                umlPackage.elements[compose cls:Class|kind='persistent']
    {
      c2t.umlClass :== cls;
    }
    map classesToTables.associationToForeignKeys[a2f|not inherited]
                #: umlPackage.elements[compose assoc:Association|
                 source.kind='persistent' and destination.kind='persistent']
    {
      a2f.association        :== assoc;
      a2f.owner.umlClass      =: source;
      a2f.referenced.umlClass =: destination;
      a2f.name                =: name;
    }


    -- rdbms of PackageToSchema
    schema : Schema to rdbms;

    schema.name :=: name;

    map schema.tables[compose tbl|kind<>'meta'] :#:
                classesToTables[compose c2t]
    {
      tbl ==: c2t.table;
    }
}

abstract class FromAttributeOwner {
    composite fromAttributes : Set(FromAttribute) opposites owner;


    -- uml
    getAllSupers(cls : Class) : Set(Class) {
      cls.general->collect(gen|
                      self.getAllSupers(gen))->including(cls)->asSet()
    }
    getAllAttributes(cls : Class) : Set(Attribute) {
      getAllSupers(cls).attribute
    }
    getAllForwards(cls : Class) : Set(Association) {
      getAllSupers(cls).forward
    }
}
```

```
class ClassToTable extends FromAttributeOwner, ToColumn {
   owner : PackageToSchema opposites classesToTables;

   composite associationToForeignKeys :
                 OrderedSet(AssociationToForeignKey) opposites owner;

   name : String;

   -- all columns are mapped via the following property
   toColumns : OrderedSet(ToColumn) :=
      OrderedSet(ToColumn){self}->union(fromAttributes.leafs)->
         union(associationToForeignKeys);


   -- uml of ClassToTable
   umlClass : Class to uml;

        umlClass.name :=: name;

        map self.associationToForeignKeys[compose a2f] :#
      self.getAllForwards(umlClass)[assoc] #
      ClassToTable->allInstances()[dest]
        {
      self.getAllSupers(c2t.umlClass)
                        ->includes(assoc.destination);

        a2f.association :== assoc;

      a2f.referenced  :== dest;
      a2f.inherited   :=  assoc.source<>self.umlClass or
                      assoc.destination<>dest.umlClass;
   }
   map fromAttributes[compose a2c:AttributeToColumn] :#
      getAllAttributes(umlClass)[attr] #
      PrimitiveToName->allInstances()[p]
   {
      attr.type      =  p.primitive;

      a2c.attribute :== attr;
      a2c.inherited :=  attr.owner<>self.umlClass;
      a2c.name            :=  name;
   }
   map fromAttributes[compose nla:NonLeafAttribute] :#
      getAllAttributes(umlClass)[attr|type.oclIsKindOf(Class)]
   {
      nla.attribute :== attr;
      nla.name      :=  attr.name;
   }
   map fromAttributes[a2c:AttributeToColumn|not inherited] #:
          umlClass.attribute[compose attr|
                   type.oclIsKindOf(PrimitiveDataType)
        {
      a2c.attribute :== attr;
      a2c.name       =: attr.name;
   }
  -- rdbms of ClassToTable
   table : Table to rdbms;

   primaryKey : Key to rdbms;

   table.name   :=: name;
        table.kind   :=  'base';
   column        :=  primaryKey.column->first();
   column.key   :=  Set(Key) {self.primaryKey};
   column.name  :=  self.name+'_tid';
   column.type  :=  'NUMBER';

   map table.column[compose c|
                   not foreignKey.refersTo.owner.kind->includes('meta')] :#
                                                 toColumns[tc]
   {
      c ==: tc.column;
   }
        -- map columns from rdbms, except columns that originated from
        -- inherited and complex-data-type attributes
   map table.column[c|key->isEmpty() and foreignKey->isEmpty() and
         self.fromAttributes->select(fa|fa.inherited or
            fa.oclIsKindOf(NonLeafAttribute)).
               leafs.column->excludes(c)] #:
                 self.fromAttributes[compose a2c:AttributeToColumn|
                    not inherited]
   {
      c                  ==: a2c.column;
      a2c.inherited :=   false;
   }
   map table.foreignKey[compose fk|
         not refersTo.owner.kind->includes('meta')] :#
      associationToForeignKeys[a2f]
   {
      fk ==: a2f.foreignKey;
   }
        -- map foreign keys from rdbms, except foreign keys that
        -- originated from inherited associations
   map table.foreignKey[fk|self.associationToForeignKeys->
         select(af|af.inherited).foreignKey->excludes(fk)] #:
      associationToForeignKeys [compose a2f|not inherited] #
      ClassToTable->allInstances()[c2t]
   {
      fk.refersTo.owner = c2t.table;

      fk     ==: a2f.foreignKey;
      c2t    ==: a2f.referenced;
      false  =:  a2f.inherited;
   }
   map table.key[compose pk] :# self[c2t]
   {
      pk       ==: c2t.primaryKey;
      pk.kind :=   'primary';
      pk.name :=   c2t.name+'_pk';
    }

 } -- end of class ClassToTable
```

```
abstract class FromAttribute {
    name : String;

    kind : String;

    owner : FromAttributeOwner opposites fromAttributes;

    leafs : Set(AttributeToColumn);

    inherited : Boolean;


    -- uml
    attribute : Attribute to uml;

    kind :=: attribute.kind;
}

abstract class ToColumn {

    -- SimpleRdbms
    column : Column to rdbms;
}

class AttributeToColumn extends FromAttribute, ToColumn {
    type : PrimitiveToName;

    leafs := Set(AttributeToColumn) {self};


    -- uml
    attribute.type := type.primitive;

    map type[t] :# attribute.type[at]
    {
        t.primitive = at;
    }


    -- rdbms
    column.name  :=:  name;
    column.kind  :=:  kind;
    column.type  :=   type.typeName;

    map type[t] :# column.type[ct]
    {
        t.typeName = ct;
    }
}
```

```
    class NonLeafAttribute extends FromAttributeOwner, FromAttribute {
       leafs := fromAttributes.leafs;


       -- uml
       map fromAttributes[compose a2c:AttributeToColumn] :#
          getAllAttributes(attribute.type.oclAsType(Class))[attr] #
          PrimitiveToName->allInstances()[p]
       {
          a2c.attribute :== attr;
          attr.type      =  p.primitive;
          a2c.name       :=  self.name+'_'+name;
       }
       map fromAttributes[compose nla:NonLeafAttribute] :#
                    getAllAttributes(attribute.type.oclAsType(Class))[attr]
       {
          attr.type.oclIsKindOf(Class);

          nla.attribute :== attr;
            nla.name      :=  self.name+'_'+name;

       }
    }

    class AssociationToForeignKey extends ToColumn {
       referenced : ClassToTable;

       owner : ClassToTable opposites associationToForeignKeys;

       name : String;

       inherited : Boolean;


       -- uml
       association : Association to uml;

       name := if (not inherited)
             then association.name
             else association.name+'_'+referenced.umlClass.name
                    endif;


       -- rdbms
       foreignKey : ForeignKey to rdbms;

       foreignKey.refersTo :=  referenced.primaryKey;
       foreignKey.name       :=: name;
       column                :=  foreignKey.column->first();
       column.foreignKey   :=  Set(ForeignKey) {self.foreignKey};
       column.name           :=  self.name+'_tid';
        column.type                          :=  'NUMBER';

    }


} –- end of package UMLTORDBMS
```

8.4. Discussion

Only the textual approach is evaluated since the diagrammatic syntax cannot be used to define the complete transformation or this definition is not given in the submission.

Table 11 Evaluation of example-dependant properties: UML → RDBMS

Criteria	Scale Measure	Absolute	Weight range = [1,6]	Comments	Score (normalized measure * weight)
Ease of use in simple transformations (TEXTUAL)	1 = Disagree	No	6	The textual syntax is very hard to understand as stand-alone without referring extensively to the diagrammatic overview. The diagram information is also copied into the textual part and will thus need to be maintained two places. There is a large variety of ways to specify assignments using different mixtures of colon and equal sign, as well as a lot of different mixtures of colon and hash-symbol (#). The clear definition of all these as well as a justification for why all of these are needed is lacking in the current submission. The textual syntax uses many code lines to accomplish the task. The map fromAttributes is repeated three times. This seems bothersome and the code is also hard to read.	1,5

8.5. Summary of QVT Compuware/Sun Evaluation

The QVT Compuware/Sun Evaluation has the shortcoming that it has only been evaluated towards one single example. In addition the evaluator has become more familiar with the QVT Merge language by looking at more examples and also communication with a QVT Merge expert that helped to sort out some misunderstandings.

The specification reuses MOF, OCL and class diagrams so that very few new constructions are needed. This will give newcomers a low learning curve. Some more examples and clarification on the syntax definitions are needed to make the submission more understandable. Since a UML class structure is used to define the entire transformation, large and complex transformations will need a large number of mapping classes and a large number of inheritance and aggregation associations. A major concern is to see if this structure becomes too difficult to follow and maintain. Even for the quite small example of UML to RDBMS the class diagram becomes overloaded with relations and is quite hard to read. Furthermore it was not clear if the graphical notation could be used to define complete transformations. The reference example uses the graphical notation only as an incomplete overview of the transformation. Unfortunately the class structure definitions and its inheritance relations and aggregations need to be copied in the accompanying textual notation. This will make it hard to maintain consistency between the two unless there is some automatic tool support.

We have compared the Compuware/Sun submission with the MergeGroup submission. The Compuware/Sun approach violates three absolute criteria (composition, black-box and traceability), while the MergeGroup approach violates none of the absolute criterion tested. For the tested criteria the MergeGroup approach achieves a significantly higher score than Compuware/Sun. Furthermore we have compared the number of code lines for the textual notation of the two approaches for the UML to RDBMS example. 209 code lines are used to define the Compuware/Sun transformation and 59 code lines are used to define the same transformation using MergeGroup. Although counting code lines is a controversial quality rating, this is a clear indication that the Compuware/Sun approach requires much more effort to write a transformation.

9. Related Work

This report has identified a list of requirements for a model to model transformation tool that are important in the Modelware project. We will compare our requirements with those identified by other parties.

The QVT Request for Proposal (QVT RFP) [5] identified a list of required and optional requirements for submissions. Compared to Modelware some of their requirements are more focused on fitting the new QVT specification into the set of existing OMG specifications so to reuse and align well with existing recommendations and on the submission form. The Modelware requirements on the other hand are higher level, yet the reuse of successful existing recommendations may lead to good evaluation towards the Modelware requirements. The QVT RFP has identified portability and declarative transformation language in addition to the Modelware requirements. But it is unclear if a declarative transformation language implies that all parts of the language must be declarative or if hybrid languages are allowed where the user may use either declarative or imperative constructions. There are several Modelware requirements not mentioned in the QVT RFP: object-orientation, QoS support, composition of transformations, multiple source models, multiple target models, repetitiveness, black-box interoperability, modularity and user transparent rule ordering.

Gardner et. al [3] have reviewed the initial 8 submissions to the QVT RFP and proposed recommendations for the final specification. These recommendations are mostly implementation proposals rather than high level desired properties as this report focuses on. Gardner et. al propose a hybrid language with both declarative and imperative constructions, where the declarative part is simple and declarative as the only option for the querying part. They share Modelware's concern that ease of use and usability are critical requirements. In addition they also emphasize the importance of transformation consistency checking, composition, reuse and the ability to define complex transformations. Langlois et. al [4] have also investigated the 8 initial submissions, reviewed Garner et. al's contribution and compiled a list of recommendations based on the end-user experience of THALES. Langlois et al. have come up with four main criteria: portability, maintainability, usability and functionality. They stress the need for defining precise semantics for the transformation execution.

Sendall and Kazaczynski [9] proposes these desired properties: executability, efficiency, fully expressive and unambiguous, clear separation of source model selection rules from target producing rules, graphical constructs to complement a textual notation, composition of transformations, and "conditions under which the transformation is allowed to execute". They also propose that declarative constructions should be used for implicit mechanisms that are intuitive, but also warns that too many implicit and complicated constructs may be more difficult to understand than the more explicit and verbose counterpart.

This report has proposed an evaluation framework based on the requirements of the Modelware project. The focus of the requirements is to measure the goodness and quality of the approach regardless of any existing inheritance and compliance issues with existing OMG recommendations. The list of requirements at this level is more extensive than all of the previously published efforts. For each requirement we have specified a measurement scale, a weight for the importance of the requirement and if the requirement is absolute. Then the actual measurements are used as input to an overall score algorithm that can be used to compare the quality of different approaches to model to model transformation. We have also gone further than previous efforts by defining six reference examples to measure the ease of use requirement which is of uttermost importance but requires such case studies in order to be measured.

10. Conclusions

This report has identified 29 weighted evaluation criteria representing desired properties of a model to model transformation language. These criteria have then been used to evaluate the current QVT Merge specification. We have so far only been able to evaluate 21 of these criteria, mainly due to missing tool support. Some of the criteria are considered absolute in the sense that missing to fulfill such a criterion is considered a failure. The 21 evaluated criteria give a score of 59 out of a maximum possible score of 68 (language-based + example-based testing). We have also compared the QVT-Merge submission with the QVT-Compuware/Sun submission and at the time being the QVT-Merge seems to be the preferred one due to more support on the absolute criteria and better easy-to-use score.

Eight transformation examples for solving six different transformation tasks have given a lot insight on the ease of use criteria for both simple and complex transformations. The average score is 2,5 as an answer to the question: Is the transformation language easy to use? (0= Strongly disagree, 1 = Disagree, 2 = Neither, 3 = Agree, 4 = Strongly agree). The advantages are the modularity and nice structure of the program code into manageable separate transformation constraints and rules. Disadvantages are that there are many ways to define a transformation using either the relations or mappings, textual or graphical. Many different programming styles can be used and mixed including imperative, declarative, object-oriented and procedural. All these options require more effort to be skilled and it may cause messy code if used incautiously. The evaluator has also experienced difficulties interpreting some of the single statements that are very long and cryptic. Such expressions are commonly used and they require a lot of mental effort.

When defining transformations using QVT Merge we believe that a lot of effort may be required in order to define the source and target metamodels. Defining the metamodels will give a nice documentation of the transformation context. Repositories should be used to register metamodels so that they can be reused by others. The graphical notation has not been investigated enough but a hypothesis is that it is well suited for simple transformations and for providing a quicker and higher level view of the elements involved in the transformation. We strongly encourage that a fully defined bidirectional transformation be defined between the graphical notation and the textual notation, and then implemented in a QVT tool so that the transformation architect at any time can switch between the two depending on the working mode.

The evaluation in this report could be improved by using the reference examples with alternative approaches published in the literature. An available QVT-Merge tool is necessary in order to provide evaluations of all the suggested criteria. In order to further investigate the usability of the graphical notation, we need to define more of the transformation examples graphically. Only one of the examples has been specified graphically in this version. The current evaluation has been done by a single evaluator who has only reviewed the transformation code that was written by somebody else. The evaluation will be further improved by incorporating input from other evaluators as well as evaluation from those who wrote the transformation code.

11. References

[1] QVT-Merge_Group, *Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10)*. 2004,www.omg.org.

[2] Compuware_Corporation_and_SUN_Microsystems, *EXMOF Queries, Views and Transformations on Models using MOF, OCL and EXMOF, Proposal to document: MOF 2.0 Query / Views / Transformations RFP ad/2002-04-10*. 2004,www.omg.org.

[3] Gardner, T. and C. Griffin. *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*. in *MetaModelling for MDA Workshop*. 2003. York, England, UK.

[4] Langlois, B. and N. Farcet. *THALES recommendations for the final OMG standard on Query / Views / Transformations*. in *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. 2003. Anaheim, California, USA.

[5] OMG, *Object Management Group MOF 2.0 Query / Views / Transformations RFP*. 2002,www.omg.org.

[6] Bézivin, J., et al. *First experiments with the ATL model transformation language: Transforming XSLT into XQuery*. in *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*. 2003. Anaheim, California, USA.

[7] OMG, *Software Process Engineering Metamodel (SPEM), version 1.0*. 2002,http://www.omg.org/technology/documents/formal/spem.htm.

[8] OMG, *UML Profile for enterprise distributed Object Computing (EDOC) version 1.0; OMG Adopted Specification ptc/02-02-05*. 2002,http://www.omg.org/technology/documents/formal/edoc.htm.

[9] Sendall, S. and W. Kozaczynski, *Model Transformation – the Heart and Soul of Model-Driven Software Development.* IEEE Software, Special Issue on Model Driven Software Development, 2003.

12. Appendix

12.1. EDOC to J2EE using the Frauhofer formalism

This section will illustrate the transformation specification used in the Fraunhofer FOKUS tool chain. The transformation rules of this tool chain are not explicitly expressed but are implemented as C++ code. For illustrational purposes the rules are also described in terms of transformation diagrams and explaining text.

In the following figures the EDOC source modelling elements are shown as green boxes and the target J2EE modelling elements are shown as blue boxes.

12.1.1. Transformation of Package Structure

Package structures in EDOC are transformed to *JavaPackage* structures. Nested packages become nested packages and names are preserved.

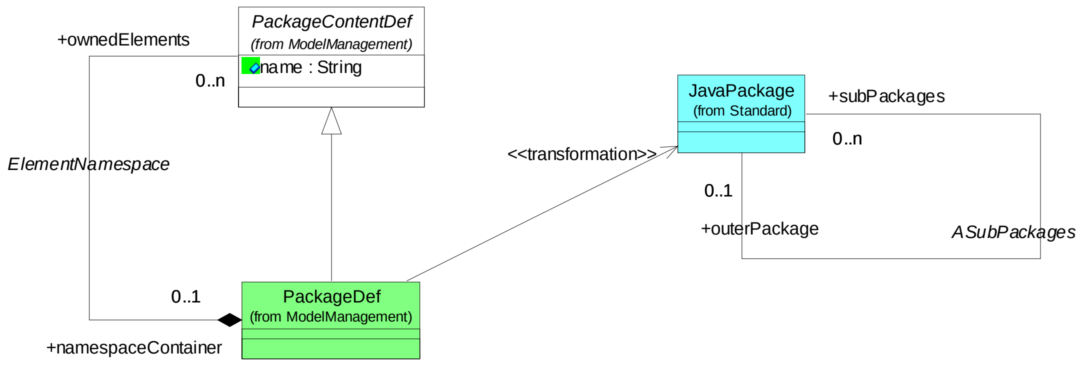


Figure 20 Transformation to Java packages

12.1.2. Transformation of primitive and composite data

EDOC *CompositeDataDefs* with attributes are transformed to *JavaClass* with *Fields.* The containment, name, and inheritance relations remain unchanged.

EDOC *DataTypeDefs* are transformed to Java *PrimitiveTypes.*

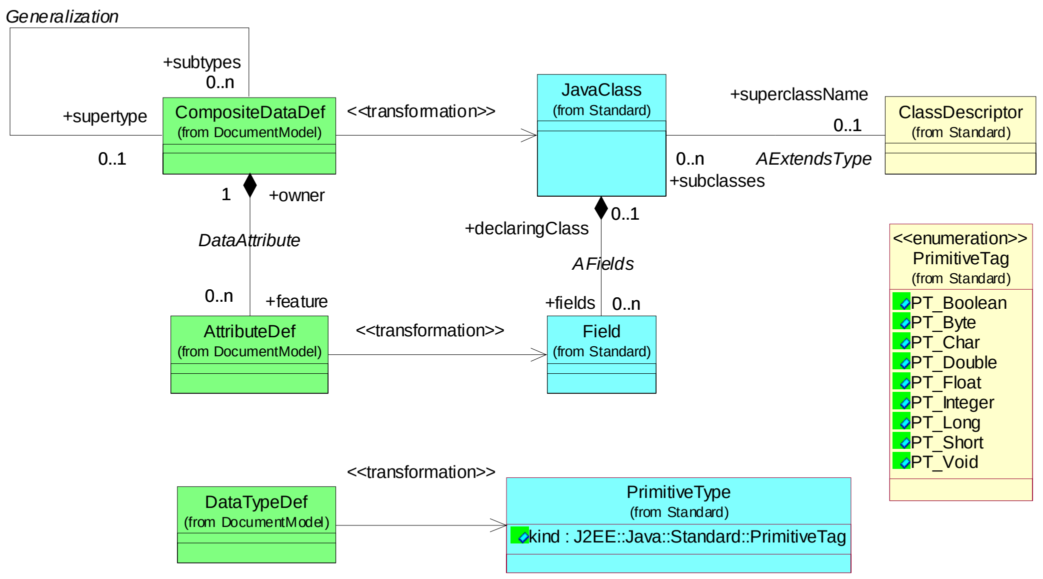


Figure 21 Transformation to Java classes and primitive types

12.1.3. Transformation of Data Managers

An EDOC *DataManager* is a functional component that provides access to and may perform operations on its associated *Composite Data,* i.e. its state. The Data Manager defines ports for access to operations on the state data. A DataManager inherits from *Process Component* and adds the quality of having an associated state. A Data Manager has two attributes:

- *Network Access:* A Boolean value which indicates if the *DataManager* is intended to be accessible over the network.
- *Sharable:* A Boolean value which indicates if the *DataManager* can be shared by multiple transactions/sessions.

DataManagers are mapped to Java classes. The fields of the Java class correspond to the attributes of the associated *CompositeData*.

If (NetworkAccess == true | Sharable == true), the *JavaClass* is implemented as a *JavaRemote* Object. The containment and name relations remain unchanged.

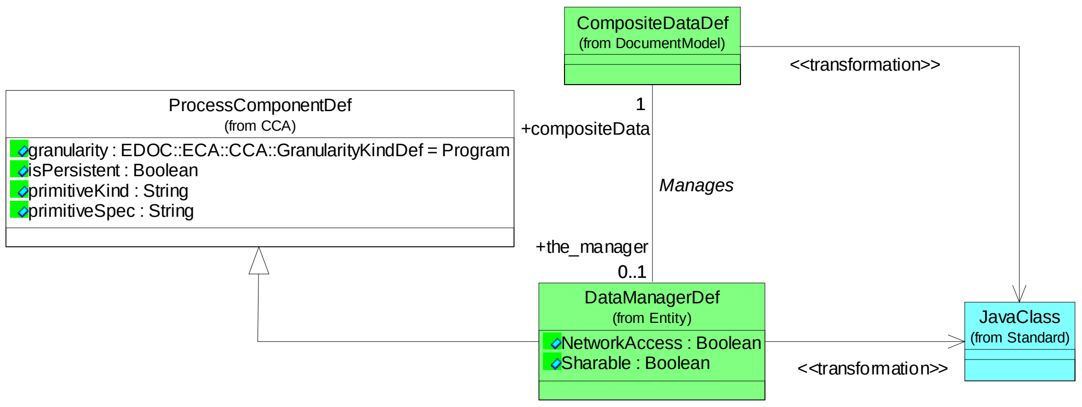


Figure 22 Transformation to Java class

12.1.4. Transformation of Entities

Entity specialises *DataManager* for the representation of identifiable business domain artefacts.

EntityData is the data structure that represents a concept in the business domain. It is equivalent to an entity in data modelling or a relation in a relational database. In a *DataManager* or its specialisation, such as *Entity*, it represents the state of an object. *EntityData* must have a prime *Key* that is unique within the extent of the *EntityData* type.

EDOC *Entities* are transformed to *EntityBeans.* Attributes of managed EntityData are mapped to *Fields* of the entity bean. Corresponding set/get methods are generated as part of the bean's remote or/and local interface.

A *JavaClass* is generated for the EDOC *KeyDef* . It becomes the *primary_key* of the entity bean.

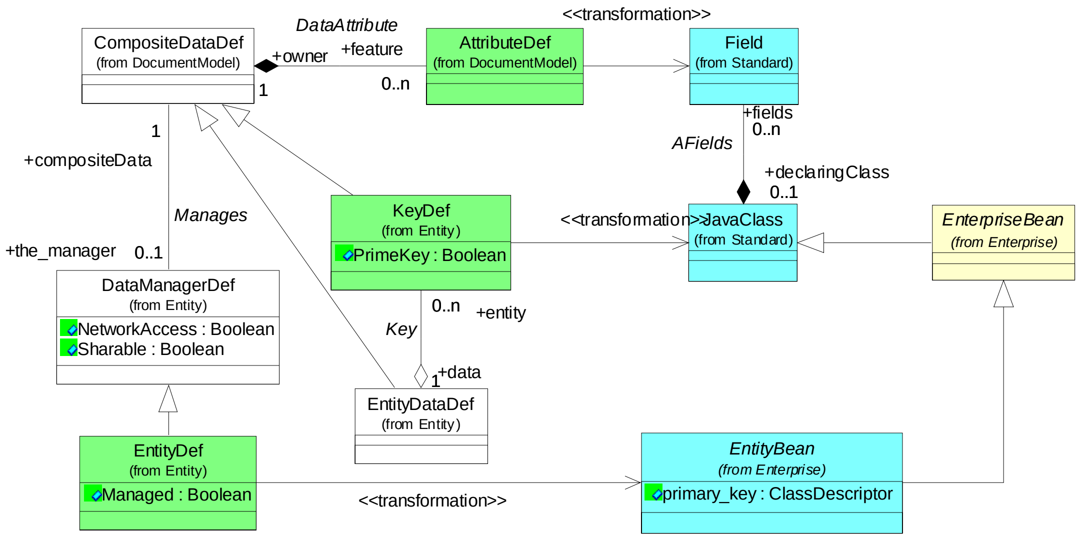


Figure 23 Transformation to entity beans

12.1.5. Transformation of Process Components

The mapping from EDOC process components to enterprise beans depends on the value of the attribute *GranularityKind,* which defines the scope in which the EDOC component operates. The values may be:

- **Program** – the component is local to a program instance (default).
- **Owned** – the component is visible outside of the scope of a particular program but dedicated to a particular task or session which controls its life cycle.
- **Shared** – the component is generally visible to external entities via some kind of distributed infrastructure.

The type of the target enterprise bean depends on the usage context of the source EDOC component.

If the granularity kind is *shared* and the component is used at the outermost level of a community process it is mapped to a *remote accessible stateless session bean* as shown in Figure 24. The containment, name, and inheritance relations remain unchanged.

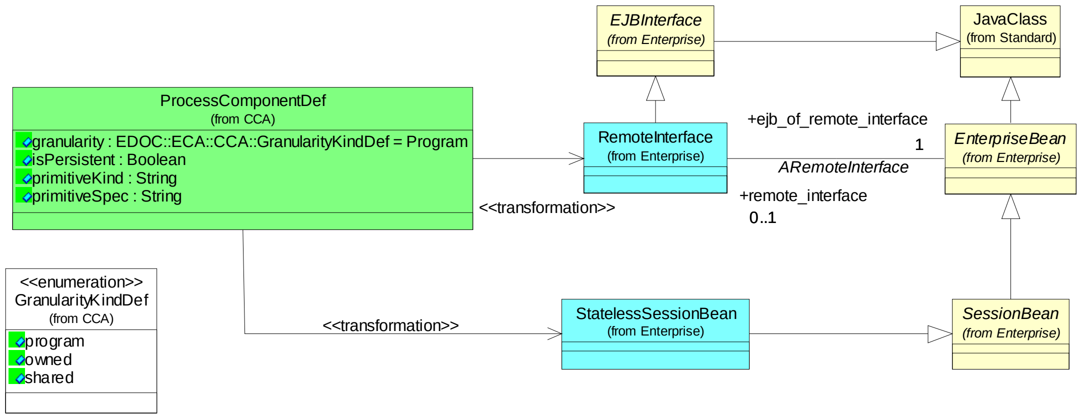


Figure 24 Transformation to a remote accessible stateless session bean

If the granularity kind is *shared* and the component is used within compositions it is mapped to a *local accessible stateless session bean* as shown in Figure 25.

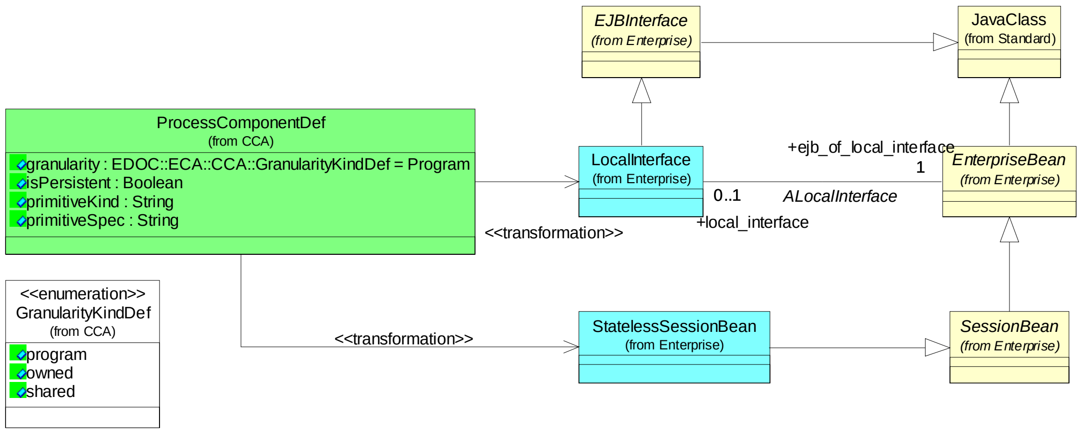


Figure 25 Transformation to a local accessible stateless session bean

12.1.6. Transformation of EDOC Ports

In general EDOC Ports are transformed to Java interfaces. The following rules depicted in Figure 26 hold for the transformation process:

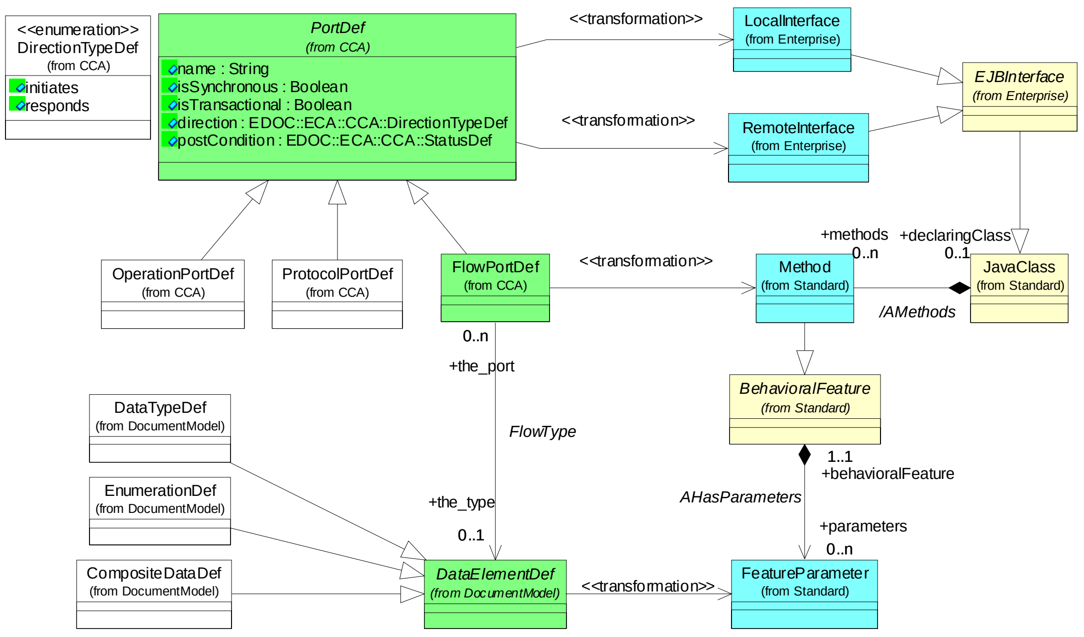


Figure 26 Transformation to interfaces, methods, and parameters

- *FlowPorts* with the direction *response* are transformed to methods of a Java interface. Interface is either:
- a bean's remote interface,
- a bean's local interface or
- a Java class with isInterface == true,
- depending on the transformation of the owning *ProcessComponent* of the flow port.
- *OperationPorts* with the direction *response* are transformed to *Methods.* Contained flow ports become parameters or return types of methods, depending of the direction (*initiates* or *response*).
- Other Ports as *ProtocolPorts* and *MultiPorts* are recursively decomposed according to their structure. Contained flow ports and operational ports with direction *response* become methods as described above. Other contained ports are further decomposed.
- Outermost flow ports and operational ports with direction *initiates* are ignored.
- A *DataElement* of a flow port becomes a *FeatureParameter*, a parameter of the method.