EMODE

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

Deliverable D2.3 – Modelltransformation

Telecooperation Report No. 5,
The Technical Reports Series of the Telecooperation Research Division,
TU Darmstadt
ISSN 1864-0516

We thank all partners who contributed to this document, namely IKV and TU-Dresden. We thank all other partners for their support. The document was edited by TU-Darmstadt.

Contributors, in alphabetical order:
Alexander Behring, TU-Darmstadt
Matthias Erche, IKV
Rene Neumerkel, TU-Dresden
Andreas Petter, TU-Darmstadt
Anja Strunk, TU-Dresden
Michael Wagner, IKV

DOCUMENT INFORMATION	
TYPE	Deliverable
ID	D2.3 – Modelltransformationen
DUE DATE	July 31st 2006

WORK PACKAGE	WP 2 – Metamodell/Modelltransformationen
PROJECT	**01ISE02 EMODE** **Enabling Model Transformation-Based Cost Efficient Adaptive Multi-modal User Interfaces**

DOCUMENT STATUS		
ACTION	BY	DATE (dd.mm.yyyy)
SUBMITTED	Andrease Petter, TU Darmstadt	31.07.2006
WP LEADER	SAP	
APPROVED		
APPROVED		
APPROVED		

REVISION HISTORY			
DATE (dd.mm.yyyy)	VERSION	AUTHOR	COMMENT
31.07.2006	1.0	Andreas Petter	Finalized Version

Table of Contents

1 Transformations in EMODE 5
1.1 Design Time Transformation 5
1.1.1 Model-To-Model Transformations 6
1.1.1.1 Model-To-Model Transformations according to the MDA 6
1.1.1.2 Multidirectionality of Transformations 7
1.1.1.3 Identified Model-To-Model Transformations 7
1.1.1.4 Functional Goal ⬄ Executable Task Node 8
1.1.1.5 Executable Task Node ⬄ AUI Interactor 10
1.1.1.6 System Task ⇨ FCA Call or Task Definition 11
1.1.1.7 Patterns 12
1.1.2 Model-To-Code Transformations 12
1.1.2.1 JET (Java Emitter Template) 14
1.1.2.2 Velocity 14
1.1.2.3 XSLT (eXtensible StyleSheet Language Transformation 15
1.1.2.4 Conclusion 16
1.2 Runtime Transformations 17
1.2.1 Definitions 17
1.2.2 Runtime Transformation Purpose 17
1.2.3 Implementing a Mixture of Different Types of Transformations to Implement Adaptation 18
1.2.3.1 Modeling at Design time 19
1.2.3.2 Transformations at Design time 20
1.2.3.3 Transformations at Runtime 22
1.2.4 EMODE Runtime Transformations 22
2 Transformation Language 25
3 Exemplary Transformations 28
3.1 Transformation examples from EMODE 28
3.1.1 Model-To-Model example 28
3.1.1.1 Transformations 29
3.1.1.2 Relations 29
3.1.1.3 Variables 30
3.1.1.4 Domains 30
3.1.1.5 Object templates 31
3.1.1.6 When 31
3.1.1.7 Where 31
3.1.2 Model-To-Code example (FCA Diagram) 32
3.1.2.1 Services 33
3.1.2.2 FCAs 34
3.2 A Small Guide for Writing QVT Transformations 38
4 Appendix 41
4.1 Bibliography 41

List of Figures

Fig 1 : Goals to Tasks 9
Fig 2 : Tasks to DialogueSpace 10
Fig 3 : System Task to FCACall 11
Fig 4 : System Task to Task Definition 12
Fig 5 : visitor based transformation 13
Fig 6 : template based transformation 13
Fig 7 : JET transformation process 14
Fig 8 : Velocity transformation process 15
Fig 9 : XSLT transformation process 15
Fig 10 : Modelling Times 18
Fig 11 : Advantages and Disadvantages of Implementation Strategies 19
Fig 12 : Modalityrestrictions in EMODE Metamodel 20
Fig 13 : UI refinement 23
Fig 14 : AUIInteractor 24
Fig 15 : QVT Languages and Their Relation [12] 25
Fig 16 : UML Class to Relational Table Relation - Graphical Notation [12] 26
Fig 17 : UML Class to Relational Table Relation - Textual Notation [12] 26
Fig 18 : QVT Metamodel 27
Fig 19 : GenerateTaskModel 28
Fig 20 : QVT graphical notation of GenerateDialogueSpace 29
Fig 21 : the FCA Model 32
Fig 22 : example FCA model 33
Fig 23 : Java class of the service „IsEmployee_S1" 34
Fig 24 : class of the service „IsEmployee_S2" 34
Fig 25 : the FCA's Java class – part 1 35
Fig 26 : the FCA's Java class – part 2 36
Fig 27 : The Case Statements Implementation 37
Fig 28 : QVT Relations example in QVT final adoption 38

1 Transformations in EMODE

Note: due to the short research times, *different parts of the document have been developed at different stages of the EMODE project and therefore are based on different versions of the metamodel. A final version of this document will be produced at the end of this project in deliverable D2.4.*

According to the MDA Guide [8], a *Model Transformation* is "`the process of converting one model to another model of the same system`". They have a wide range of applications within the EMODE project. The two main purposes are:

- to be executed in order to transform one model into another one, which the development or runtime process continues with (generative/unidirectional), and
- to ensure consistency between two models that have mappings between them (relational/multidirectional).

As hinted in the first bullet point, transformations are not limited to be used at design time, as it is intended by the Model Driven Architecture (MDA). They can as well be used at runtime, e.g., to adapt to the current context the application is used in.

For transformations at design time, EMODE focuses only on a specific set of development phases, as introduced in D2.1 [10]. This enables us to perform more in-depth research, as the full development process is too broad to be covered in the frame of this project. Hence, this document only covers transformations for the main development phases (as described in D2.1):

- High-level Design,
- Detailed Design, and
- Implementation.

Following these introductory notes, this document is divided into three main sections. This section (1) introduces transformations in EMODE. The second section (2) elaborates on the transformation language for model-to-model transformations. The document closes by giving concrete examples in the last section (3).

This first section starts by introducing Design Time Transformations. It elaborates on how Design Time Transformations are embedded into EMODE, what metamodel packages could be used for transformation definition and how models could be transformed into code,

1.1 Design Time Transformation

With *Design Time Transformations*, we denote all transformations that occur during the design process of the application (see e.g. [2] for a definition of terms). This may range from early project phases, like a high-level semi formal project description to very late phases, like building deployment packages. Hence, they are used to support the development process, which is the focus of the transformations introduced in this section.

Design Time Transformations can occur as model-to-model or model-to-code. Theses two types are introduced in the following subsections.

1.1.1 Model-To-Model Transformations

As introduced in the beginning of this document, Model-To-Model Transformations produces models out of other models. A model is defined in the MDA Guide [8] as: "`A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language.`"

Hence, a model is a description of the system, often capturing information about a specific aspect of the system (e.g., a task model which describes tasks of the system). Consequently, models can overlap in their contained information, or one model can capture the same information as another one, only with more implementation relevant details. To synchronize the overlapping aspects and produce the already existing facts for a new model from a preexisting one, the need for transformations arises.

The first subsection introduces some of the concepts defined in the MDA Guide [8] for transformations at design time. Following this, the subsequent subsection goes into Model-To-Model Transformations that can be identified in EMODE. The last subsections introduce various examples, where Design Time Model-To-Model Transformations can be applied in EMODE.

1.1.1.1 Model-To-Model Transformations according to the MDA

This subsection introduces some of the concepts advocated in the MDA Guide [8]. It briefly describes what types of Design Time Transformations exist. Hereby it must be noted that all transformations introduced in the MDA Guide have generative character and require a Platform Independent Model (PIM), which is transformed into a Platform Specific Model (PSM) utilizing a Platform Model (PM).

$$PIM \xrightarrow{PM} PSM$$

The PM is left out in the following, since it in all cases, it is either implicitly included within the source model, has no major influence on the transformations definitions within EMODE, or is to be defined, yet.

Model Type Mappings
Map elements of the metamodel of the PIM to elements of the metamodel of the PSM. With this abstract mapping of elements, very powerful transformations with a high degree of reuse can be generated that operate on all models that are compliant to the PIM-metamodel, the transformation is defined for. The downside of model type mappings is that their identification is very difficult and might be seen to be on a very abstract level by developers, who have never seen the metamodel (e.g. developers who only use EMODE editors for development).

Model Instance Mappings
Transformations that map model elements of the PIM (i.e. not by selecting them via the metamodel) to PSM model elements are called Model Instance Mappings. These transformations have less impact per selected element, because they transform only one element per selection. Involved model elements can be identified fairly easily,

because they are directly available. Usually, instances need to be selected by markings, which have to be set by hand.

Combined Type and Instance Mappings
To leverage the trade-off between powerful Model Type Mappings and specific Model instance Mappings, the two techniques could also be combined. This actually is the most feasible approach to be used in practice.

1.1.1.2 Multidirectionality of Transformations

Another important feature of a transformation is in which direction the transformation alters and creates elements. The two most common cases would be generative and relational.

Generative Transformations
Generative Transformations have one or more target models that they operate on. The source models deliver input information which (by the transformation rules) specifies how the target model elements should look like. Hence, the transformation generates the elements in the target models.

Relational Transformation
If not only the source models specify how the target models should look like, but as well from the configuration of the target model, the elements in the source model can be derived, the transformation defines a relation between source and target model. Accordingly, it is called a Relational Transformation. Consequently, it can be used to ensure consistency between the models, that is, if the configuration in question will allow for that, the transformation can be applied in both direction, switching source and target models.

The Relational Transformations can be seen as a generalization of Generative Transformations. If a Relational Transformation is defined and its source models are protected from modification, it is restricted to a Generative Transformation.

It is also noteworthy that Relational Transformations can very conveniently be expressed in a declarative language. The transformation engine then has to take the appropriate steps to ensure the consistency (constraint) between the source and target models, as defined in the transformation.

1.1.1.3 Identified Model-To-Model Transformations

Through Model-To-Model Transformations in EMODE, specific metamodel packages[1] are connected. By transformation, e.g., a goal model into a task model, the goal and task metamodel packages have an implicit connection through the defined transformation.

All Design Time Model-To-Model Transformations in EMODE map from and to the EMODE metamodel, they are called “inplace” transformations. This equivalence of PIM (source) and PSM (target) metamodel is not the case normally (it is a special case), but here it is a reasonable assumption, since the whole design process for EMODE is based on the EMODE metamodel[2].

[1] Please refer to the deliverable D2.2 for a more detailed description of the EMODE metamodel.
[2] Of course, one could imagine transformations in and out of the EMODE metamodel [9], but the current focus of this work is not on this direction.

The transformations used in EMODE at design time further comply to the combined type and instance mappings class, described in the previous subsection. Specifically, in the more abstract design levels (e.g., goal and task model), mostly model type mappings are used, whereas in more concrete levels (e.g., AUI), model instance mappings come into play.

Model type mappings are applied during development phases, since the mapping between metamodel elements can be identified clearly. For example, in section 1.1.1.4 a mapping between a goal and a task node is defined. This mapping is purely based on the identification of metamodel properties and not concrete element values.

When going into the AUI-Refinement process, as described in D2.2 [9], the variety of element features imposes the need to detail the transformations on how properties of the metamodel elements are set by the developer. For example the transformation of a certain AUI library element into a more concrete version should be defined. Hereby, the transformation must be based on the values of the AUI elements properties. It is important to note in this context that library elements are not metamodel elements, but rather instances of them. The actual “library information” lies in the properties of the metamodel element and not in what metamodel element is used.

The following subsection introduces metamodel mappings between different packages. These mappings are exploited in section 3 to give some exemplary transformations. As noted in the introduction, these mappings and transformations may be based on different versions of the metamodel. They are given here anyway, because their basic idea would remain the same in most metamodel versions.

In the following subsections, several transformations are introduced that can be applied during design time through the development environment.

1.1.1.4 Functional Goal ⬄ Executable Task Node

Starting with the definition of Goals a model of an EMODE based application is created. To help the developer with the creation of further model elements a transformation from FunctionalGoals to ExecutableTaskNodes has been defined. Depending on the transformation direction chosen by the developer, the transformation creates model elements normally creates model elements in the task diagram from model elements out of the goal diagram and vice versa.
The transformation process may be invoked at different stages of the development process. Therefore, the transformation has been split up into two different transformations.
If the modeling phase will be at a very early stage, the developer may want to create an initial task diagram from the goal diagram, she already created (all lines in Fig 1).
At a later stage, the developer might want to create model elements using the names from the corresponding model elements of the other diagram, if they won’t exist (solid lines of Fig 1, only). This might be done iteratively in both directions, if new goals cross the mind of the developer while refining the tasks, leading to “round-trip engineering”.

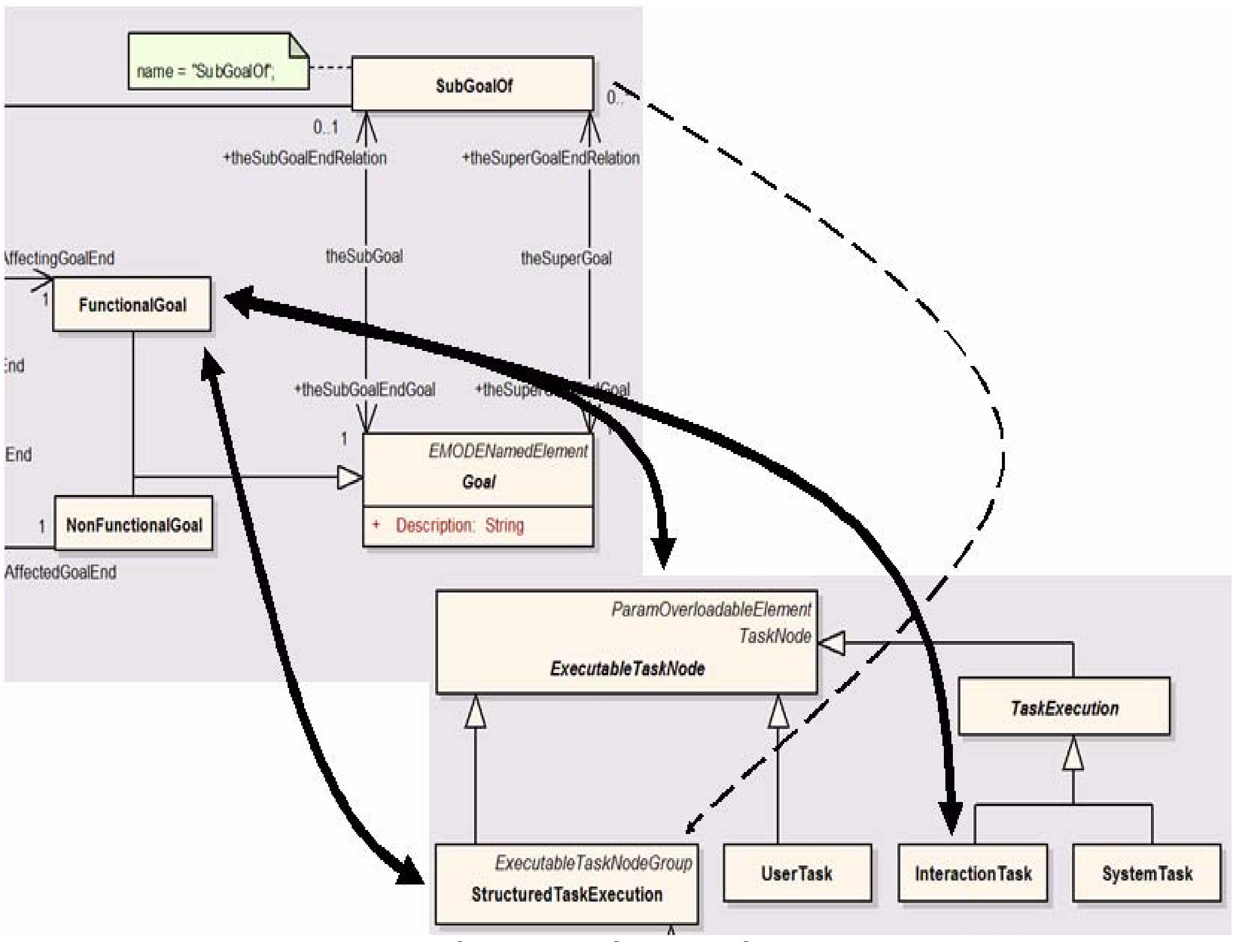


Fig 1 : Goals to Tasks

Fig 1 illustrates the transformations by arrows between metamodel elements involved in the transformation processes. While the arrows which are not dashed show the transformation of model elements the dashed line illustrates the creation of hierarchies.
FunctionalGoals are transformed into InteractionTasks if they are leafs in the tree spawned by the SubGoalOf relation, and a ExecutableTaskNode with the same name doesn't exist, yet (InteractionTasks are most probable to occur in a multimodal application and therefore have been chosen to be created). Otherwise, it is either no leaf or has been switched to another model element conforming to ExecutableTaskNode with the same name (e.g. SystemTask). If it won't be a leaf a StructuredTaskNode with the same name must either exist or is going to be created in the transformation process. As can be seen here the transformation does not take into account that several tasks with the same name do exist as no other key is available to the transformation. This transformation is meant to be used at any time in the modelling phase to help the developer with synchronizing tasks with the goals.
The dashed line implicates the use of hierarchy information, only when the developer wants to create an initial task diagram which conforms to the hierarchy of Goals implied by the SubGoalOf relation. This relation is transformed by using the structuring attribute available in StructuredTaskExecution ("containedNode").

1.1.1.5 Executable Task Node ⇔ AUI Interactor

When the developer has created a task diagram, she might want to create initial user interface elements from the task diagram elements. The transformation presented is divided into 2 smaller transformations.

The first one creates an initial set of AUIInteractors which are aligned at the tree structure of the tasks. Therefore it is the first step a developer takes to create model elements of the dialoguespace package. The names of the newly created model elements are derived from the name of the tasks, which are transformed. The hierarchy of the tree structure of tasks is preserved upon the transformation process. This may seem strange since it is probable that the tree structure of the AUIInteractors will be different after the developer finished editing the dialogue space diagrams. However, the developer will probably be guided by the structure of the AUIInteractors such that he has a better understanding of the domain, which will ease her user interface development, or might even stick with the structure for a first prototype.

The second transforms ExecutableTaskNodes and AUIInteractors (and vice versa) to help the developer during an iterative development process, by providing basic means of transformation.

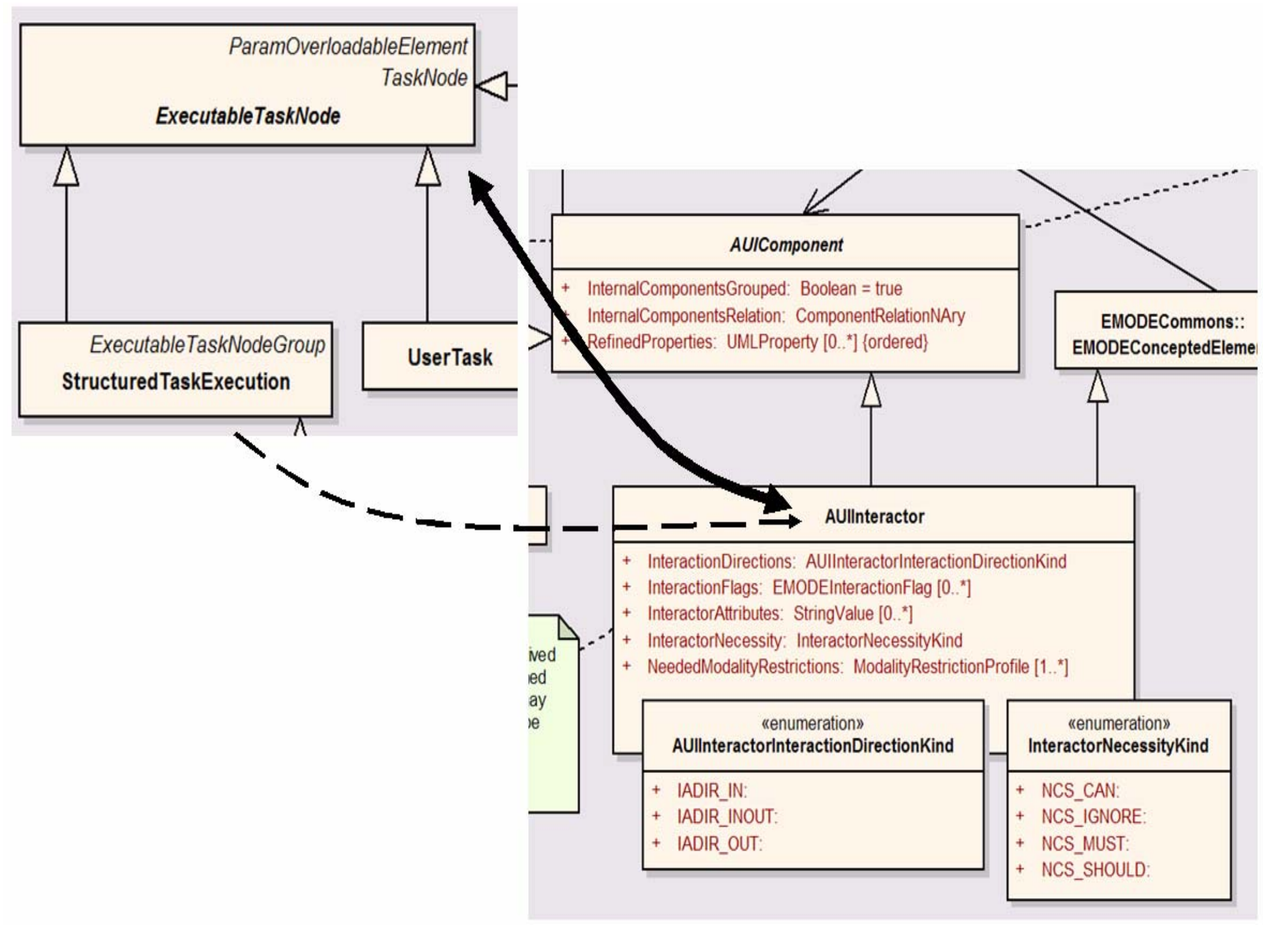

ParamOverloadableElement
TaskNode
ExecutableTaskNode
ExecutableTaskNodeGroup
StructuredTaskExecution
UserTask
AUIComponent
+ InternalComponentsGrouped: Boolean = true
+ InternalComponentsRelation: ComponentRelationNAry
+ RefinedProperties: UMLProperty [0..*] {ordered}
EMODECommons::
EMODEConceptedEleme
AUIInteractor
+ InteractionDirections: AUIInteractorInteractionDirectionKind
+ InteractionFlags: EMODEInteractionFlag [0..*]
+ InteractorAttributes: StringValue [0..*]
+ InteractorNecessity: InteractorNecessityKind
+ NeededModalityRestrictions: ModalityRestrictionProfile [1..*]
«enumeration»
AUIInteractorInteractionDirectionKind
+ IADIR_IN:
+ IADIR_INOUT:
+ IADIR_OUT:
«enumeration»
InteractorNecessityKind
+ NCS_CAN:
+ NCS_IGNORE:
+ NCS_MUST:
+ NCS_SHOULD:

Fig 2 : Tasks to DialogueSpace

As seen from the metamodel level, AUIInteractors may be created from ExecutableTaskNodes, as is depicted in Fig 2 by the solid line. This transformation will work both directions (source and target may be exchanged by the developer using the transformation frontend). The dashed line shows the transformation that is done additionally, when the model elements of the dialoguespace are created the first time. The structuring attribute of the StructuredTaskExecution (the attribute is called "containedNodes", which is inherited from ExecutableTaskNodeGroup) is transformed into the structuring attribute of the AUIInteractor (the attribute is called "nestedClassifiers", which is inherited from UMLComponent) by adding those AUIInteractors to the structuring attribute, which have the same name as the children of the corresponding ExecutableTaskNode.

1.1.1.6 System Task ⇨ FCA Call or Task Definition

A System Task describes a task that the system has to accomplish. These tasks are more specifically described through FCACalls (Functional Core Adapter Calls) that route the system task to the object that executes it. Alternatively, if a System Task is not realized through a FCACall, it may be defined by a Task Definition node.

Hence, if at a stage of development, a system task is found that neither is connected to an FCACall nor defined by a Task Definition, the corresponding FCACall or Task Definition should be created. Of course, the decision whether an FCACall or a Task Definition should be created is made by the developer. In a development environment only supporting one or the other transformation at a time is feasible.

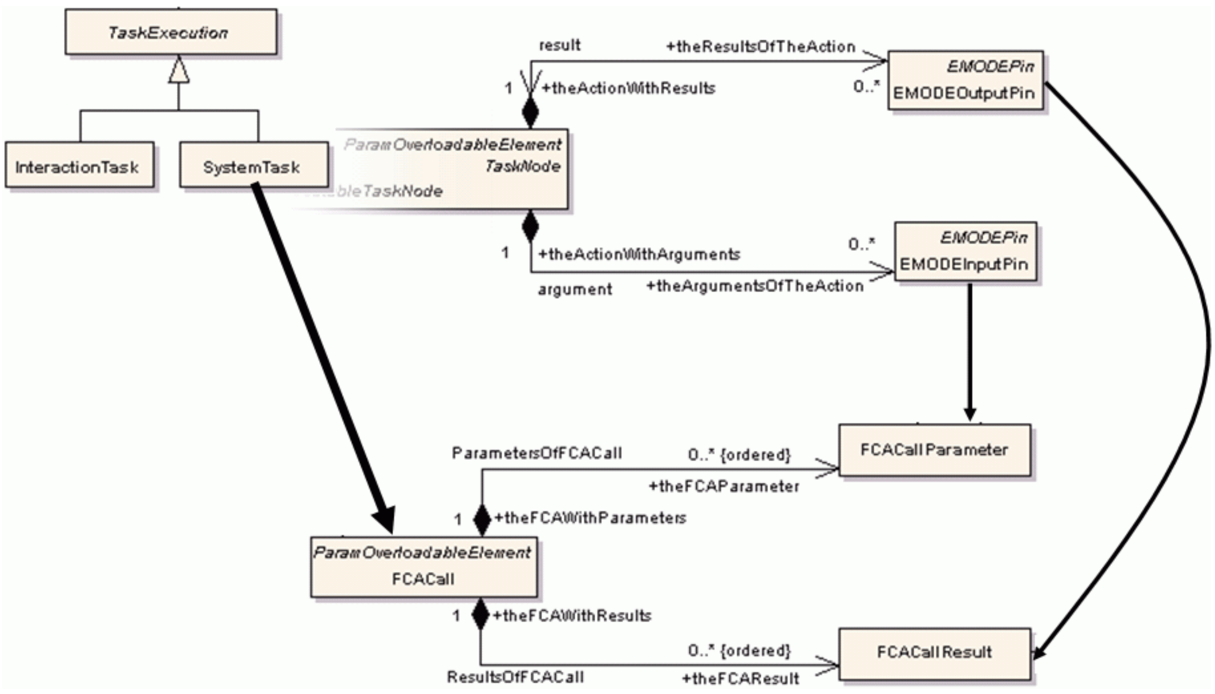


Fig 3 : System Task to FCACall

A System Task is used for generation of an FCACall. The transformation has to make sure that for every connected input pin, there exists an FCA call parameter. Also, for every output pin of the System Task, a FCA call result must be created.

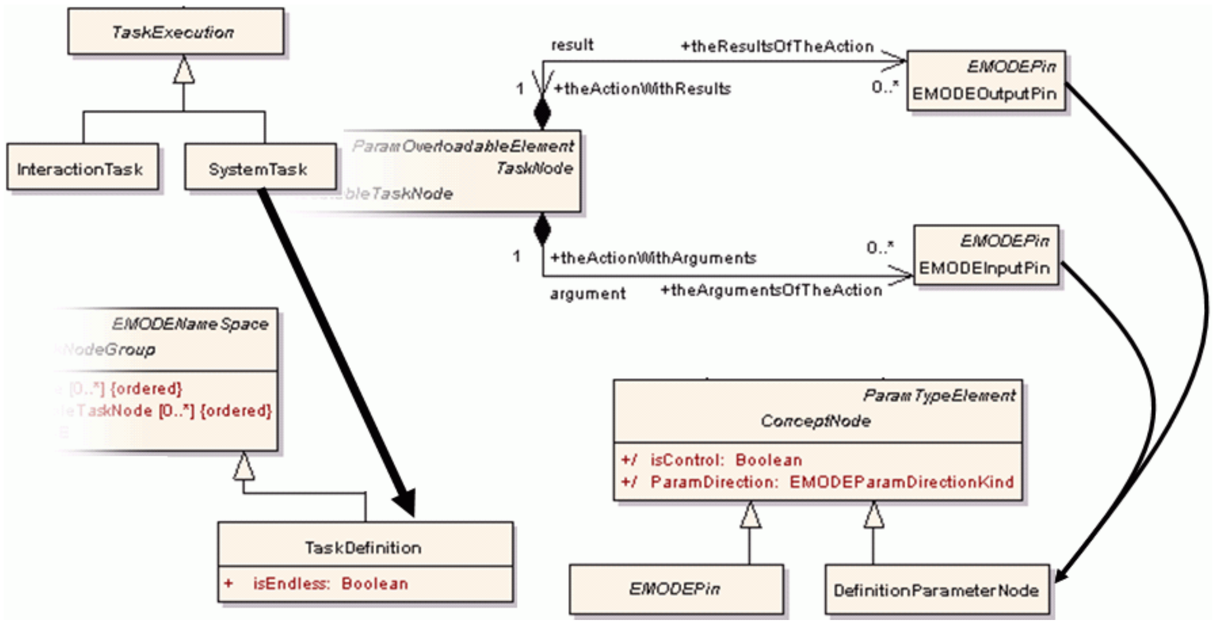


Fig 4 : System Task to Task Definition

A System Task is used for generation of a Task Definition. The transformation has to make sure that for every connected input pin, there exists a Definition Parameter Node. Also, for every output pin of the System Task, a Definition Parameter Node must be created.

1.1.1.7 Patterns

Using EMODE model elements while developing multimodal applications will probably lead to patterns which developers might want to use while developing new applications. Patterns of model elements might be generated using transformations. Therefore we imagine a set of transformations which will generate patterns of model elements in EMODE. However, since EMODE hasn't identified any patterns, yet, the transformations will be defined as soon as the patterns, and the use of transformations makes sense for those patterns identified.

The transformations introduced in the last sections support the development process for EMODE applications. They are based on the EMODE metamodel and will be formulated in QVT to be tested in the development environment. Through developer feedback, more transformations can be identified and formulated.

After introducing more abstract Model-To-Model Transformations, the next section deals with the concretization of models to code.

1.1.2 Model-To-Code Transformations

A special case of model-to-model transformation is the model-to-code transformation. Today, one distinguishes two different implementation approaches: the visitor based approach and the template based approach.

Within a visitor based model transformation some visitor mechanism exists to traverse the internal model structure and produce the whole code as text stream or a number of text files. That means the target model is created completely during the transformation (See Fig 5).

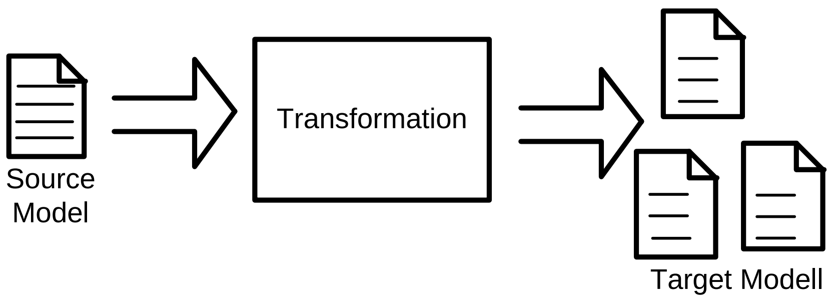


Fig 5 : visitor based transformation

A template based transformation uses templates (text files) in order to specify most of the target model before transformation. These text files are not complete, but have some gaps filled with statements of a special template language. During the transformation process the statements are executed and the missing values depending on the source model are inserted. (See Fig 6)

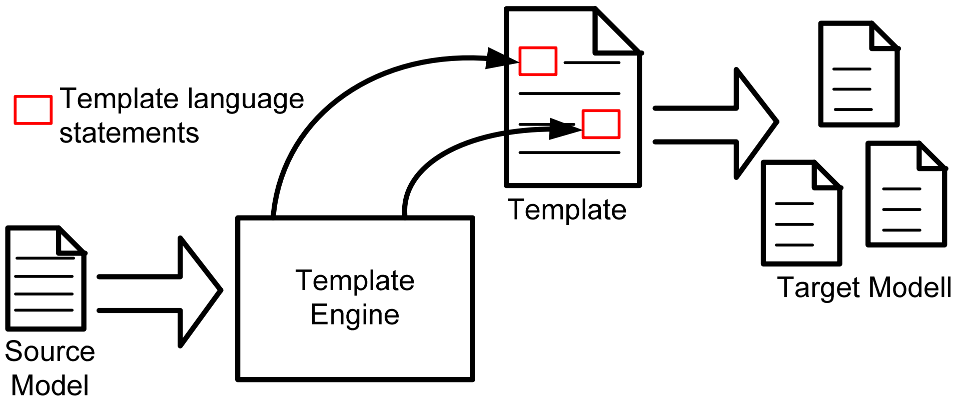


Fig 6 : template based transformation

Although visitor based transformation is very simple and for smaller models will be written quickly, template based transformations are the better approach. Limiting the transformation to the source dependent parts and separating them from the independent ones makes the transformation more effective and better to maintain than a large program as it is used in visitor based transformations.

Even though there are visitor based transformation frameworks (e.g. Jamda[3], an object oriented Java Framework, which supports creation of transformations on UML source models) only template based frameworks will be taken into account when finding the best model-to-code language for EMODE.
The best known template based transformation frameworks are compared in table1. Nearly all MDA tools which use template based transformations and don't provide their own transformation language (like OptimalJ[4] or b+m Generator[5]) make use of one of the frameworks shown in table 1 (e.g. AndroUML[6] or UMT[7]).

[3] http://sourceforge.net/projects/jamda
[4] http://www.compuware.com/
[5] http://www.architectureware.de/produkte/generator_framework.htm
[6] http://www.andromda.org/
[7] UML Modeling Tool; http://umt-qvt.sourceforge.net/

Framework	Source Model	Target Model
JET[8]	any kind	any text based kind
Velocity[9]	any kind	any text based kind
XSLT[10]	tree based (XMI, XML)	any kind

Table 1: most well-known template based transformation frameworks

1.1.2.1 JET (Java Emitter Template)

JET is a Java based model-to-code transformation framework, which translates every kind of source model in every text based kind of target model. Similar to Java Server Pages a JET template contains portions of executable Java code. The JET Engine translates these templates into Java classes implementing the template's filling process in the *generate()* method. In order to get the complete target model (as text stream) defined within the template, one only needs to call this method with the source model or a relevant part of it as parameter. (See the Fig 7).

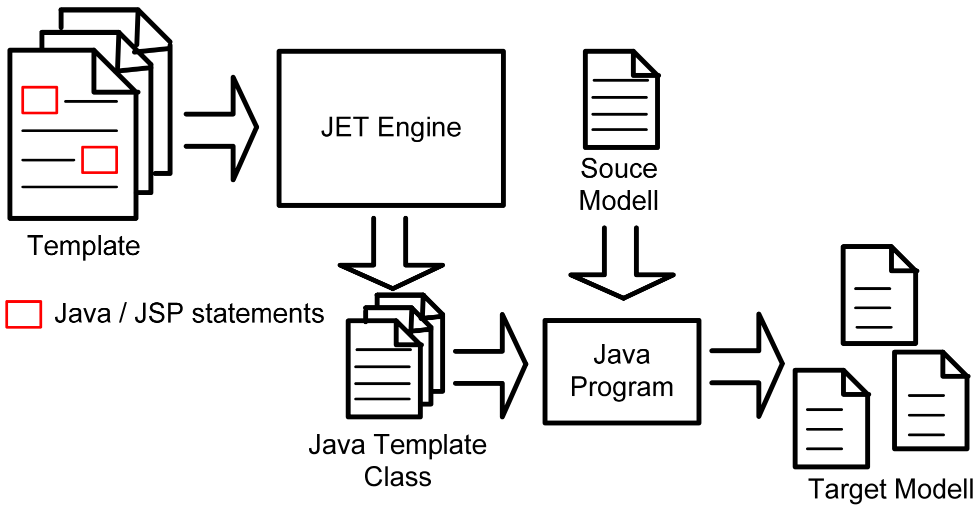


Fig 7 : JET transformation process

1.1.2.2 Velocity

Velocity (see Fig 8) can - like JET - transform any kind of source model into any kind of text based target model. But unlike JET, it uses its own template language called Velocity Template Language (VTL), in order to express source model dependent target model parts. The information needed from the VTL statements in order to fill the template's gaps, aren't read directly from the source model, but from a special Java object, called context object. The Java program, which starts the VTL Engine and processes the filled target model text stream, reads the source model and writes the required data into the context object in order to provide them for usage in the engine which will process the VTL statements.

[8] http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html
[9] http://jakarta.apache.org/velocity/
[10] http://de.wikipedia.org/wiki/XSLT

Thus, VTL is as powerful as JET, because VTL can access any Java object method or property (also user defined) if its result or the providing class is available inside the context object.

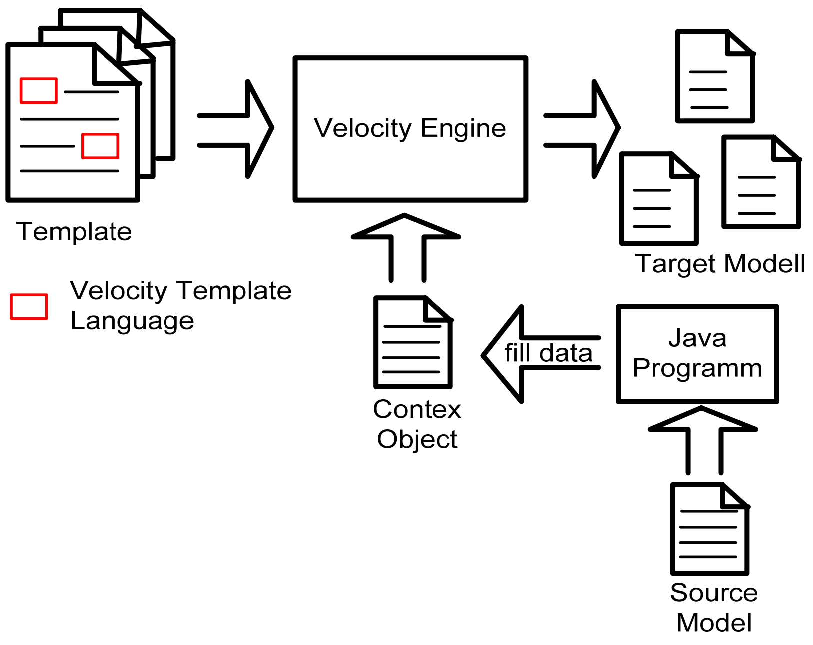


Fig 8 : Velocity transformation process

1.1.2.3 XSLT (eXtensible StyleSheet Language Transformation

XSLT is the W3C standard for transforming tree structured source models stored in XMI or XML. Unlike Velocity and JET, which can produce any kind of text based model, the XSLT templates, better known as XSLT StyleSheets, must be structured as tree. In this tree the gaps are expressed with XPath statements, which define a pattern. All elements which conform to this pattern are inserted in the gap in order to complete the target model. Fig 9 shows this transformation process.

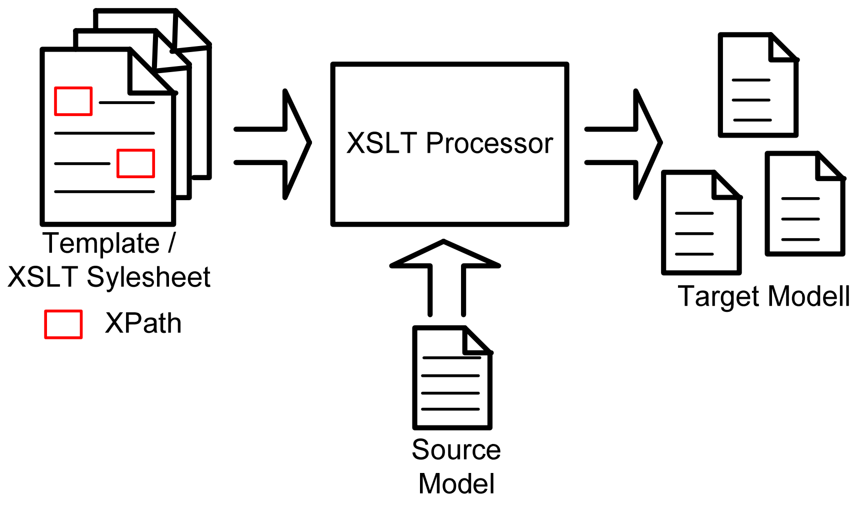


Fig 9 : XSLT transformation process

1.1.2.4 Conclusion

EMODE uses JET in order to define the model-to-code transformations. Unlike XSLT, JET can transform any kind of source models and nearly any kind of target models, thus with the use of JET the EMODE model format is not limited. Furthermore, the tree format is very difficult to read, especially if one wants to transform huge models.

JET's advantage over Velocity is its utilization of Java as template language. Java is known to most developers, whereas the use of Velocity would require them to learn a new language.

1.2 Runtime Transformations

A meta-model based project that claims to handle adaptation will eventually come to the point, where adaptation of models seems appropriate. This chapter gives some insight into a decision process of how to model adaptation and why adaptation based on models will be connected to the term “runtime transformations” (if open adaptivity is a requirement, see definition 1).
In this chapter we will dive into transforming user interfaces, since EMODE decided on using different types of modeling techniques for user interfaces (generating and modifying model elements using transformations may be seen as automatic modeling) and therefore a concept, which is mostly (this may be misleading) called runtime transformations, in the context of the AUI diagram.

1.2.1 Definitions

In EMODE everybody has a common understanding of the term “context”, therefore it is not defined here.

Definition 1 Reacting to context parameters, which are unknown at modeling time, or a set of all possible values of the context parameters (situations) which will be unknown, is called *open adaptivity* [15].

Definition 2 A *context parameter* is a variable or a vector of context parameters that models (represents) a carefully chosen part of context.

Definition 3 A vector of values being of a special type of vector of context parameters is called a *situation*.

1.2.2 Runtime Transformation Purpose

At a first glance, one might wonder, why one would ever need runtime model transformations as this is not a standard way to include model transformations into application development, e.g. MDA. Most of the applications that are developed in industry do not take into account the mechanism needed for adaptation.

Since we will support some kind of open adaptivity in EMODE, we need an implementation of this feature. In the past we saw that using model transformations at design time has increased efficiency of model based software development. By using the experience gained from that, we will use transformations at runtime to increase the efficiency to model adaptivity of applications in a way that supports open adaptivity (in a very restricted setting). This way to implement adaptivity will work for applications, which have been developed using model driven development, since the adaptation will take place within the models developed, which are available to tools, applications, or developers, both, at design time and at runtime.

A modeling process that supports open adaptivity within models may be implemented only using runtime transformations, since the developer may not know all parameters of entities involved or all possible situations and therefore may not know all model changes that may happen during runtime, which prevents her from using a standard modeling approach which is performed during design time only.

In the following sections a more detailed description of several possible ways to model adaptation is given and advantages and disadvantages are shown.

1.2.3 Implementing a Mixture of Different Types of Transformations to Implement Adaptation

Adaptation may be modelled at different stages of an application development phase or even at runtime using appropriate mechanisms. Modelling adaptation at runtime is excluded from this document.

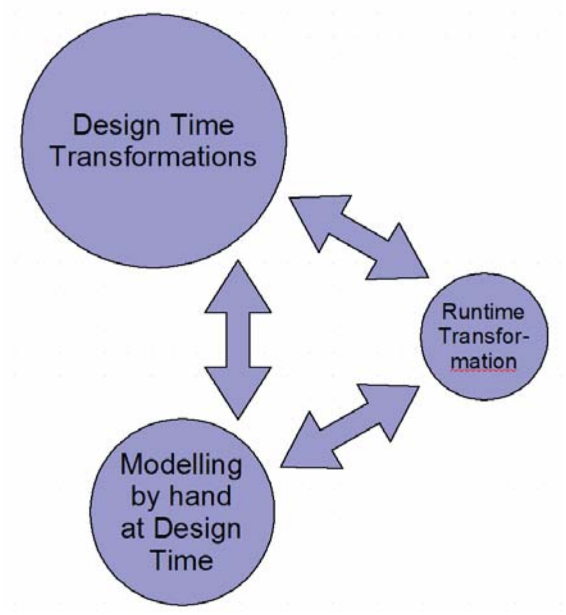
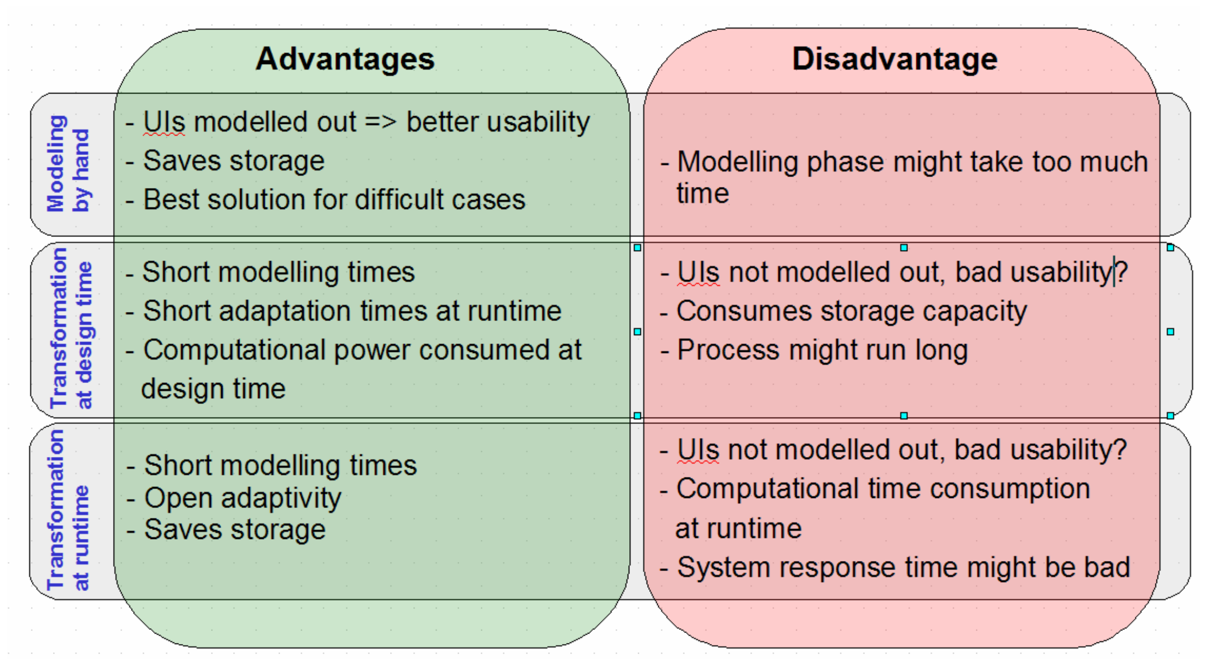


Fig 10: Modelling Times

There are 3 extreme positions that may be taken during the software design process, when modeling adaptation. The developer may either *model adaptation at design time* by modeling all models which correspond to all possible situations which may occur, may choose to *let this be done by design time transformations* or last but not least *let this be done at runtime*. Each of these 3 possible implementation strategies for adaptation has its individual advantages or disadvantages. It is left to the developers to choose a strategy between those extreme positions based on the given project needs.



	Advantages	Disadvantage
Modeling by hand	- UIs modelled out => better usability - Saves storage - Best solution for difficult cases	- Modelling phase might take too much time
Transformation at design time	- Short modelling times - Short adaptation times at runtime - Computational power consumed at design time	- UIs not modelled out, bad usability? - Consumes storage capacity - Process might run long
Transformation at runtime	- Short modelling times - Open adaptivity - Saves storage	- UIs not modelled out, bad usability? - Computational time consumption at runtime - System response time might be bad

Fig 11: Advantages and Disadvantages of Implementation Strategies

1.2.3.1 Modeling at Design time

Currently, the standard way to achieve adaptation through modeling is to model out the adaptation. This is done by providing special nodes or attributes that give hints to the transformation process or the runtime that this model is to be used under a given situation or that parts of the model are to be used. For example in [16] the CTT Metamodel is enhanced by a decision node that give hints to a transformation process what subtree is to be used under a certain situation.
Fig 11 depicts several different Advantages and Disadvantages of different adaptation strategies which incorporate modeling or model transformations. The current paragraph reflects the first bar titled with “Modeling”.

Advantages

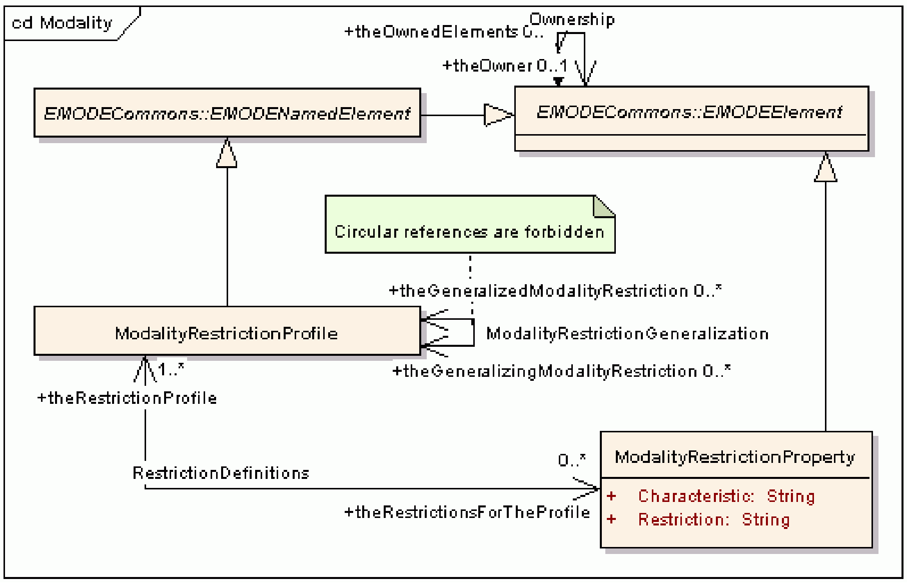
This approach gives the developer the chance to fully control the outcome of the adaptation and gives transparency to the developer which situation will produce what changes to the model(s) and the application thereby. Especially for UIs this is important, since the UI may then be modeled (designing being a special case of modeling) by some UI specialist (UI designer), who will produce more comfortable UIs than automatic means.
Furthermore, under certain conditions it might not be possible (too difficult or work intensive) to implement automatic transformations to implement adaptation (e.g. there are too many possible ways that an application needs to be adapted), then modeling at design time is the only way to implement adaptation within the modeling phase.
Since the only models available at runtime using this approach are the ones modeled out, this approach will save storage and computational power.

Disadvantages

Modeling at design time does not support open adaptivity, since parameters might not be known in advance or all possible situation that these context parameters occur in are unknown. Instead of open adaptivity, every possible situation and adaptation mechanism must be clear to the developer when designing her application.

If there are many model elements which need adaptation, the modeling phase will take very long if everything is modeled out explicitly. Therefore the model-out-everything-approach will render application development inefficient and too expensive, if done over a set of many dependent context parameters.

Usage in EMODE



cd Modality
+theOwnedElements 0..*
Ownership
+theOwner 0..1
EMODECommons::EMODENamedElement
EMODECommons::EMODEElement
Circular references are forbidden
+theGeneralizedModalityRestriction 0..*
ModalityRestrictionProfile
ModalityRestrictionGeneralization
+theGeneralizingModalityRestriction 0..*
1..*
+theRestrictionProfile
RestrictionDefinitions
0..*
+theRestrictionsForTheProfile
ModalityRestrictionProperty
+ Characteristic: String
+ Restriction: String

Fig 12 : Modalityrestrictions in EMODE Metamodel

In EMODE several meta-model elements have been provided to support the explicit modeling according to different situations (e.g. see AUIInteractor and its modalityrestriction attribute, Fig 12). However, since meta-models are not part of D2.3 we refer to D2.2 and D2.4 in which the EMODE meta-model is to be detailed further.

1.2.3.2 Transformations at Design time

Adaptation may be modeled using design time transformations by writing transformations that change the source model at design time, to fit all (or a subset thereof) possible situations, that may occur during runtime.

Normally, context parameters are not dependent on each other, e.g. the application will normally not change tremendously, if the illumination changes while it will take into account being inside or outside a building. In this case, it will be sufficient to write transformations, that are described in chapter 1.1.

However, there are context parameters that do depend on each other, e.g. the display size and an existing amblyopia of the user (the GUI elements might have to be displayed larger) or the illumination (because of bad/low illumination the GUI elements might have to be displayed larger). The number of situations will increase exponentially over the number of dependent context parameters, and the modeling process might be slowed down significantly, if every situation will be modeled out explicitly.

Definition 4 An exponential increase of model elements over a certain set of parameters *X* is called *model explosion* over the set *X*.

Since dependent context parameters impose an exponential increase of model elements and thereby of model variants they impose model explosion.

Transformations automate modeling using algorithms (imperative transformation language or programming languages) or descriptions of “model states” (descriptive transformation language or programming language). Due to automation the length of time of the development phase is reduced. Particularly, automation is of interest when developers are not able to model out every model element for every situation, because the amount of model elements involved would be too large, as it is the case for model explosion.

However, to circumvent model explosion, transformations at design time must be written in a descriptive manner, because otherwise the exponential nature of the modeling process is just shifted into the transformation definition. Using a descriptive transformation language such as QVT Relations and a smart transformation definition, which is composed of small transformation steps as well as appropriate platform models it is possible to describe the outcome of the transformation process using QVT Relations (descriptive) and the transformation steps will be described iterative (this reflects the current state of research of the EMODE project on runtime transformations).

Advantages and disadvantages of this approach are noted in Fig 11 along the bar titled “Development”.

Advantages

Besides the advantage that the time to model is reduced, design time transformations enable shorter adaptation times at runtime and therefore might increase the usability of the application (time to react) and feasibility in certain situations, since the models have already been transformed before, at design time, and don’t need to be computed at runtime.

Disadvantages

A disadvantage can be that too many different precomputed models must be stored on small (ubicomp) devices.

The major disadvantage with all generated user interfaces is that they are vulnerable to usability problems. It is known that generated user interfaces generally do not include enough usability guidelines (which are hard to formulate in terms of mathematical logic and therefore programming languages) [17].

Furthermore, since a declarative language is used it is normally hard to tell whether the executing algorithm will terminate or not.

Usage in EMODE

Transformations at design time are described using QVT Relations (see chapter 2 for details) and may fulfill the following purposes (most of them may be implemented using different techniques, too), which are not restricted to adaptation in EMODE (For a complete definition of model transformations specifications see D3.1 Part 1, chapter “Specifying Transformation Definitions”.):

- Generate new model elements (generative transformation)
- Give an aid to the developer to support him in implementing the EMODE methodology
- Update model elements
- Check consistency of models
- Combine models to form a new model
- Circumvent model explosion

These tasks do not exclude each other, but might be tasks that are orthogonal.

The use of some of the EMODE design time transformations is part of section 1.1.

1.2.3.3 Transformations at Runtime

IEMODE applications need to react to situation changes by changing behavior or the user interfaces. In EMODE a small part of applications will need to adapt to situation changes, which are not foreseeable at design time. These adaptations include transformations at runtime.

Advantages

Using runtime transformations open adaptivity may be implemented on models. Transformation rules will normally be written at design time although there might exist situations when the use of automatically generated transformation rules seems possible. Since the possible adaptations don't need to be stored as models during runtime, runtime transformations save storage dependent on the number of depending context parameters, which might help deploying EMODE applications on small devices with small storage capacity.

Disadvantages

The time to react to context changes may be very long, since possible adaptations have to be computed. Since the adaptation mechanism will use lots of computation power, memory and energy, it might not scale well on small devices. Usability may be low since the models are generated automatically.

1.2.4 EMODE Runtime Transformations

In EMODE we want to combine all three techniques, modeling, transformations at design time and at runtime to give the developer the choice to implement adaptation the way which best suits the requirements of the user interfaces of the application.
This is achieved by providing model editors to edit abstract user interfaces with abstract user interface interactors, which then may be transformed to include more platform specific interactors (this transformation is called a "refinement").

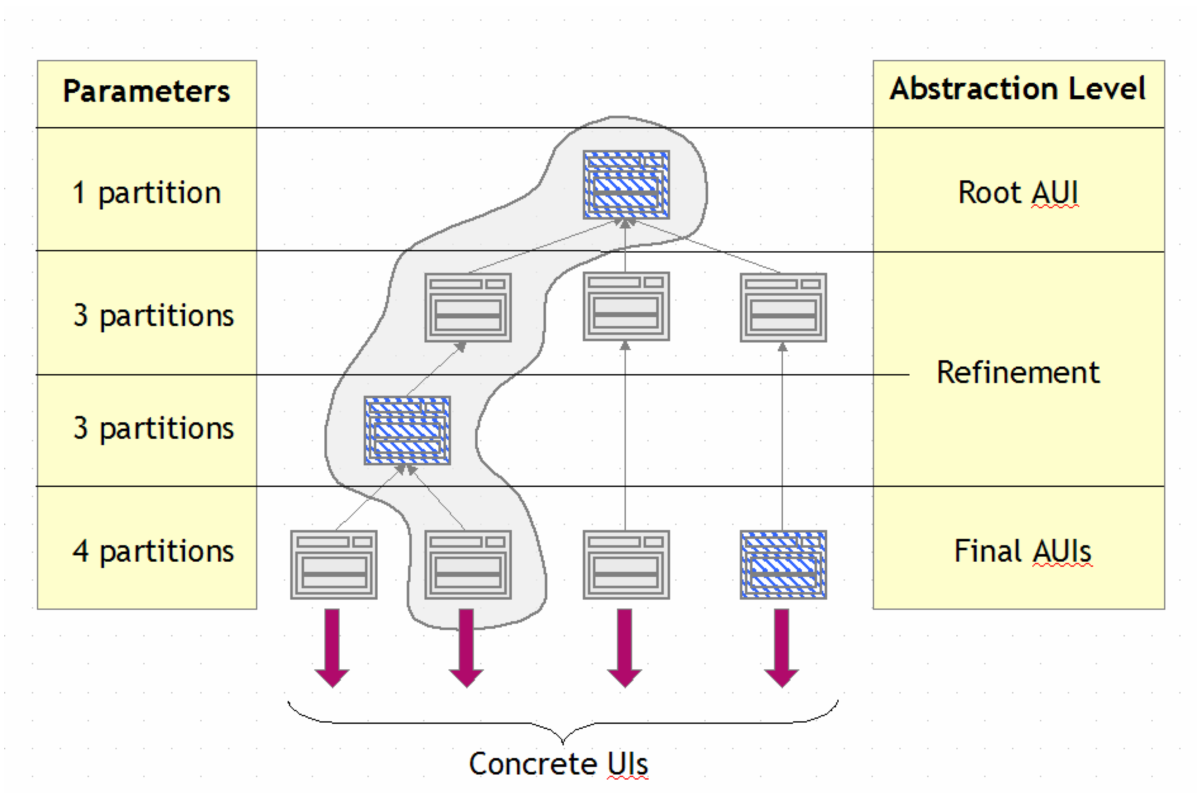


Fig 13 : UI refinement

In Fig 13 a refinement is shown as a small arrow pointing to an AUI, denoted by a small gray box (with has some interactors in it, denoted by darker gray boxes). The arrow notes which more concrete AUI is derived from other AUIs by refinement. Final AUIs can be derived from parent AUIs by adding additional properties to their “RefinedProperties” attributes (these attributes are platform specific information such as modality to be used and/or size of the component, etc.) and (to support situation changes) to its “NeededModalityRestriction” attribute (this property contains information about the situation, which this AUI is intendend to be used for). The state of a final AUI is reached, if the user interface will contain enough information to be displayed (displav may either mean output or input), and the developer doesn’t want to refine the interactors any more. Both attributes may be seen in Fig 14.
A small example would be the choice of modality for an AUIInteractor for new newly introduced modalities, e.g. voice. The runtime adaptation process will set the appropriate Property (within “RefinedProperties”) to voice.
Again, one may question the use of transformations at runtime, since adding model elements may either be done by transformation or by hand at design time, too. However, refinements may be needed, which are unknown at design time, since there might be too many situations to be stored within the model or there might be refinements needed which are not known to the developer, because the parameters of the dialoguespace of a newly introduced device have not been taken into account at design time.

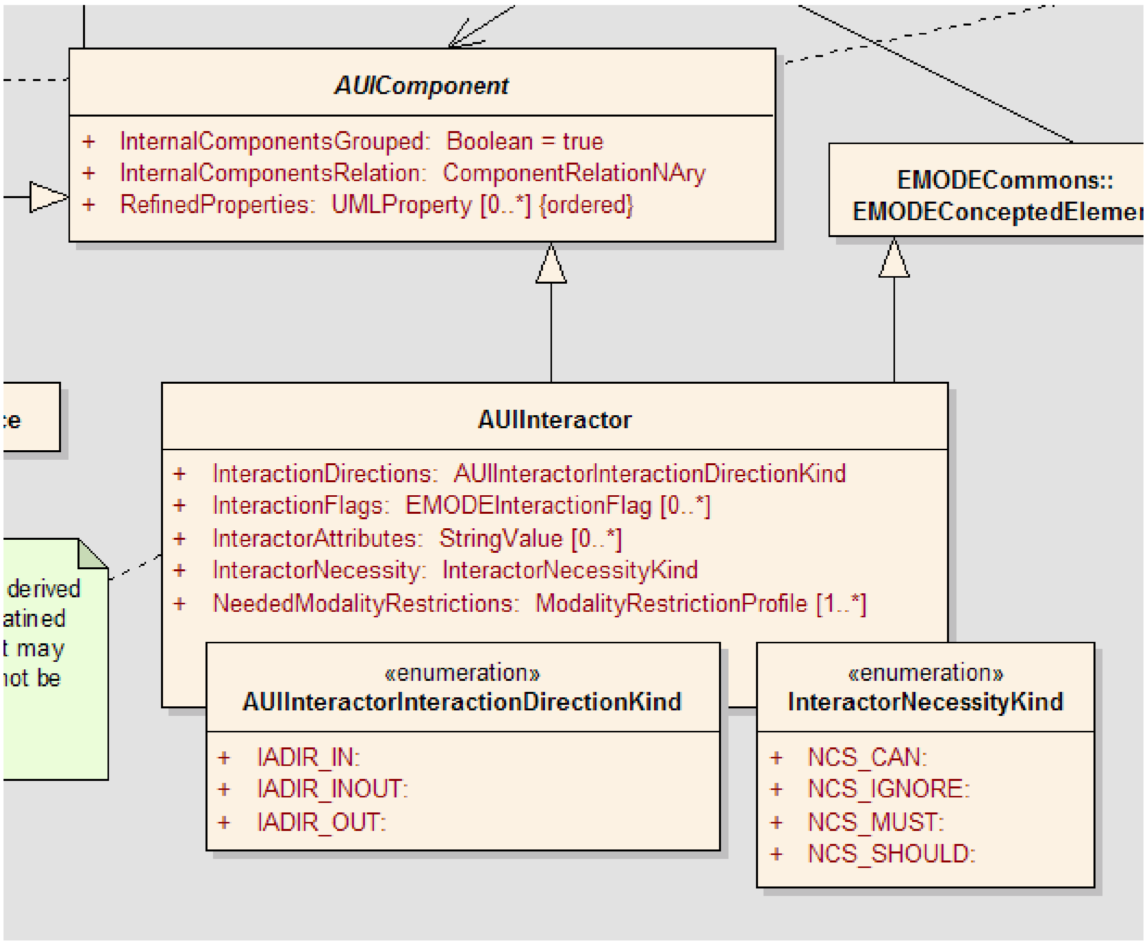

AUIComponent
+ InternalComponentsGrouped: Boolean = true
+ InternalComponentsRelation: ComponentRelationNAry
+ RefinedProperties: UMLProperty [0..*] {ordered}
EMODECommons::
EMODEConceptedElemen
AUIInteractor
+ InteractionDirections: AUIInteractorInteractionDirectionKind
+ InteractionFlags: EMODEInteractionFlag [0..*]
+ InteractorAttributes: StringValue [0..*]
+ InteractorNecessity: InteractorNecessityKind
+ NeededModalityRestrictions: ModalityRestrictionProfile [1..*]
«enumeration»
AUIInteractorInteractionDirectionKind
+ IADIR_IN:
+ IADIR_INOUT:
+ IADIR_OUT:
«enumeration»
InteractorNecessityKind
+ NCS_CAN:
+ NCS_IGNORE:
+ NCS_MUST:
+ NCS_SHOULD:
derived
atined
t may
not be

Fig 14: AUIInteractor

Currently EMODE imagines to use a model of the Dialoguespace at runtime, which will enable EMODE applications to adapt their user interfaces at runtime.
Additionally to refinement, which only takes place within the attributes of interactors (see “RefinedProperties”), changes of the actual interactors must be performed, if EMODE is going to support pagination or other techniques which are known as “ui remoulding”. These changes will require that interactors are either to be removed, inserted or replaced. This can also be done by hand, by design time transformation and by runtime transformation. We suspect, that the scenarios presented in workpackage 4 do not require runtime transformations for ui remoulding, since the combinations of devices used by the application are known. However, future EMODE applications which will depend on several input and output modalities, which may change over time will probably require runtime transformations for ui remoulding.
Currently, there haven’t been any runtime transformations identified. At the current stage the EMODE project researches different representations of created models at runtime, to which runtime transformations may be applied. The EMODE project will address at least some of the following research goals:

- Definition of runtime transformations for UI remoulding
- Identification of ways of how to represent and transform UI models
- Selection of a way to represent and transform UI models
- Giving a ratio for selection between different UI adaptation modeling techniques

2 Transformation Language

In EMODE QVT Relations was chosen as model transformation language for design time transformations. Reasons include

- QVT is a standardized model transformation language, which is not only a de facto standard (as ATL), but a standard by the OMG which defined the standard on which the EMODE metamodel has been based on (MOF)
- QVT Relations is known to have a very high level of abstraction [7]

Besides MOF, QVT in its final state, will probably be an important standard in the area of MDA, because it is defined by the most frequently referred to standardization organization for MOF and UML, the OMG. As demanded in [13], the standard addresses the definition of model queries, model views and model transformations, which are MOF compliant.

- Queries are used to retrieve specific elements from a model in an ad-hoc manner to be able to process them afterwards.
- For accenting parts of a model, views can be defined. Therefore, one model is deduced from another model and unwanted model parts are hidden.
- The most important issue the QVT standard deals with, is the lack of standardized ways to define model transformations. While the standard is still work in progress, it has reached the state of a Final Adopted Specification. In fact, the normative document results from a combination of a list of submitted proposals.

The QVT standard defines both, declarative and imperative approaches for model transformation definition on the metalevel where metamodels are defined. According to [14] and [11], declarative languages focus on the definition of relationships between elements and the language processor applies a fixed algorithm to produce a result. In contrast, an imperative language provides constructs to explicitly define changes of the system state. In other words, the first defines what has to be changed and the latter how changes are computed.

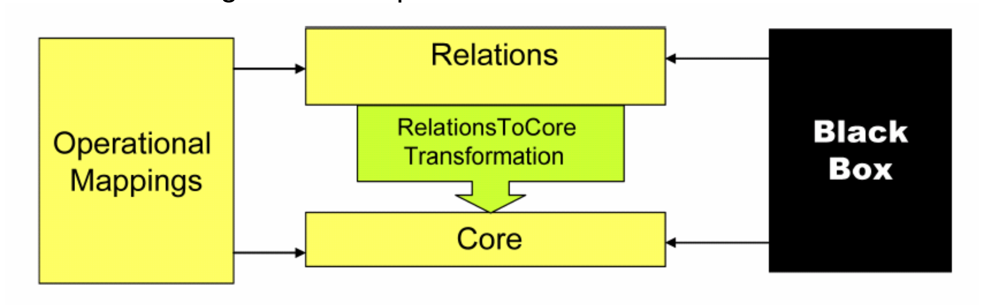


Fig 15 : QVT Languages and Their Relation [12]

Several partly dependent metamodels encapsulate the different methodologies. Fig 15 illustrates the relationships between those models. The picture is taken from [12]. All boxes must be seen in the context of QVT (that is: QVT Relations, QVT Core, …). The declarative parts are organized in two abstraction levels. While it provides little extensions to EMOF, QVT Core forms a small language for transformations. It can directly be implemented but also used as formal semantics for QVT Relations as both have the same expressiveness. Therefore, the standard defines a transformation from Relations to Core. As the language itself is very simple, it is easier to describe its semantics but transformation descriptions are less user friendly. Implementers must take care of traces* by themselves and the pattern definition for object matching only relies on flat variables.

As opposite to the lower level QVT Core, QVT Relations is more suited for end users, providing a higher level, purely declarative transformation definition language. It supports both, a textual and graphical syntax. Fig 16 shows the “UML Class to Relational Table” relation in graphical syntax and listing Fig 17 illustrates its textual pendant. Traces are implicitly created and QVT Relations allows the definition of complex object patterns for matching elements in a model.

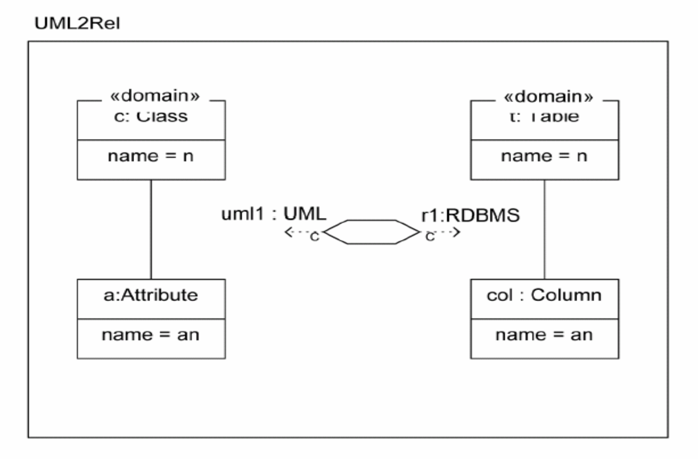
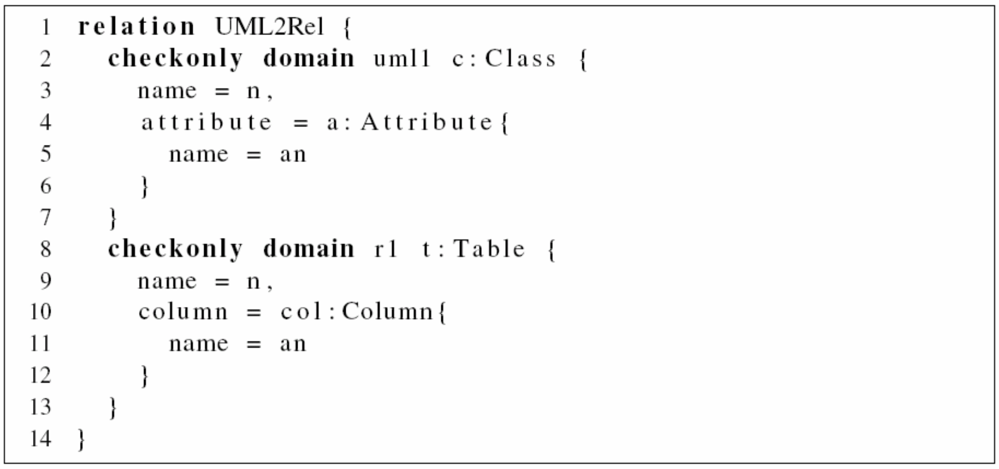


Fig 16: UML Class to Relational Table Relation - Graphical Notation [12]

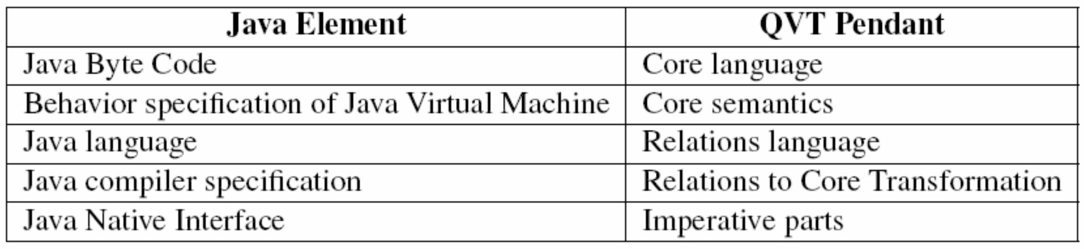


```
relation UML2Rel {
  checkonly domain uml1 c:Class {
    name = n,
    attribute = a:Attribute{
      name = an
    }
  }
  checkonly domain r1 t:Table {
    name = n,
    column = col:Column{
      name = an
    }
  }
}
```

Fig 17: UML Class to Relational Table Relation - Textual Notation [12]

In addition to the declarative possibilities QVT offers for transformation definition, two imperative approaches are supported. There is a standard syntax called Operational Mapping which comes with an imperative syntax similar to nowadays imperative programming languages. It can either be used to define complete transformations, which are not described easily using QVT Relations, or to be called from a Relations relation to express parts which are hard to describe in a declarative manner. Furthermore, QVT allows the definition of so called black-box implementations to specify a transformation in an arbitrary programming language. This is meant to use domain specific libraries for transformations. For better understanding, the QVT standard states an analogy between the different parts of QVT and the Java Virtual Machine. Table QVTJAVA shows the analogous elements.

* A trace model defines relationships between source and target model elements, i.e. an instance of a trace class specifies which element in the source model a target model construct originates from.



Java Element	QVT Pendant
Java Byte Code	Core language
Behavior specification of Java Virtual Machine	Core semantics
Java language	Relations language
Java compiler specification	Relations to Core Transformation
Java Native Interface	Imperative parts

Tab. QVTJAVA

The above mentioned parts of the QVT standard are reflected by its metamodel, which is depicted in figure Fig 18. On the one hand, the QVT metamodel is based on the EMOF package. On the other hand, it relies on the EssentialOCL package which provides a small OCL fundament, necessary to work with EMOF. Besides the main packages QVTCore, QVTRelation and QVTOperational which encapsulate the three language parts of QVT, some additional ones are defined [12]:

- QVTBase provides basic transformation elements.
- The QVTTemplate package is used by QVTRelation for the definition of template expressions.
- Imperative expressions for QVTOperational are defined in ImperativeOCL.

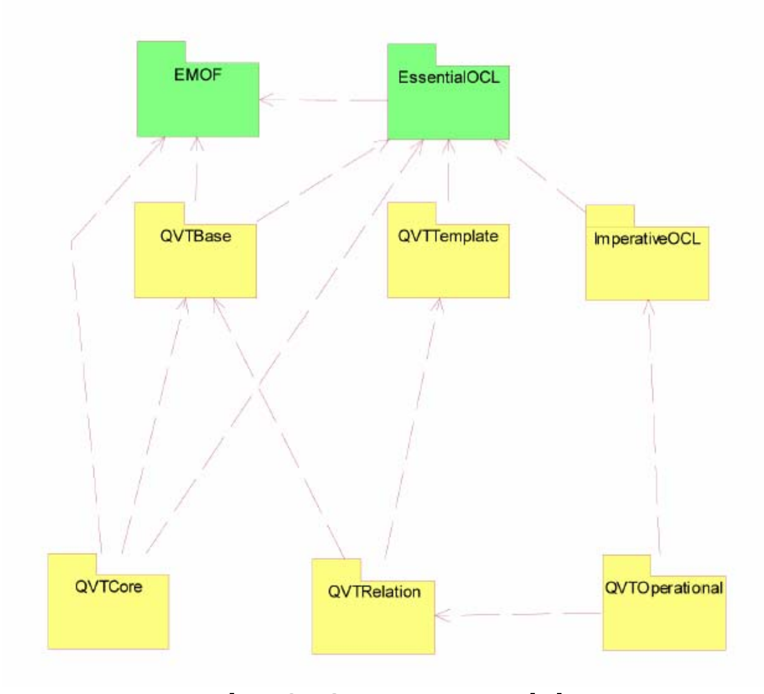


Fig 18: QVT Metamodel

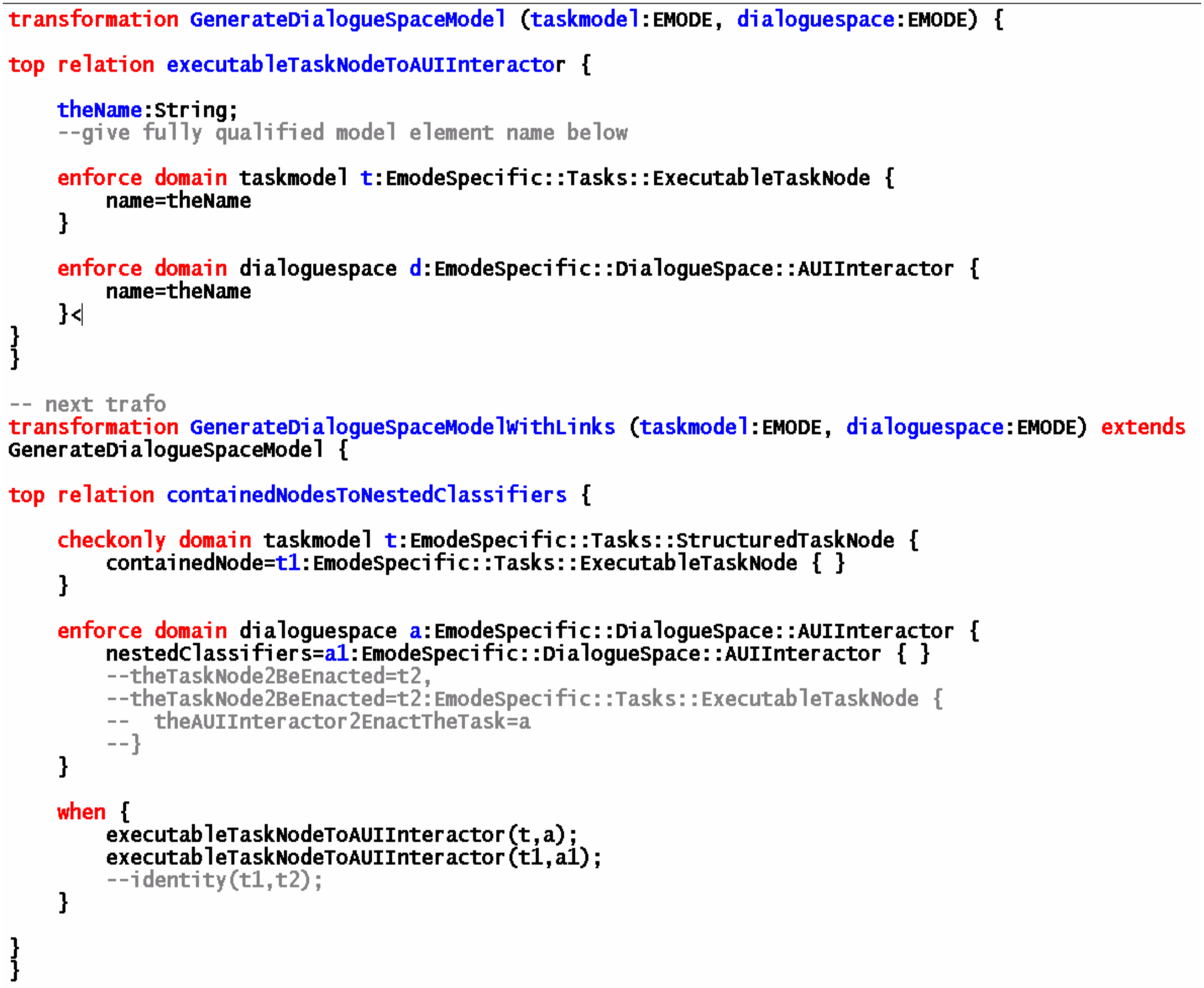
3 Exemplary Transformations

This section presents several exemplary EMODE transformations. The first part demonstrates a model-to-model example and a model-to-code one. The last section is concerned about giving the reader an idea on how to write design time transformation using QVT Relations.

3.1 *Transformation examples from EMODE*

3.1.1 Model-To-Model example

In chapter 1.1.1.5 the Tasks to DialogueSpace (also called “AUI”) transformation has been defined informally. As it is only a first version of this transformation, future enhancements to this transformation are planned to be integrated into the code of the transformation.



```
transformation GenerateDialogueSpaceModel (taskmodel:EMODE, dialoguespace:EMODE) {

top relation executableTaskNodeToAUIInteractor {

    theName:String;
    --give fully qualified model element name below

    enforce domain taskmodel t:EmodeSpecific::Tasks::ExecutableTaskNode {
        name=theName
    }

    enforce domain dialoguespace d:EmodeSpecific::DialogueSpace::AUIInteractor {
        name=theName
    }<
}
}

-- next trafo
transformation GenerateDialogueSpaceModelWithLinks (taskmodel:EMODE, dialoguespace:EMODE) extends
GenerateDialogueSpaceModel {

top relation containedNodesToNestedClassifiers {

    checkonly domain taskmodel t:EmodeSpecific::Tasks::StructuredTaskNode {
        containedNode=t1:EmodeSpecific::Tasks::ExecutableTaskNode { }
    }

    enforce domain dialoguespace a:EmodeSpecific::DialogueSpace::AUIInteractor {
        nestedClassifiers=a1:EmodeSpecific::DialogueSpace::AUIInteractor { }
        --theTaskNode2BeEnacted=t2,
        --theTaskNode2BeEnacted=t2:EmodeSpecific::Tasks::ExecutableTaskNode {
        --  theAUIInteractor2EnactTheTask=a
        --}
    }

    when {
        executableTaskNodeToAUIInteractor(t,a);
        executableTaskNodeToAUIInteractor(t1,a1);
        --identity(t1,t2);
    }

}
}
```

Fig 19: GenerateTaskModel

In Fig 19 the transformations needed to transform the model elements used within the task diagram to model elements used in the dialoguespace diagram are defined formally (syntactically correct) as they are used within the German EMODE project.

Fig 20 is a graphical representation of the transformations. Since the QVT standard is somewhat incomplete in terms of its graphical notation, we only present a rough outline for it and defer further improvements to the point until the full specification has been worked out. As one can see, the whole transformation definition consists of 2 transformations used at different stages of the development process.
In the following, an explanation for the example is given. Lines commented out as seen in Fig 19 indicate possible implementation options to be evaluated.

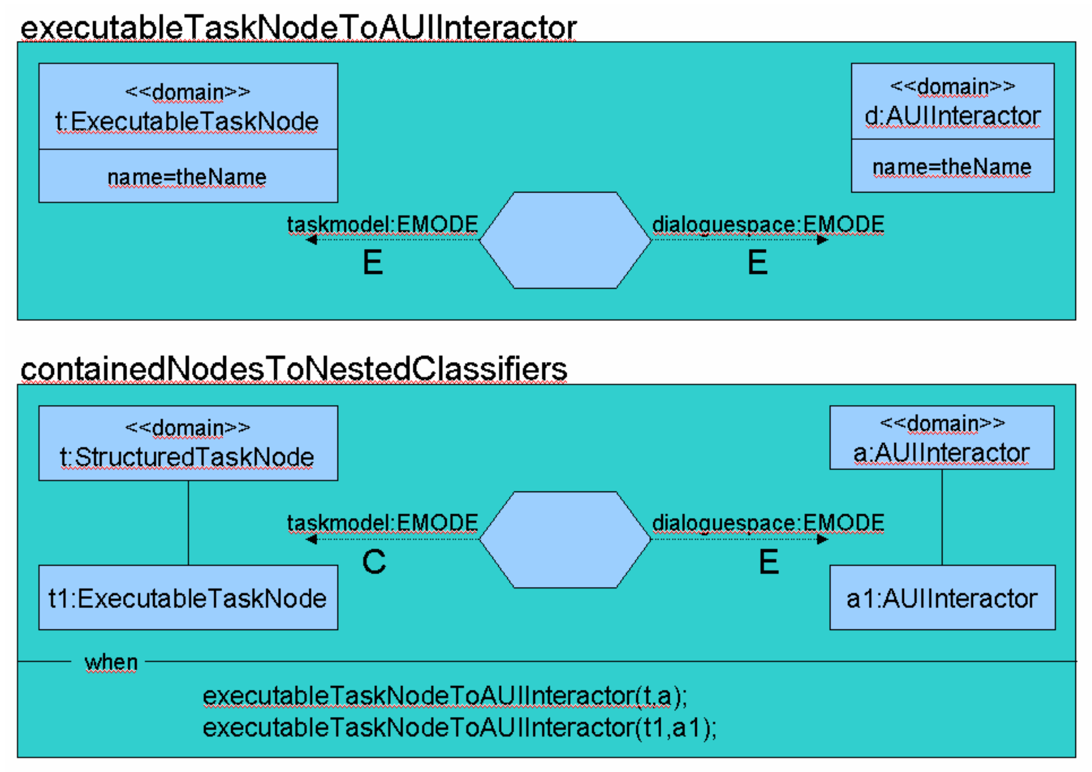


Fig 20: QVT graphical notation of GenerateDialogueSpace

3.1.1.1 Transformations

The keyword transformation starts a definition of a QVT Relations transformation followed by its name, here “GenerateDialogueSpaceModel” and the metamodels used for input and output, which will be “EMODE” for all transformations that are to be used in conjunction with the EMODE project. A transformation is enclosed in brackets.
As it may be seen easily, a second transformation is defined “GenerateDialogueSpaceModelWithLinks”, which is derived from GenerateDialogueSpaceModel by adding the keyword extends.
A developer would then be able to call 2 different transformations from the transformation frontend. The first one is a more general one, which will work both ways, from task to dialogspace and the other way round. But the second one would be used to generate the first version of the dialoguespace model from the task model and is therefore not to be applicable for transforming from dialoguespace to task.

3.1.1.2 Relations

Within a transformation definition, the transformation implementer will define relations. Here, for each transformation one top relation, has been defined, which

means that those relations must hold on the top level of the definition, that is it may be called by the transformation engine and its user. Relations that are not marked as top may be called from other relations, only. “executableTaskNodeToAUIInteractor” and “containedNodesToNestedClassifiers” are defined as relations in the example. QVT Relations is not restricted to one top relation per transformation, so more relations can be defined and called later on using a transformation frontend.
Using graphical notation (Fig 20) the name of the relations is written above the boxes which represent one relation, each. As one can see, the name of the transformation is missing, and therefore the textual representation cannot be derived fully from the graphical one, at the moment.

3.1.1.3 Variables

In addition to the definition of the domains, free variables are defined on the same level within a definition of a relation. Those variables may then be used within OCL statements (e.g. in the domains block, see 3.1.1.4). The OCL statements are evaluated by the transformation engine. Informally explained, if OCL statements contain variables that are not filled with content yet (and the variable occurs in a definition where content is available (e.g. source domain)), they will be filled with content from the current domain specification they reside in, such that the OCL expression evaluates to true. If they already have content (and the variable occurs within an OCL statement that needs input (e.g. within the target domain block)), this content is used within the OCL statements, again, such that the OCL expression evaluates to true.
In the executableTaskNodeToAUIInteractor example a free variable “theName” has been defined to be of type String.

3.1.1.4 Domains

Within a relation, domains must be used to represent sets (these sets are called “object templates” within the QVT relations domain) with all model elements that adhere to the OCL expressions (OCL expressions must evaluate to true), that have been enclosed in brackets after the start of the definition of the domain (its corresponding block).
Within the relation named “GenerateDialogueSpaceModelWithLinks” 2 domains exist:

- A domain named “taskmodel” which contains model elements of type StructuredTaskNodel
- A domain named “dialoguespace” which contains model elements of type AUIInteractor

The Domain dialoguespace is marked as to be enforced, which means that it may be changed by the transformation engine, until the relation does hold (that is, all OCL expressions evaluate to true). Taskmodel is marked as “checkonly” such that it may not be changed, but just checked whether the OCL expressions in its corresponding block hold. (OCL is used by QVT, but it is defined as a separate language, well known to metamodel experts, and because of its lengthiness, not presented here) Informally, the OCL expressions within the domain specifications require, that for each model element of type StructuredTaskNode that has children an AUIInteractor exists, that has the name that has been assigned to the free variable from the StructuredTaskNode block (see when clause) and has children conforming to those of the StructuredTaskNode.

3.1.1.5 Object templates

Object templates are used to search and select elements of a special type of model elements or to produce them, depending on the context those templates appear in. An object template is defined using a type definition resembling its fully qualified name in the metamodel and a set of OCL expressions, which are to be enclosed in brackets. Each of these expressions must evaluate to true, regardless whether these templates are used in target or source domains.
In a template that is used for searching or selection by the transformation engine model elements will be checked to conform to the expressions of the template.
In a template that is used for production of model elements by the transformation engine, model elements will be produced, such that they are conforming to the OCL expressions of the template.

3.1.1.6 When

“When” clauses reference relations that must hold, if the relation should hold that this “when” clause appears in. This allows for definition of complex relations. Since GenerateDialogSpaceModelWithLinks extends GenerateDialogueSpaceModel the relation executableTaskNodeToAUIInteractor can be requested to hold in the “when” clause. The 2 ExecutableTaskNodes and AUIInteractors used as parameters, are checked for equal names, otherwise the “when” clause will not hold for these model elements and therefore the relation doesn’t need to hold. This ensures, that only those model elements which have the same name as the children of a specific ExecutableTaskNode are children of the corresponding AUIInteractor.
“When” clauses may contain additional OCL expressions.

3.1.1.7 Where

The “where” clause (which is not present in the example) ensures for given relations in the “where” clause, that the corresponding relation must hold for all model elements that are given as parameters to this relation (if the calling relation will hold).
“Where” clauses may contain additional OCL expressions.

3.1.2 Model-To-Code example (FCA Diagram)

Although JET is used to define model-to-code transformations, we will not present template source code here, but give a short overview about the algorithm performing the transformation. We chose the FCA model to demonstrate an EMODE JET model-to-code transformation.

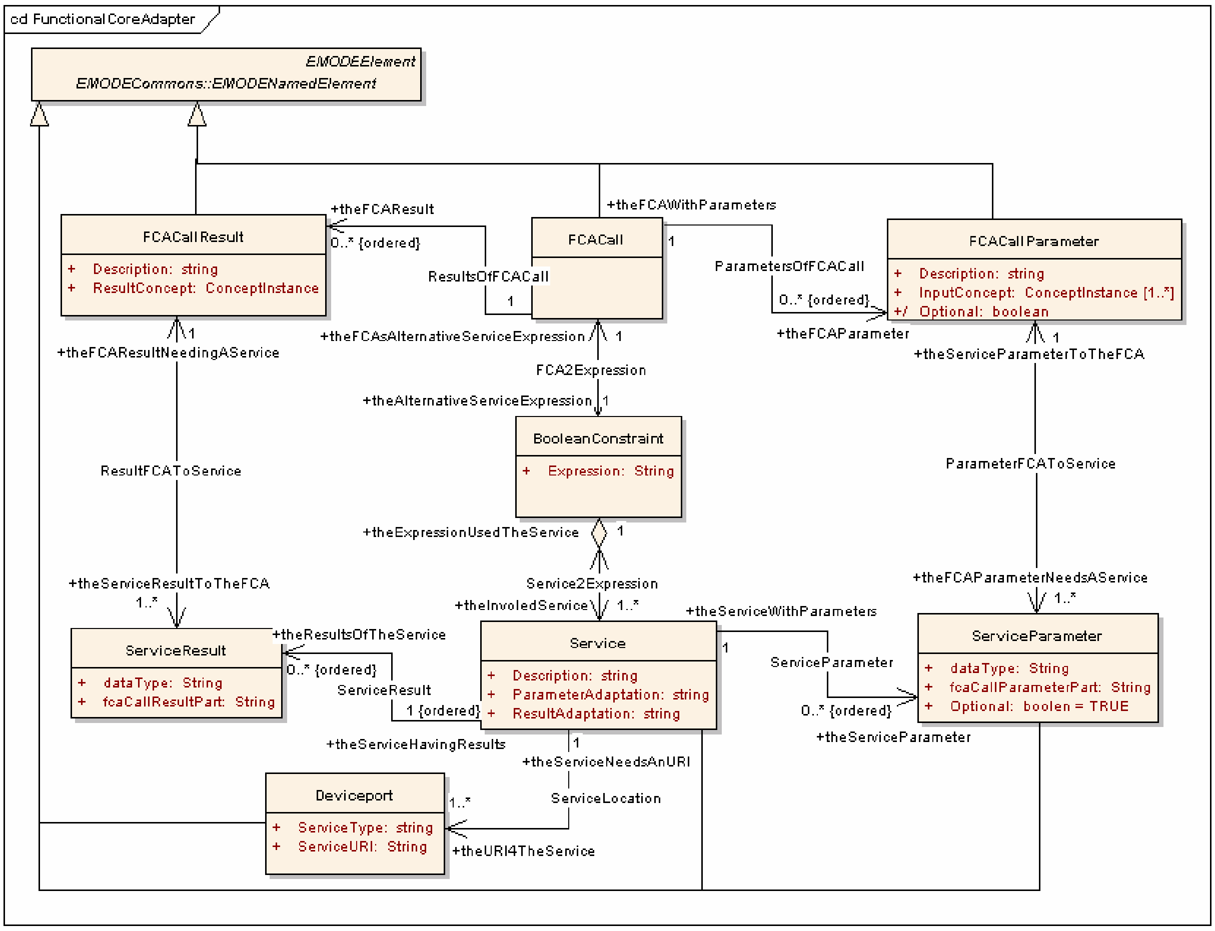


Fig 21: the FCA Model

Fig 21 shows the current FCA meta model, on which the transformation definition is based. To illustrate the transformation the example shown in Fig 22 is used.
The example presents one FCA instance (isEmployee_FCA) which has two alternative services (BooleanConstraint.Expression = “isEmployee_S1 v isEmployee_S2”). The example’s Parameter and Result mapping can be found in table FCAServ. In Fig 22 one can see the name in the first part of each box, the parameters can be seen in the second and the result in the third.

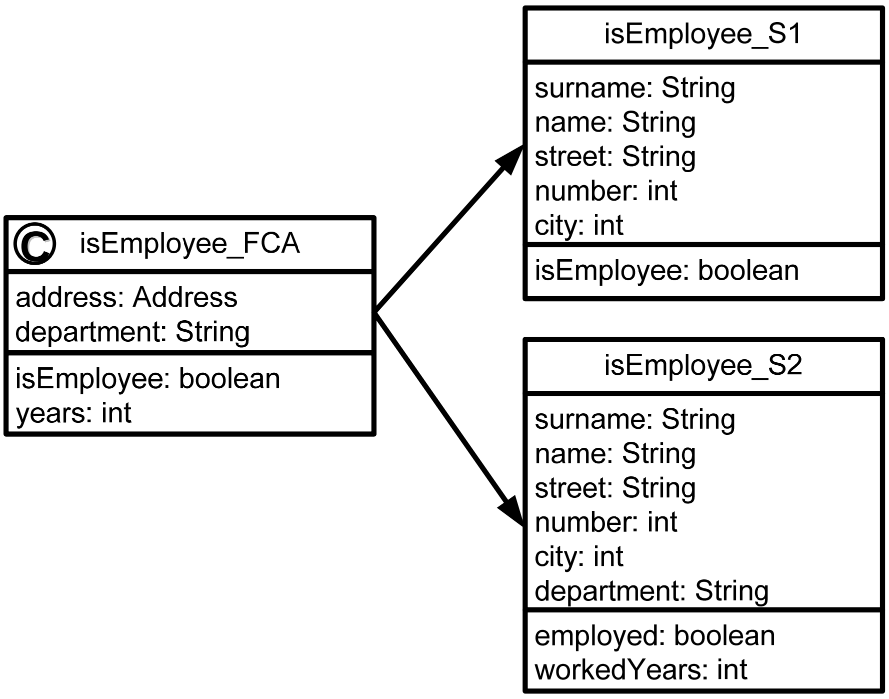


Fig 22: example FCA model

IsEmployee_FCA	IsEmployee_S1	IsEmployeeS2
address.surname	surname	Surname
address.name	name	Name
address.street	street	Street
address.number	number	Number
address.city	city	City
department.value	----------------	Department
isEmployee.value	isEmployee	Employed
years.value	----------------	workedYears

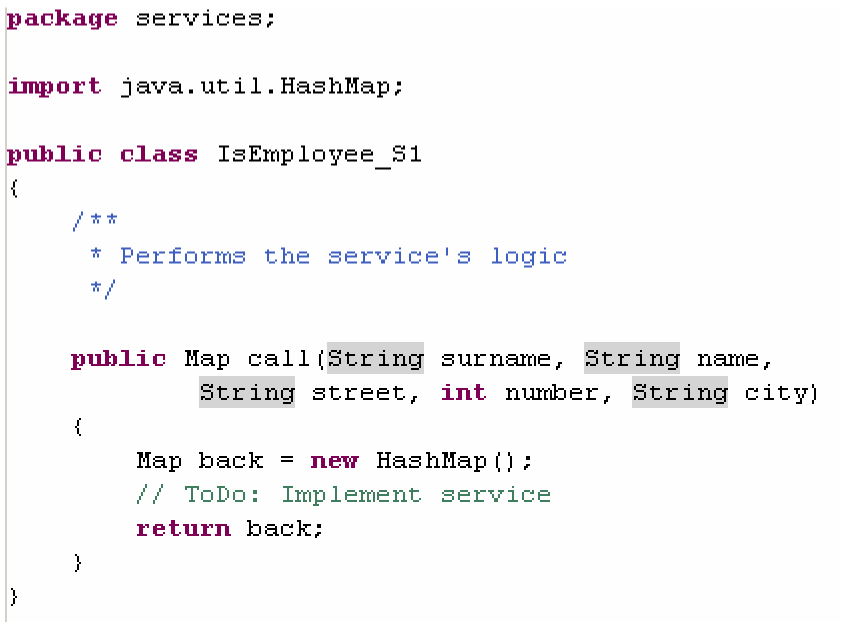
Table FCAServ: FCA – Service - Mapping

3.1.2.1 Services

Each service from the FCA model (see D2.2) is mapped on one Java class, having the service's name. In order to execute the service, a so called *call()* method exists, which has all the service's parameters as input and returns a map. This map contains all result values, registered by its name.
In the case of the example, the map would look as follows.

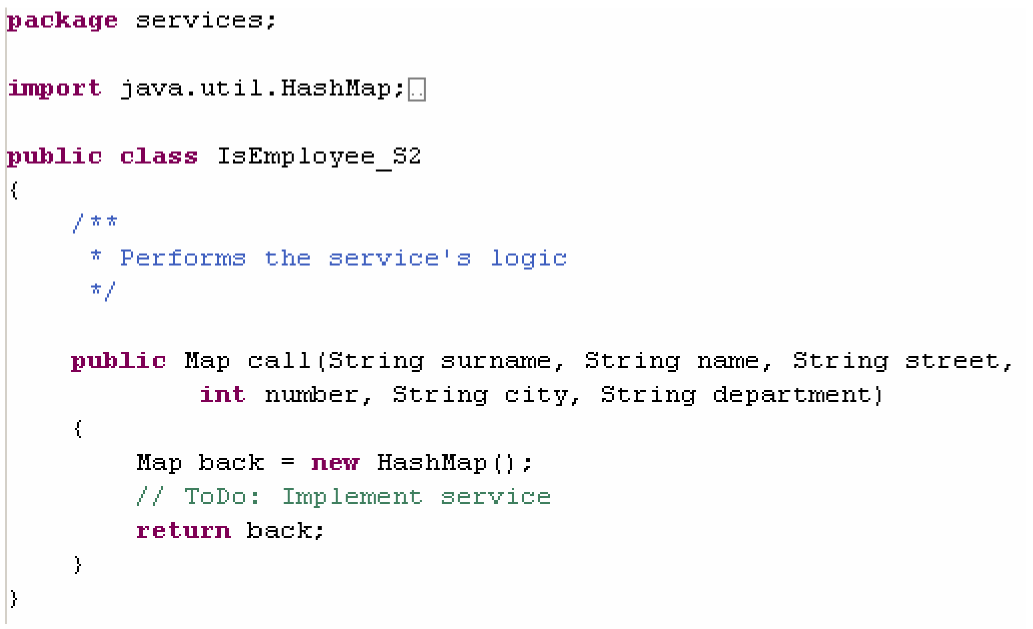
returnedMap_S1 = {(isEmployee, value)}
returnedMap_S2 = {(employed, value), (workedYears, years)}

Because the service's logic can't be stored in the model instance, the developer must write the *call()* method's implementation by hand.
Accordingly, the Java class of the service "IsEmployee_S1" and "IsEmployee_S2" looks like figure Fig 23 and Fig 24.



```java
package services;

import java.util.HashMap;

public class IsEmployee_S1
{
    /**
     * Performs the service's logic
     */

    public Map call(String surname, String name,
            String street, int number, String city)
    {
        Map back = new HashMap();
        // ToDo: Implement service
        return back;
    }
}
```

Fig 23: Java class of the service „IsEmployee_S1“



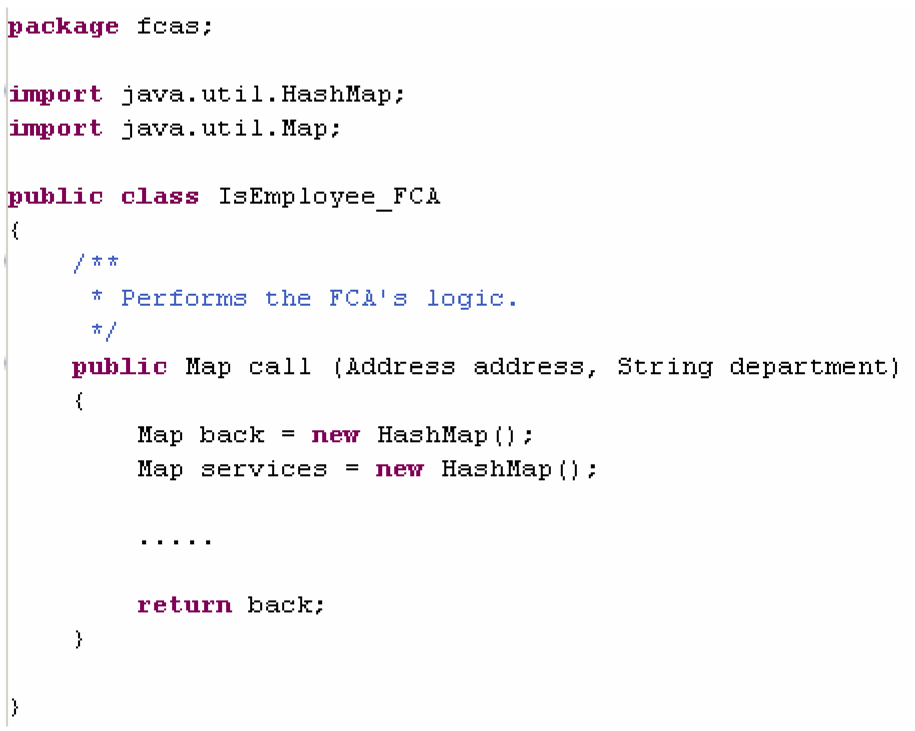
```java
package services;

import java.util.HashMap;

public class IsEmployee_S2
{
    /**
     * Performs the service's logic
     */

    public Map call(String surname, String name, String street,
            int number, String city, String department)
    {
        Map back = new HashMap();
        // ToDo: Implement service
        return back;
    }
}
```

Fig 24: class of the service „IsEmployee_S2“

In order to clearly separate the FCAs and Services, each of them is saved in a special package, called “services” and “fcas”.

3.1.2.2 FCAs

Just as the Service, each FCA is also mapped on one Java class, having the name of the FCA element instance and the method *call(),* in order to execute it. Fig 25 shows the FCA instance’s class.



```
package fcas;

import java.util.HashMap;
import java.util.Map;

public class IsEmployee_FCA
{
    /**
     * Performs the FCA's logic.
     */
    public Map call (Address address, String department)
    {
        Map back = new HashMap();
        Map services = new HashMap();

        .....

        return back;
    }

}
```

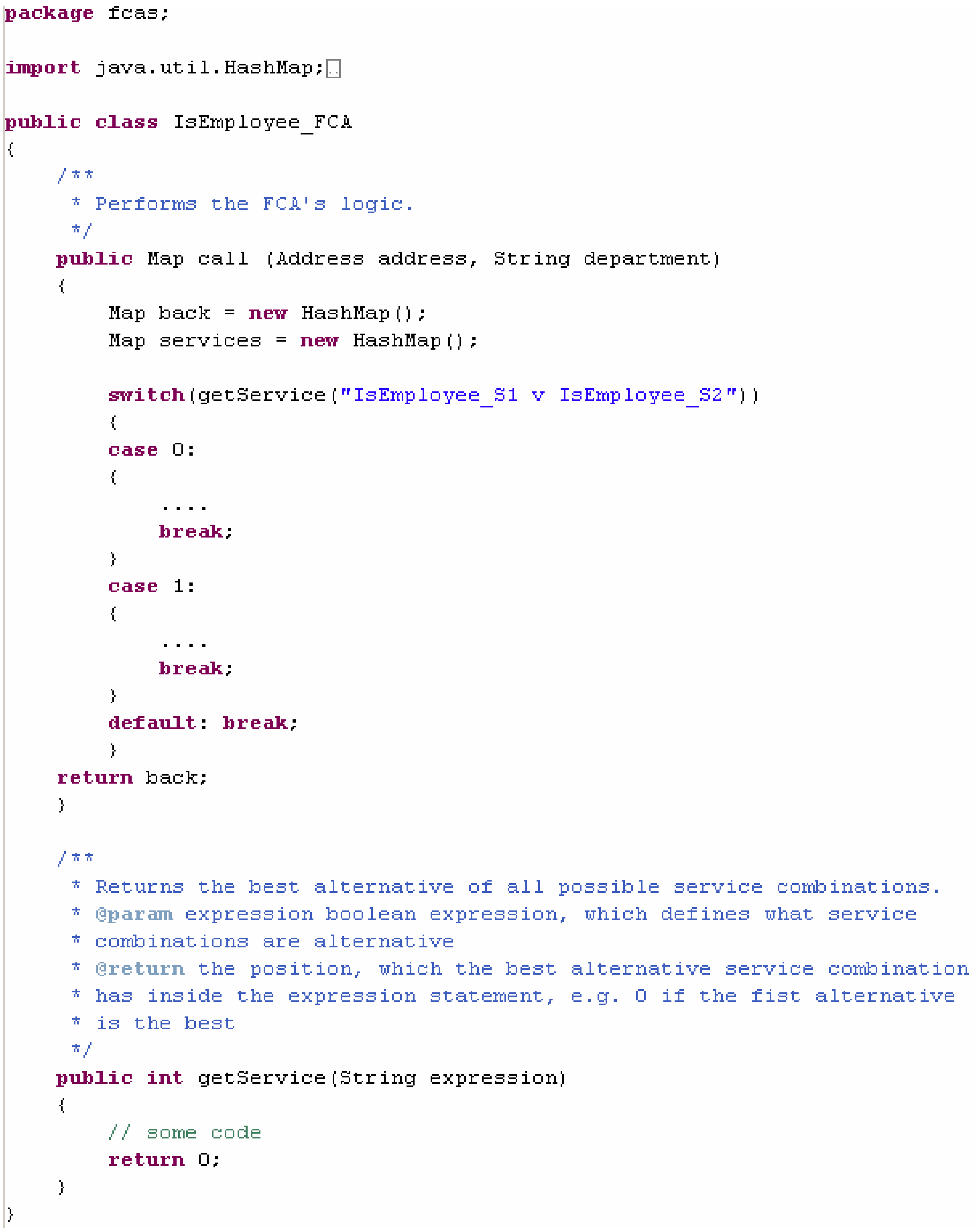
Fig 25: the FCA's Java class – part 1

Unlike Services, the implementation details of the FCA's call method are contained in the model. The link between the two components as well as its results and parameters allow to clearly allocate

1. which services belong to one FCA
2. how can one FCA result be derived from several services
3. which FCA parameter acts as input for which service.

This information is sufficient to implement the call methods body inside the FCA's Java class.

At first the different service combination alternatives must be extracted from BooleanConstraint.expression. In the example it has the value "isEmployee_S1 v isEmployee_S2" that means, the two services are equivalent and can replace each other. In order to support this behaviour the *call()* methods consist of a switch statement, which contains one case statement for each alternative service combination. While executing, the FCA will choose the best one using the not further specified method "getService()". Fig 26 presents the FCA's java class extended with this implementation details.



```java
package fcas;

import java.util.HashMap;

public class IsEmployee_FCA
{
    /**
     * Performs the FCA's logic.
     */
    public Map call (Address address, String department)
    {
        Map back = new HashMap();
        Map services = new HashMap();

        switch(getService("IsEmployee_S1 v IsEmployee_S2"))
        {
        case 0:
        {
            ....
            break;
        }
        case 1:
        {
            ....
            break;
        }
        default: break;
        }
    return back;
    }

    /**
     * Returns the best alternative of all possible service combinations.
     * @param expression boolean expression, which defines what service
     * combinations are alternative
     * @return the position, which the best alternative service combination
     * has inside the expression statement, e.g. 0 if the fist alternative
     * is the best
     */
    public int getService(String expression)
    {
        // some code
        return 0;
    }
}
```

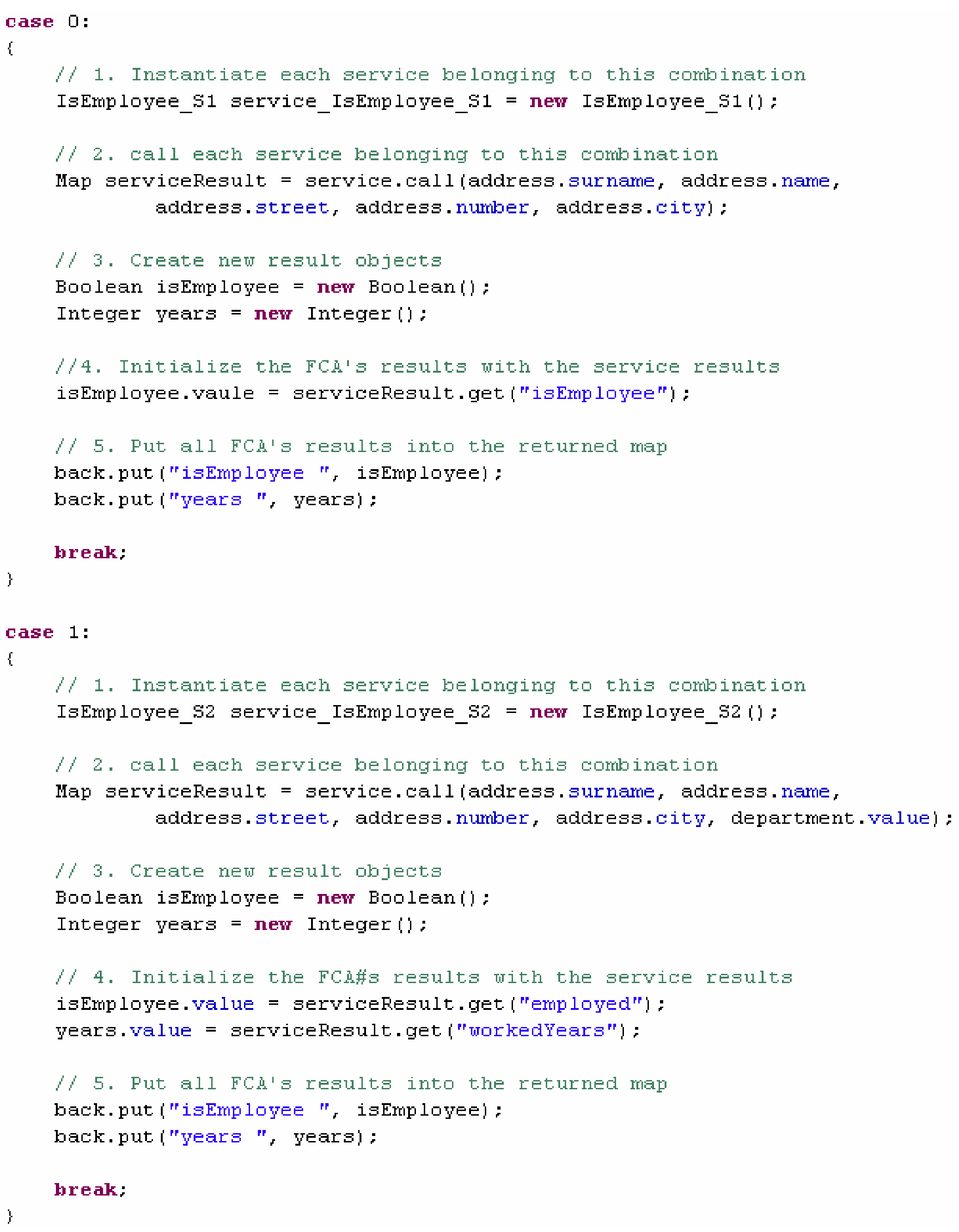
Fig 26: the FCA’s Java class – part 2

Each case statement is filled with the services belonging to this alternative service combination. That means:

1. A local service instance of all services belonging to this alternative must be created if it does not already exist.
2. Each service must be executed by calling the service’s *call()* method. The parameter’s value of the service’s *call()* method parameter can be read from its attribute “fcaCallParameterPart”. At this point the developer defined which part of the FCA’s input concept will be mapped on the service parameters.

3. Create a new Object for each FCAResult element
4. Initialize the FCA’s result objects with the correct service results. The value of the service result’s attribute “fcaCallResultPart” indicates which FCA result value will be initialized with which service result.
5. Put all FCA’s results into the returned map registered with the name of the corresponding result.
6. Return the map

Fig 27 shows the case statements implementation for the Fig 22 example and completes the FCA’s Java class.

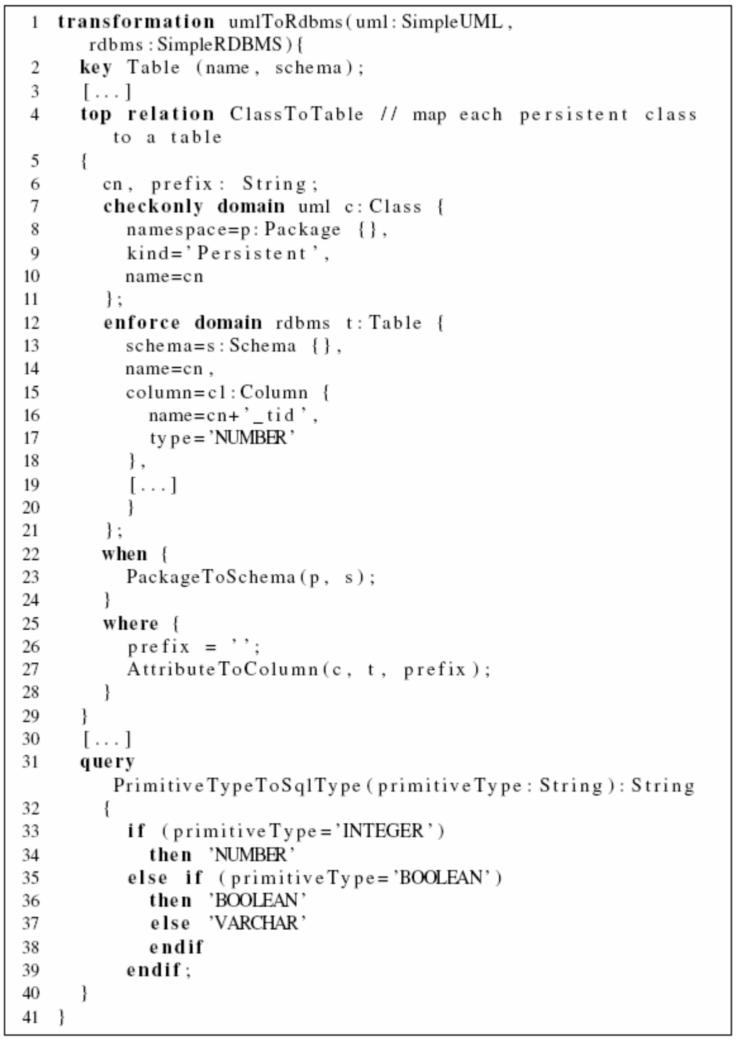


```
case 0:
{
    // 1. Instantiate each service belonging to this combination
    IsEmployee_S1 service_IsEmployee_S1 = new IsEmployee_S1();

    // 2. call each service belonging to this combination
    Map serviceResult = service.call(address.surname, address.name,
            address.street, address.number, address.city);

    // 3. Create new result objects
    Boolean isEmployee = new Boolean();
    Integer years = new Integer();

    //4. Initialize the FCA's results with the service results
    isEmployee.vaule = serviceResult.get("isEmployee");

    // 5. Put all FCA's results into the returned map
    back.put("isEmployee ", isEmployee);
    back.put("years ", years);

    break;
}

case 1:
{
    // 1. Instantiate each service belonging to this combination
    IsEmployee_S2 service_IsEmployee_S2 = new IsEmployee_S2();

    // 2. call each service belonging to this combination
    Map serviceResult = service.call(address.surname, address.name,
            address.street, address.number, address.city, department.value);

    // 3. Create new result objects
    Boolean isEmployee = new Boolean();
    Integer years = new Integer();

    // 4. Initialize the FCA#s results with the service results
    isEmployee.value = serviceResult.get("employed");
    years.value = serviceResult.get("workedYears");

    // 5. Put all FCA's results into the returned map
    back.put("isEmployee ", isEmployee);
    back.put("years ", years);

    break;
}
```

Fig 27: The Case Statements Implementation

3.2 A Small Guide for Writing QVT Transformations

This section shall give the reader an introduction on how to write transformations in QVT Relations and therefore in the EMODE project. Main elements of the QVT Relations language are described while referring to an example present in the QVT standard, already mentioned above, the umlToRdbms (see Fig 28) example. Line numbers mentioned below are referring to the example. More details on the QVT Relations language can be found in [12].



```
1  transformation umlToRdbms(uml:SimpleUML,
     rdbms:SimpleRDBMS){
2    key Table (name, schema);
3    [...]
4    top relation ClassToTable // map each persistent class
       to a table
5    {
6      cn, prefix: String;
7      checkonly domain uml c:Class {
8        namespace=p:Package {},
9        kind='Persistent',
10       name=cn
11     };
12     enforce domain rdbms t:Table {
13       schema=s:Schema {},
14       name=cn,
15       column=cl:Column {
16         name=cn+'_tid',
17         type='NUMBER'
18       },
19       [...]
20       }
21     };
22     when {
23       PackageToSchema(p, s);
24     }
25     where {
26       prefix = '';
27       AttributeToColumn(c, t, prefix);
28     }
29   }
30   [...]
31   query
       PrimitiveTypeToSqlType(primitiveType:String):String
32     {
33       if (primitiveType='INTEGER')
34         then 'NUMBER'
35       else if (primitiveType='BOOLEAN')
36         then 'BOOLEAN'
37         else 'VARCHAR'
38         endif
39       endif;
40   }
41 }
```

Fig 28: QVT Relations example in QVT final adoption

A transformation definition begins with the **transformation** keyword. The models, involved in the transformation, are defined as its parameters. The number of models being supplied as parameters is not restricted to two; in fact, QVT Relations supports an arbitrary number of models greater than two and in-place transformations, where source and target models are the same, too.

For the purpose of identification of objects MOF2 allows to define an identifying property. The QVT standard states that this is not enough for the majority of metamodels. But a clear identity concept is necessary for a transformation engine to allow updates on objects already created. The QVT Relations language introduces keys to define a set of identifying properties for a class. Line 2 defines a **key** for class Table consisting of its attributes name and schema. QVT Relations transformations rely on the definition of relations between elements of source and target models

which are defined on their metamodel types. Based on these relations a transformation engine can manipulate a target model.
Within a relation definition which starts with the **relation** keyword, variables of complex and primitive type can be used to refer to model elements or their values. Line 6 depicts the definition of two variables “cn” and “prefix” of type String. As they are not bound to a value at this moment, these variables are called free. The ClassToTable relation is marked as **top**. All top-level relations must hold when a transformation is executed. On the other hand, non-top-level relations are required to hold when invoked by other relations only.

> *“A **domain** is a distinguished typed variable that can be matched in a model of a given model type. A domain has a pattern, which can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain’s type. Alternatively a pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must satisfy to qualify as a valid binding of the pattern. A domain pattern can be considered a template for objects and their properties that must be located, modified, or created in a candidate model to satisfy the relation.” [12, p.13]*

The example defines two domains: “uml” and “rdbms”. Domains can either be marked as **checkonly** or **enforce**. For checkonly domains no objects will be created or modified in the target model. It will only be checked whether a relation is satisfied or not. In contrast, objects are created or modified if a domain is marked as enforce. This implies that a transformation which contains multiple domains marked as enforce will be a multidirectional transformation. So the example shows an unidirectional one. The rdbms model will only be modified as uml is marked as checkonly.
Line 7 to 10 shows the object template expression which is associated to the uml domain. Matched objects are bound to the variable c so they can be referred. The pattern matches for objects which fulfil the following constraints:

There are two possibilities for the namespace attribute. On the one hand the variable p could be bound to an object of type Package. So all objects are matched whose namespace refers to this object. Otherwise p could be unbound. Those objects are matched whose namespace refers to an arbitrary object of type Package and p is bound to it.
The kind attribute refers to the value ’Persistent’.
Again, the variable cn can be free. It will bind to the value of name of matched objects, no matter what it is. So it can be used in other expressions or relation invocations. If cn is bound to a value, only those objects are matched whose name refers to this value.

The rdbms object template expression is declared similar to the uml one. Two facts must be mentioned:

- It contains a nested object template expression for the column attribute. This shows that QVT Relations supports complex object templates.
- The variable cn, which was bound by the uml object template expression, is referred here. This means the name attribute of the matched Table object must have the same value as the name of the Class object.

A relation declaration can contain the optional **when** and **where** clause. The when clause declares conditions and relations under which a relation needs to hold. For the ClassToTable relation this means the relation PackageToSchema holds for Package p and Schema s. If this wouldn't be the case, ClassToTable wouldn't hold either. The where clause is used to modify variables or to specify relations which must hold when the actual one holds. In line 26 of the example the above declared variable prefix is associated to an empty string and the AttributeToColumn relation is invoked with c, t and prefix. This implies AttributeToColumn declares a primitive domain of type String, one of type Class and another of type Table. The domain variables are bound to the supplied arguments so that further object template expressions are evaluated on these elements only. If ClassToTable holds, AttributeToColumn holds too for the supplied arguments. In conclusion, “when” and “where” clauses can be used to decompose and structure complex relation declarations.
For both, the definition of constraints in “when” clauses and manipulation of variables in “where” clauses, OCL expressions are used. Detailed information about allowed expressions can be found in [12, section 7.13.2]. In the case, OCL expressions become complex and hardly readable, QVT relations supports the definition of OCL queries which can be invoked. In line 31 to 40 a query is defined which transforms a primitive type to an SQL type.

4 Appendix

4.1 Bibliography

[1]		Blunk, A.: “Klassifikation von Modelltrandformationen” Humbold Universität zu Berling, Lehrstuhl Systemanalyse. Februar 2006. http://www2.informatik.hu-berlin.de/~blunk/pdf/se_modelltransformationen.pdf
[2]		Czarnecki K., Helsen S.: „Classification of model transformation approaches“. University of Waterloo, Canada. Workshop on Generative Techniques, 2003. http://www.swen.uwaterloo.ca/~kczarnec/ECE750T7/czarnecki_helsen.pdf
[3]		Gottschau J.: „Velocity“. FH Wedel, University of applied sience, 2002. http://www.fh-wedel.de/~si/seminare/ws02/Ausarbeitung/d.velocity/layout0.htm
[4]		Montero M.: „XSLT – Einführung in die Transformationssprache“ dpunkt.verlag GmbH, 2004 http://www.data2type.de/xml/Einfuehrung_in_die_Transformationssprache.html
[5]		Schmauder R., Schill P.: „Codegenerierung mit dem ’Eclipse Modelling Framework’ und JET“. Sig datacom 2005. http://www.omondo.de/pdfs/schmauder_schill_OS_01_05.pdf
[6]		Wagner, C. „Allgemeine Transformationstechniken“ Universität Padaborn, 2005. http://wwwcs.uni-paderborn.de/cs/kindler/Forschung/ComponentTools/ct-seminar/Seminar-Folien-Modelltransformation.pdf
[7]		Frederic Jouault and Ivan Kurtev, On the Architectural Alignment of ATL and QVT, Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 06). ACM Press, Dijon, France, pages 1188-1195, 2006
[8]		MDA Guide Version 1.0.1, OMG, document number: omg/2003-06-01, June 2003
[9]		EMODE Germany, Deliverable 2.2, July 2006
[10]		EMODE Germany, Deliverable 2.1, February 2006
[11]		Olof Torgersson. A Note on Declarative Programming Paradigms and the Future of Definitional Programming. In Das Winteroete’96, 1996.
[12]		Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification (ptc/05-11-01). http://www.omg.org/, November 2005.
[13]		Object Management Group: MOF 2.0 Query / Views / Transformations RFP, Request for Proposal (ad/2002-04-10). http://www.omg.org/, April 2002.
[14]		Free On-Line Dictionary Of Computing. http://foldoc.org/.
[15]		Klaus Schmid, The Self-Adaptation Problem in Software Specifications, Proceeding of the Workshop on Software Engineering Challenges in Ubiquitous Computing, Lancaster, UK, 2006
[16]		Clerckx, T.; Van den Bergh, J. & Coninx, K., Modeling Multi-Level

		Context Influence on the User Interface, Fourth IEEE Conference on Pervasive Computing and Communications WORKSHOPS, IEEE Computer Society, pages 57-61, 2006
[17]		Myers, B.; Hudson, S.E. & Pausch, R. , Past, present, and future of user interface software tools , ACM Trans. Comput.-Hum. Interact., ACM Press, pages 3-28, 2000