Consistency and Transformation Rules in the MDA-based Modeling of an Enterprise Software Architecture

Tejas Doshi

Masters Thesis

submitted in partial fulfillment of the requirements for the degree
Masters of science in Information and Media Technologies

supervised by

Prof. Ralf Möller
Prof. Dieter Gollmann
Miguel Garcia

Software Technology and Systems **(STS)**
Technical University of Hamburg-Harburg **(TUHH)**

Krisztián Szitás

CORYX Software GmbH

Hamburg, December 2005

Thanks to

My Parents

For helping me start off with a good education,
from which all else springs...

Abstract

'Consistency checking' and 'Transformation of models' are some of important links in MDA process, but because of immaturity of technology and lack of interest many MDA tools had overlooked them. Extending the existing MDA based tools for making them more mature and advance in MDA sense is the target of this thesis. Prototype of Extended Coryx Platform Technology (CxPT) and Extended Octopus (an open source MDA based tool) has been described in this thesis followed by the study of different techniques and tools in this area.

Declaration

I declare that:

this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, 4th December 2005.

(Tejas Doshi)

Content

At a Glance

Table of Contents

List of Figures

Preface

Enterprise software systems are often complex systems. They are not complex only because they are distributed but also because they are systems that evolve quickly in time, implemented by different technologies, and possibility of integration with the legacy systems. Many technologies such as Electronic data interchange (EDI), Transaction processing monitors, Distributed components (e.g., .NET/Enterprise Java Bean components) and recently webservices too, have supported such complex systems development. Technology never last long, but business logic does. Protecting the investment done for the software development from obsolescence is very important goal. On the other side enterprise software systems are and will continue to be developed using multiple technologies, so integration and harmonization between different technologies for the development of such systems is another important goal.

Few years back, Model Driven Architecture (MDA) has been proposed to support such large and complex software system development. MDA proposes an architecture where software systems can evolve and different technologies can be integrated and harmonized. To get the full out of Model driven development we need to take care of some issues like checking constraints on models, mapping definition between related metamodels and defining transformation rules.

In the rest of the work these issues will be discussed with the review of available tools which supports to solve the same, followed by the real implementation in the MDA based enterprise software system CxPT from CORYX Software GmbH. We will also see the extension of an open source MDA based tool called Octopus.

Acknowledgments

I, hereby take an opportunity to express my gratitude towards courtesy extended to me by assigning the masters thesis, Consistency and Transformation Rules in the MDA-based Modeling of An Enterprise Software Architecture.

Prof. Ralf Möller and Prof. Dieter Gollmann, Technical University of Hamburg-Harburg and Herr Uwe Schenk, CORYX Software GmbH, for believing in me and keeping faith in my work, for giving me an opportunity to work on the project under their supervision.

Miguel Garcia, for taking his precious time out of his busy schedule to discuss the real-time problems, and also for explaining me the insight of the existing and ongoing work of the project throughout this project development, for giving me a freedom to introduce my ideas and apply them, for discussing problems and exploring the way out, for guiding me throughout the development period towards the goal.

Krisztiàn Szitàs, who gave me support to persevere through what seemed like a totally overwhelming and never-ending task, for giving me an inspiration and courage to work in the project. I couldn't have done it without him.

Aravind Alagia Nambi, Stefan Huth and Gregor Härty for giving me support throughout the project in parallel design, development, and testing.

I also appreciate all the people of this project; I have left out the names of some of the people who helped me. For that I am sorry but nonetheless grateful for the many hearts and hands that made this project possible. Thank you for them, for their vision, their caring, their commitment and their actions.

And at last, I would like to thank my beloved problems which occurred on site for directing me to the goal of the complete solution.

Tejas Doshi
tejas.doshi@tuhh.de

1 MDA Basics

1.1 Introduction

Technology never last long, but business logic does. Causes of change in technology are: Programming languages, middleware, operating system, hardware platforms, networks and more. "You must preserve your software investment as the infrastructure landscape changes around it" [1].

Model-Driven Development (MDD) attempts to solve the above mentioned problem. MDD raises the level of abstraction by which software and systems engineers carry out their tasks. This is done by emphasizing the use of models - i.e., abstractions - of the artifacts that are developed during the engineering process. Models are representations of phenomena of interest, and in general are usually easier to modify, update, and manipulate than the artifact or artifacts that are being represented. Models are expressed using a suitable modeling language; one of them is UML, which is a widely used.

MDD is not a development method or process; it can be implemented in a number of ways. The key element in MDD is the construction and transformation of models that are fit for the purposes of the project development. The languages and processes used in construction and transformation will vary from project to project.

The Model-Driven Architecture (MDA) is an initiative of the Object Modeling Group (OMG), aimed at providing a standard approach for MDD. While MDD does not prescribe the use of specific process or sequence of steps to follow, MDA requires the use of a standard meta-steps that should be followed in the development of models and systems.

1.1.1 Platform Independent Models (PIMs) and Platform Specific Models (PSMs)

Key to MDA is the importance of models in the software development process. Within MDA, the software development process is driven by the activity of modeling your software system. The MDA process is divided into three steps:

1. Build a model with a high level of abstraction, which is independent of any implementation technology. This is called a Platform Independent Model (PIM).

2. Transform the PIM into one or more models that are tailored to specify your system in terms of the implementation constructs that are available in one specific implementation technology; e.g., a database model or an EJB

(Enterprise Java Beans) model. These models are called Platform Specific Models (PSMs).

3. Transform the PSMs to code.

Because a PSM generally fits its target technology very closely, the PSMs to code transformation is straightforward. The complex step is the one in which a PIM is transformed to a PSM. The relationships between PIM, PSM, source code, and the transformations between them, are depicted in following figure 1.1.

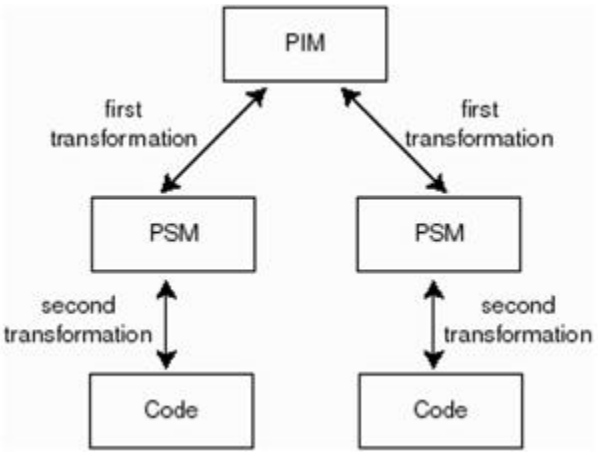

Figure 1.1 The relationship between PIM, PSM, and code

1.1.2 Automation of Transformations

Another key element of MDA is that the transformations are executed by tools. Many tools have been able to transform a platform-specific model to code; there is nothing new about that. What's new is that the transformation from PIM to PSM is automated as well. This is where the obvious benefits of MDA lie. In the field of software development developer spends a lot of time on tasks that are more or less routine. For example, building a database model from an object-oriented design, or building a COM (Common Object Model) component model or an EJB component model from another high-level design. The MDA goal is to automate the cumbersome and laborious part of software development.

1.2 MDA Building Blocks

The MDA framework consists of a number of highly related parts. To understand the framework, you must understand both the individual parts and their mutual relationships. Therefore, let's take a closer look at each of the parts of the MDA framework: the models, the modeling languages, the transformation tools, and the transformation definitions, which are depicted in following figure 1.2.

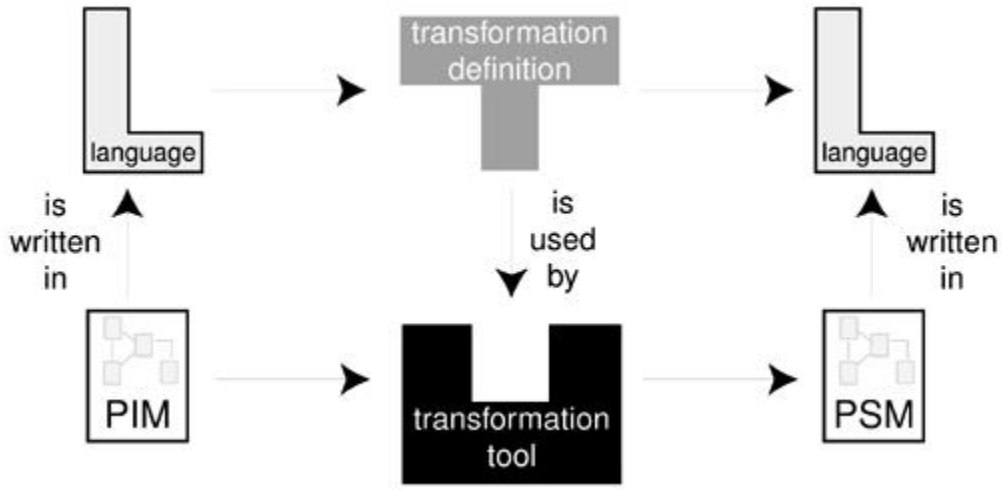

Figure 1.2 The MDA Framework

1.2.1 Models

The first and foremost element of MDA is formed by models—high-level models (PIMs) and low-level models (PSMs). The whole idea of MDA is that a PIM can be transformed into more than one PSM, each suited for different target technologies. If the PIM were to reflect design decisions made with only one of the target technologies in mind, it could not be transformed into a PSM based on a different target technology; the PIMs must truly be independent of any implementation technology.

A PSM, conversely, must closely reflect the concepts and constructs used in the corresponding technology. In a PSM targeted at databases, for instance, the table, column, and foreign key concepts should be clearly recognizable. The close relationship between the PSM and its technology ensures that the transformation to code will be efficient and effective.

All models, both PSM and PIM, should be consistent and precise, and contain as much information as possible about the system. This is where Object Constraint Language (OCL) [51] can be helpful, because models alone do not typically provide enough information.

1.2.2 Modeling Languages

Modeling languages form another element of the MDA framework. Because both PIMs and PSMs are transformed automatically, they should be written in a standard, well-defined modeling language that can be processed by automated tools. Before a system is built, only humans know what it must do. Therefore, PIMs must be written to be understood and corrected by other humans. This places high demands on the modeling language used for PIMs. It must be understood by both humans and machines.

The PSMs, however, will be generated, and the PSM needs to be understood only by automated tools and by experts in that specific technology. The demands on the languages used for specifying PSMs are relatively lower than those on the language for PIMs. Today, there are a number of so-called profiles for UML, define UML-like languages for specific technologies, e.g., the EJB profile. Other modeling languages includes xml based modeling languages, e.g., Eclipse Modeling Framework (EMF) .

1.2.3 Transformation Tools

Transformation tools implement the central part of the MDA approach, thus automating a substantial portion of the software development process. Many tools implement the PSM-to-code transformation. Today, only a few implement the execution of the transformation definitions from PIM to PSM. Most of the PIM-to-PSM tools are combined with a PSM-to-code component.

1.2.4 Transformation Definitions

Another vital part of the MDA framework is formed by the definitions of how a PIM is to be transformed to a specific PSM, and how a PSM is to be transformed into code. Transformation definitions are separated from the tools that will execute them, in order to re-use them, even with different tools. It is not worthwhile to build a transformation definition for one-time use. It is far more effective when a transformation can be executed repeatedly on any PIM or PSM written in a specific language.

Some of the transformation definitions will be user-defined, that is, written by the developers that work according to the MDA process. Preferably, transformation definitions would be in the public domain, perhaps even standardized, and tunable to the individual needs of its users. Some tool vendors have developed their own transformation definitions, which unfortunately usually cannot be adapted by users because their use is not transparent, but hidden in the functionality of the tools and moreover is not reused with different tools.

1.3 MDA Benefits

This section describes some of the advantages of MDA [2]:

1. **Portability**, increasing application re-use and reducing the cost and complexity of application development and management, now and in the future. MDA brings us portability and platform independence because the PIM is indeed platform independent and can be used to generate several PSMs for different platforms.

2. **Productivity**, by enabling developers, designers, and system administrators to use languages and concepts they are comfortable with, while still supporting seamless communication and integration across the teams. A productivity gain can be achieved by using tools that fully automate the generation of code from a PSM, and even more when the generation of a PSM from a PIM is automated as well.

3. **Cross-platform interoperability**, using rigorous methods to guarantee that standards based on multiple implementation technologies all implement identical business functions. The promise of cross-platform interoperability can be fulfilled by tools that not only generate PSMs, but also the bridges between them, and possibly to other platforms as well.

4. **Easier maintenance and documentation**, as MDA implies that much of the information about the application must be incorporated in the PIM. It also implies that building a PIM takes less effort than writing code.

1.4 MDA, The Silver Bullet ?

When explaining the MDA to software developers, one get a sceptical response: "This can never work. You cannot generate a complete working program from a model. You will always need to adjust the code." Is MDA just promising another silver bullet ?

At OOPSLA 2003, Dave Thomas of Pragmatic Programming fame virtually flayed MDA alive, and even Microsoft has questioned MDA's underpinnings. Why the criticisms? We've seen similar visions fail miserably in the past. Also, many correctly believe that MDA could take IT organizations seriously off track if they don't navigate the software process waters effectively. [3]

2 Problem Definition

2.1 Context

In the enterprise whenever basic internal conditions change must be communicated in the enterprise. Because new standards are defined, new technologies and products come on the market, updating systems accordingly is difficult. Enterprises cannot react always fast, purposefully and above all flexibly to the constantly changing conditions. At the same time it requires to keep cost lower for the same.

The answer of the CORYX software GmbH to these requirements is the CxPT (CORYX Platform Technology). CxPT is defined as a model-driven and generative platform which is based on J2EE for software development, the one row of Architecture patterns for applications of business, by means of Frameworks, Design Patterns, standard components and code generators. Thus the complex technical concepts of the J2EE details shielded for developers and high-quality, scalable J2EE applications produced. That means: With CxPT, systems become more efficient, flexible, platform independent and thus more futurable.

2.1.1 Generation by Templates: (MDA Light?!)

One of the beginnings for the practical conversion of the MDA (we call it times MDA light) is based on the use of Plug-ins. The platform-independent model is jumped over as source code is generated directly from the PSM, which is very specific to J2EE technology. With this MDA light, one can describe many details of the specialized logic in specification instead in source code. Predivide the business logic will make system technologically flexible. The investments into specialized models live longer - much longer than the technology, with which one has implemented these models.

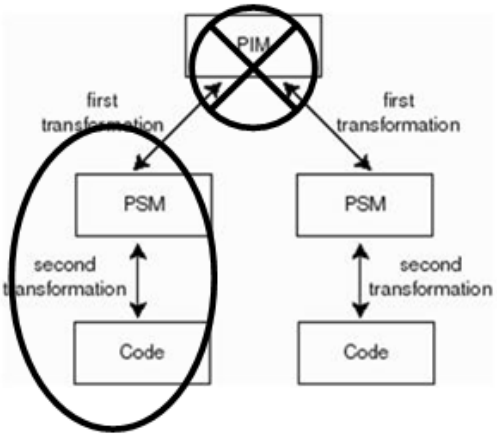

Figure 2.1 MDA Light implementation

The concept of the Model Driven Architecture makes a modeling possible of middleware solutions for the different technology platforms. High degree is given to the reusability, since the same model of the application of business can be used with different implementation technologies. With this proceeding more time can be invested into the modeling of the business processes, without thinking over the details of the implementation on the technology platform. The "service generator" of the CxPT support the beginning of the MDA to generate J2EE based middleware solution from the XML based specification and offer a comfortable possibility to convert that concept of the MDA into the practice. Basic concept of CxPT can be seen in the figure 2.2.

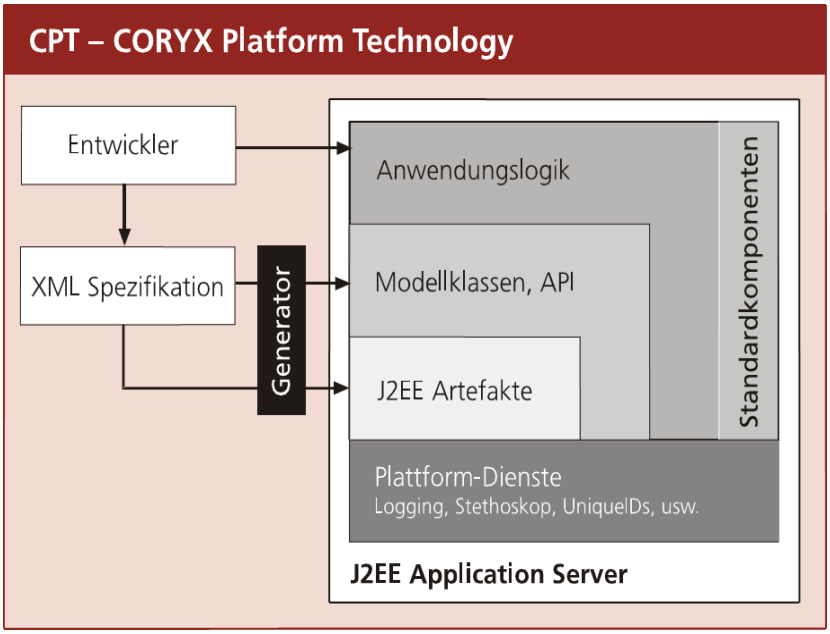

Figure 2.2 Basic concept of the CxPT

2.2 Missing Links

There are some missing links in CxPT as an matured MDA tool. Links where consistency must be checked at the PIM level and then application of transformation rules to transform PIM to PSM, which then can be used to generate J2EE based middleware solution as before in CxPT. Filling this gap will make CxPT more consistent, time effective, more maintenable and stable in development of its solutions.

2.3 Purpose of this Thesis

The purpose of this thesis is to explore areas: consistency checking and transformation of models in MDA, exploring related tools and design a prototype solution for the CxPT. Purpose also includes to solve some of the generic problems in this area like integrate different tools or standards and transformation between them.

3 Consistency in MDA

3.1 Introduction

In MDA based development environment Modeling Languages are used for software design and modeling. The main strength of a modeling language lies in its ability to express many facets of design, ranging from structural (ex: class diagram in UML) to behavioral (ex: interaction and state machine diagrams in UML), using a single integrated formalism. This language comes with a set of syntactic and semantic rules that derives the understanding and interpretation of models written in this language. Some of the rules that are deemed necessary are formally expressed in the specification to assert the well-formedness of models. Others, mainly semantic rules, are informally defined in the specification to give more flexibility and expressive power to designs at different levels of abstraction, by different modeling methodologies or for different application domains. Such syntactic and semantic rules are needed to keep the model consistent as per the specification for which it is targeted.

The motivation for model consistency are:

Correctness : Usually, consistency problems reveal design problems or misuse of modeling language. When those problems are discovered early in the design process, it is easier and more **cost effective** to fix than if they were discovered at a later stage.

implementability, which usually involves translating a platform independent model into a platform specific model (e.g. Programming language), a usually precise and unambiguous notation.

Without consistency analysis, it would be hard to evolve the model and ensure that the collaborative effort is coherent. The notion of consistency, or lack of, has its roots in formal methods. To assert that something is consistent, you have to declare what it is consistent with. Any language has its own unique syntax and semantics. The syntactic correctness or well-formedness of a model is usually a prerequisite to any further consistency analysis. Syntax is what makes the model readable and hence verifiable.

Failing to maintain the well-formedness of a model often leads to ambiguity. Another source for ambiguity is the existence of incomplete semantics. Usually a model goes through a series of refinement transformations before finally getting translated into code, an inherently formal language. While syntax helps readability, semantics is what gives meaning to language constructs. While consistency at the semantic level is generally a desired property to ensure the integrity of a model, it is mostly needed when transforming a model into a formal notation.

Consistency is either intra-model (also called horizontal), which is a property of a model asserting its syntactic and semantic conformance, or inter-model which is between different models related together by one or more transformation

relationships. In general, most of the work in this area tends to focus on ensuring that these transformations are consistency-preserving. Another consistency classification distinguished between static and dynamic constraints. A static constraint is one that can be verified statically without running the model, while a dynamic constraint cannot be verified until runtime.

Defining inconsistency is one task; detecting it is another. Consistency assertion or inconsistency detection falls within three main categories. The first category tries to complete the meta-model to allow for easier accessibility from a model element to all its associated elements. The second category tries to enhance the language used for expressing constraints on the meta-model. The Object Constraint Language (OCL) is the native language for expressing constraints in UML.

The process of translating a model into a formal notation involves first clearing the ambiguity by agreeing on the interpretation of the expressions according to some semantic domain.

The constraint checking framework has to be:

Flexible, to allow for different configurations, and

extensible, to allow for new constraints to be added

It should also **support a batch checking mode** in addition to an **incremental on-demand mode**.

The framework has to be **user-friendly** in its presentation and allow for easy expression of constraints.

Most importantly, the framework has to be **efficient**, which means among other things **scalable** to the size of the model, **fast** in completing the analysis, and **accurate** in reporting results.

It is also desirable if the framework can offer consistency correction actions and design **assist tips**.

3.2 Consistency Classification

The following table summarizes different consistency classifications [2]:

Classification	**Features**
Syntactic vs. Semantic	Consistency rules that can be expressed by a formal language are syntactic, otherwise they are semantic
Static vs. Dynamic	Consistency rules that can be verified without executing a model are static, otherwise they are dynamic
Intra-Model vs. Inter-Model	Consistency rules within the same model are intra-model. Those that span models are inter-model

Table 3.1 Consistency Classifications

3.2.1 Syntactic vs. Semantic Consistency

The language specifications introduce two initial levels of consistency, the meta-model and the well-formedness rules. The meta-model is a schema that precisely defines the constructs and rules needed for creating models. A model is inconsistent if it does not conform to the meta-model. However, it may not be consistent even if it conforms to the meta-model. This is mainly due to the limited expressiveness of the meta-language.

In an effort to complement the meta-language, the OMG proposed OCL, a higher order logic language for denoting well-formedness constraints in a modeling language like UML. OCL, a pure expression language, has constructs to inspect and navigate objects and their structure and return a true or false value but it does not change the model. Well-formedness, as expressed by OCL constraints, is usually a prerequisite to any further consistency analysis. The following are some examples of well-formedness rules as expressed in OCL:

An element may not directly or indirectly own itself:

```
not self.allOwnedElements()->includes(self)
```

Elements that must be owned must have an owner:

```
self.mustBeOwned() implies owner->notEmpty()
```

Once the semantics are formalized, further consistency analysis can proceed by tailoring rules to the selected interpretations.

3.2.2 Static vs. Dynamic Consistency

Most languages distinguish between their static and dynamic semantics. The static semantics, or the syntax, of a modeling language is formally described in terms of its meta-model and OCL constraints. While syntax can usually be checked by a static inspection of a model, dynamic semantics cannot be completely verified until runtime. For example, it may not be possible to statically check that a precondition to an operation is satisfied before the operation is called. The problem here is that not all modeling languages are an executable language, and therefore the dynamic constraints have to be embedded into an executable formalism (like code) that is translatable from a modeling language.

3.2.3 Intra-Model vs. Inter-Model Consistency

One recurring classification of consistency in the literature distinguishes between intra- model and inter-model consistency or between horizontal and vertical consistency. Intra- model or horizontal consistency is a property of a model. It indicates that all the elements of a model are syntactically and semantically correct.

Multiple viewpoints for modeling the same system. They range from structural (ex: class and instance diagrams in UML) to behavioral (ex: interaction and state machine diagrams in UML). These viewpoints or perspectives are usually inter-dependent. For

example, a synchronous message in a sequence diagram has to match an operation in a class diagram. This intersection of viewpoints leads to a very interesting class of inconsistencies that is not formalized as part of the specifications. While some of these constraints are obvious, others are mainly heuristic in nature. The following is a sample list of constraints between the class and the object diagrams:

The number of values for an attribute of a given object violates the multiplicity lower bound for that attribute as defined by the object's classifier.

```
self.value ->size() >= self.definingFeature.lowerBound()
```

The attribute's value in an object does not conform to the corresponding attribute type as defined by the object's classifier.

```
if self.definingFeature.type ->isEmpty()
     then true
else
     self.value ->forAll( v:ValueSpecification |
      v.type.conformsTo(self.definingFeature.type))
endif
```

The object is not classified (heuristic since it is allowed by the specifications)

```
self.classifier.size() > 0
```

On the other hand, inter-model consistency is a relationship between models. These models are usually related to each other by some sort of a transformation relationship. A transformation describes the application of some procedures to one model to create a new model. The new model can be another representation of the same information in the old model, or a modified version of the original model. In fact, the relationship between transformation and consistency characterized a transformation as consistent if a model before the transformation is consistent with the new model after the transformation.

Refinement is a transformation that takes a model from an abstract level to a more concrete or detailed level. One essential characteristic of this kind of transformation is consistency preservation. Such consistency is also called a vertical consistency since it is between models at different levels of abstraction. A refinement transformation left the old and the new models consistent between each other.

3.3 Introduction to OCL

The Object Constraint Language (OCL) is a language that enables one to describe expressions and constraints on object-oriented models and other object modeling artifacts. An *expression* is an indication or specification of a value. A *constraint* is a restriction on one or more values of (part of) an object-oriented model or system.

The OCL is a standard query language, which is part of the Unified Modeling Language (UML) set by the Object Management Group (OMG) as a standard for object-oriented analysis and design.

3.3.1 Types of expressions

Expressions can be used in a number of places:

- To specify the initial value of an attribute or association end.
- To specify the derivation rule for an attribute or association end.
- To specify the body of an operation.
- To indicate an instance in a dynamic diagram.
- To indicate a condition in a dynamic diagram.
- To indicate actual parameter values in a dynamic diagram.

3.3.2 Types of constraints

There are four types of constraints:

- An *invariant* is a constraint that states a condition that must always be met by all instances of the class, type, or interface. An invariant is described using an expression that evaluates to true if the invariant is met. Invariants must be true all the time.
- A *precondition* to an operation is a restriction that must be true at the moment that the operation is going to be executed. The obligations are specified by postconditions.
- A *postcondition* to an operation is a restriction that must be true at the moment that the operation has just ended its execution.
- A *guard* is a constraint that must be true before a state transition fires.

3.3.3 The Context of an OCL expression

The *context definition* of an OCL expression specifies the model entity for which the OCL expression is defined. Usually this is a class, interface, datatype, or component. In terms of the UML standard, this is called a *Classifier*.

Sometimes the model entity is an operation or attribute, and rarely it is an instance. It is always a specific element of the model, usually defined in a UML diagram. This element is called the *context* of the expression.

Next to the context, it is important to know the *contextual type* of an expression. The contextual type is the type of the context, or of its container. It is important because OCL expressions are evaluated for a single object, which is always an instance of the contextual type. To distinguish between the context and the instance for which the expression is evaluated, the later is called the *contextual instance*. Sometimes it is necessary to refer explicitly to the contextual instance. The keyword *self* is used for this purpose.

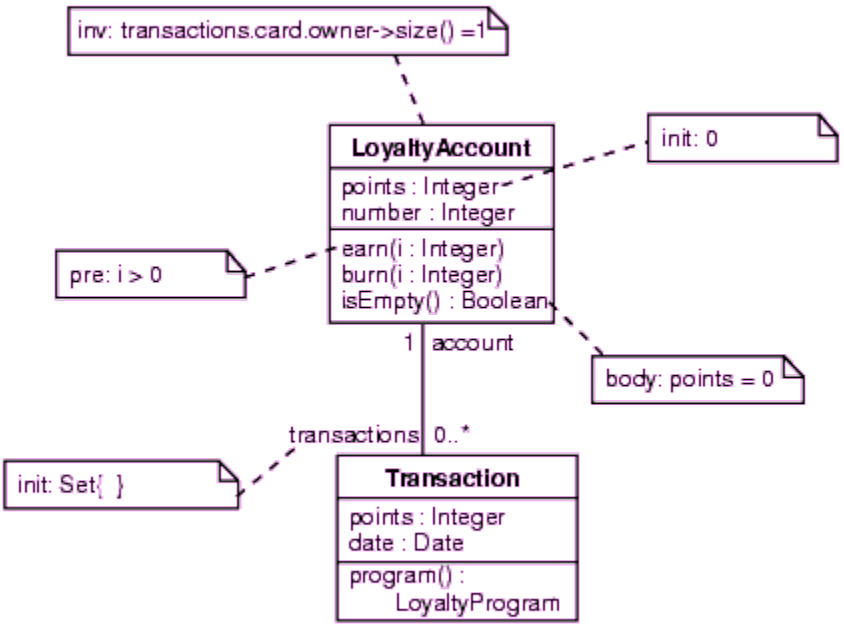

Figure 3.1 Class diagram with OCL constraints for LoyaltyAccount

For example, the contextual type for all expressions in above figure is the class *LoyaltyAccount*. The precondition (*pre: i>0*) has as context the operation *earn.* When it is evaluated, the contextual instance is the instance of *LoyaltyAccount* for which the operation has been called. The initial value (*init: 0*) has as context the attribute *points*. The contextual instance will be the instance of *LoyaltyAccount* that is newly created.

3.3.4 Invariants on attributes

The simplest constraint is an invariant on an attribute. Suppose our model contains a class *Customer* with an attribute *age*, then the following constraint restricts the value of the attribute:

```
context Customer inv:
age >= 18
```

3.3.5 Invariants on associations

One may also put constraints on associated objects. Suppose in our model contains the class *Customer* has an association to class *Salesperson*, with role name *salesrep* and multiplicity 1, then the following constraint restricts the value of the attribute *knowledgelevel* of the associated instance of Salesperson:

```
context Customer inv:
salesrep.knowledgelevel >= 5
```

3.3.6 Collections of objects

In most of the cases the multiplicity of an association is not 1, but more than 1. Evaluating a constraint in these cases will result in a collection of instances of the associated class. Constraints can be put on either the collection itself, e.g. limiting the size, or on the elements of the collection. Suppose in our model the association between Salesperson and Customer has role name *clients* and multiplicity 1..* on the side of the Customer class, then we might restrict this relationship by the following constraint.

```
context Salesperson inv:
clients->size() <= 100 and clients->forAll(c: Customer | c.age >= 40)
```

3.3.7 Pre- and postconditions

In pre- and postconditions the parameters of the operation may be used. Furthermore, there is a special keyword *result* which denotes the return value of the operation. It can be used in the postcondition only. As an example we have added an operation *sell* to the Salesperson class.

```
context Salesperson::sell( item: Thing ): Real
pre: self.sellableItems->includes( item )
post: not self.sellableItems->includes( item ) and result = item.price
```

3.3.8 Derivation rules

Models often define derived attributes and associations. A derived element does not stand alone. The value of a derived element must always be determined from other (base) values in the model. Omitting the way to derive the element value results in an incomplete model. Using OCL, the derivation can be expressed in a derivation rule. In the following example, the value of a derived element *usedServices* is defined to be all services that have generated transactions on the account:

```
context LoyaltyAccount::usedServices : Set(Services)
derive: transactions.service->asSet()
```

3.3.9 Initial values

In the model information, the initial value of an attribute or association role can be specified by an OCL expression. In the following examples, the initial value for the attribute *points* is *0*, and for the association end *transactions,* it is an empty set:

```
context LoyaltyAccount::points : Integer
init: 0

context LoyaltyAccount::transactions : Set(Transaction)
init: Set{}
```

Note the difference between an initial value and a derivation rule. A derivation rule states an invariant: The derived element should always have the same value that the rule expresses. An initial value, however, must hold only at the moment when the contextual instance is created. After that moment, the attribute may have a different value at any point in time.

3.3.10 Body of query operations

The class diagram can introduce a number of query operations. Query operations are operations that have no side effects, i.e., do not change the state of any instance in the system. Execution of a query operation results in a value or set of values, without any alterations in the state of the system. Query operations can be introduced in the class diagram, but can only be fully defined by specifying the result of the operation. Using OCL, the result can be given in a single expression, called a *body expression*. In fact, OCL is a full query language, comparable to SQL. The use of body expressions is an illustration thereof.

The next example states that the operation *getCustomerName* will always result in the name of the card owner associated with the loyalty account:

```
context LoyaltyAccount::getCustomerName() : String
body: Membership.card.owner.name
```

3.3.11 Broken constraints

Note that evaluating a constraint does not change any values in the system. A constraint states "this should be so". If for a certain object the constraint is not true, in other words, it is broken, then the only thing we can conclude is that the object is not correct, it does not conform to our specification. Whether this is a fatal error or a minor mistake, and what should be done to correct the situation is not expressed in the OCL.

3.4 Requirements for OCL tools

3.4.1 Functional requirements

OCL is not a stand-alone language. The language was conceived to express information that the graphical formalism of UML cannot. By using the graphical and textual formalisms jointly, the UML models gain precision. OCL cannot be used apart from UML; consequently, the OCL tools must support both OCL and UML formalisms.

The OCL support covers both the user model and the metamodel level. OCL offers a more intuitive access at the metamodel level. To define OCL specifications at M2 level of OMGs 4 layers, the user needs to understand the metamodel, to navigate it. The specifications made at the user model level express the "business semantic" of each model.

The OCL functionalities concern the support for: specifying, compiling, evaluating, debugging and reusing OCL specifications.

The UML models need to be validated against Well Formedness Rules (WFR) and their Business Rules. Until now, this activity was not performed in modeling, exclusively due to the lack of appropriate OCL support in UML tools.

Regarding the evaluation process, there is a particular requirement induced by the dynamic characteristic of the user model. At runtime, the objects can change their state and the application consistency must be kept through dynamic evaluation of the OCL constraints that specify model semantics. This can be realized if the OCL specifications are translated into source code, corresponding to a target programming language, integrated with source code generated from the UML and evaluated everywhere and every time this is necessary. Thus, an UML-OCL tool needs to support automatic code generation from OCL specifications and the integration of this code.

In addition, each tool supporting rule evaluation must provide the user with functionalities needed to identify the eventual rule failures and to modify the model in order to comply with evaluated rules.

The OCL specifications made at metamodel level are reusable because these specifications can be used irrespective of the evaluated UML model and the tool used in the evaluation process.

Also, taking into account the object-oriented nature of UML and of most of the user models, UML CASE tools must support design by contract. This means that each invariant, pre or post condition, specified in a parent or for a parent operation, must be satisfied in all its descendants. Another important functional requirement concerns the possibility to redefine in descendants the functions specified by means of the OCL def – let mechanism.

3.4.2 GUI requirements

The GUI component must support the user in interacting with the tool in the simplest, natural and intuitive manner. Mainly, the GUI must provide support for: model management, model navigation, synchronization of information displayed in different views. Considering the functionalities needed for managing medium and large size models, the tools must support information filtering. Extended searching functionalities for the model information are very useful. The text editor needs to support: auto indentation, syntax highlighting, auto completion.

3.5 Related Work

3.5.1 Dresden OCL ToolKit for OCL2.0

http://dresden-ocl.sourceforge.net/

Instead, many of the contained tools are meant to be used as a library, integrated into other tools, but there exist also some standalone tools in the toolkit. The whole toolkit is metamodel-based and relies on a common metamodel derived from the MOF14 and UML15 metamodels.

All models and metamodels are stored in a MDR metadata repository and then Java interfaces for accessing models can be generated using JMI based API. The contained OCL parser uses two passes. Pass one creates a concrete syntax tree (CST) from the textual constraint. During pass two, the attribute evaluator performs the transformation from CST to Abstract Syntax Tree (AST).

It allows loading MOF14 based models, creating constraints over these models, generating code for them and evaluating the constraints in the context of a concrete model (M1). It also allows loading or editing textual OCL constraints, loading UML15 models through XMI import and parsing the constraints in the context of the model loaded. After parsing, the constraints will be attached to their contextual model elements and can be exported using XMI export.

An example of simple constraint that manager in the company is also an employee can be specified by

```
@invariant manager_is_employee2:
  manager.employers->includes(self)
```

Its freely available with source code to download.

3.5.2 USE – UML based Specification Environment

http://www.db.informatik.uni-bremen.de/projects/USE/

USE is a system for the specification of information systems. It is based on a subset of the Unified Modeling Language (UML). A USE specification contains a textual description of a model using features found in UML class diagrams (classes,

associations, etc.). Expressions written in the Object Constraint Language (OCL) are used to specify additional integrity constraints on the model. A model can be animated to validate the specification against non-formal requirements. System states (snapshots of a running system) can be created and manipulated during an animation. For each snapshot the OCL constraints are automatically checked. Information about a system state is given by graphical views. OCL expressions can be entered and evaluated to query detailed information about a system state. The figure 3.2 below gives a general view of the USE approach.

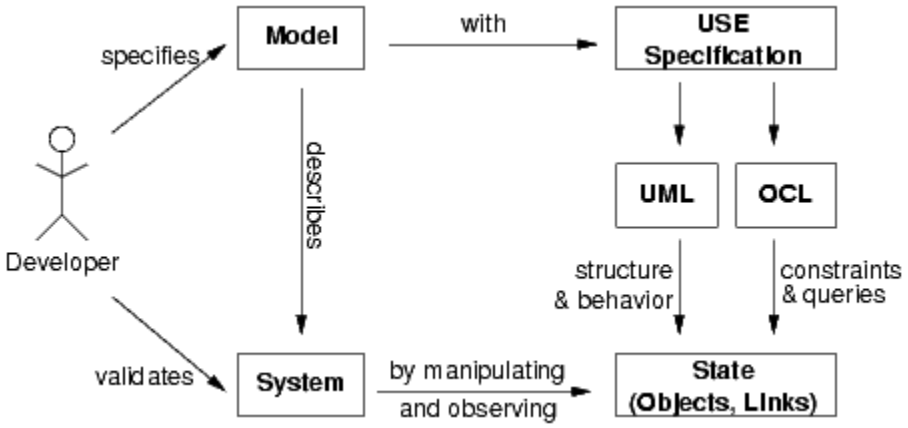

Figure 3.2 General view of the USE approach

The distribution of USE comes with full sources.

Example of partial textual specification of Employee in a company working on different projects would be specified by

```
model Company

-- classes

class Employee
attributes
  name : String
  salary : Integer
end
```

and example of one OCL constraint on that model can be specified by

```
context Employee
   -- employees get a higher salary when they work on
   -- more projects
  inv MoreProjectsHigherSalary:
    Employee.allInstances->forAll(e1, e2 |
      e1.project->size > e2.project->size
        implies e1.salary > e2.salary)
```

3.5.3 OCLE 2.0 – Object Constraint Language Environment

http://lci.cs.ubbcluj.ro/ocle/

OCLE is a UML CASE Tool (developed in "BABES-BOLYAI" University Computer Science Research Laboratory), offering full OCL support both at the UML metamodel and model level. It checks the well formedness of UML models against the WFR specified in UML 1.5. OCLE offers a very strong support for compiling and debugging OCL specifications. UML models saved in XMI 1.0 or 1.1, regardless of the tools and parsers used in producing and transferring the models can be used. Apart from the OCL support offered at the metamodel level, OCLE helps users in realizing both static and dynamic checking at the user model level. Dynamic support is offered by means of the generated Java source code. the graphical interface was conceived and implemented with the aim of supporting the use of OCLE in a natural and intuitive manner. A User Manual and some detailed examples are included in the distribution package.

Certainly, a lot of possible extensions can be realized and probably enough functions can be improved like to offer a better and complete support for the diagrams editor, better code generation, a strong support for transforming different models (MDA), and so on. As source code is not available its difficult to integrate into existing framework.

3.5.4 Octopus

http://www.klasse.nl/english/research/octopus-intro.html

The **OC**L **T**ool for **P**recise **U**ML **S**pecifications (Octopus) of the Netherlands company Klasse Objecten serves the syntactic examination of in the OCL formulated expressions. The tool is available as Plug-in for the Eclipse development environment. Octopus supports the OCL in version 2.0 and is in further development.

Octopus is able to statically check OCL expressions. It is able to transform the UML model, including the OCL expressions, into Java code. UML model here is the octopus' own textual syntactic format which will also be refered as OctopusUML in this document for clarification. Octopus has xmiimport eclipse plug-in which can import UML model from xmi into OctopusUML. It also has internal visitors for OctopusUML model, which can be used in case of export or transform purpose in some special requirements.

Octopus fully conforms to version 2.0 of the OCL standard. Furthermore, Octopus offers the possibility to view expressions in an SQL-like syntax.

Octopus is able to generate a complete 3-tier prototype application from your UML/OCL model.

The middle tier consists of plain old Java objects (POJOs). These POJOs include code for checking invariants and multiplicities from the model. OCL expressions that define the body of an operation are transformed into the body of the corresponding Java method. Derivation rules and initial value specifications are transformed as expected. Optionally a number of convenience methods may be created for each class, for example, a *getCopy()* method. The creation of a visitor interface and the corresponding *accept* methods is also optional.

The storage tier consists of an XML reader and writer dedicated to the given UML/OCL model. It stores any data content in your prototype application in an XML file. It is also able to read the content of this XML file into prototype application. Furthermore, application can be regenerated, for instance, because one of the classes was missing an attribute, and the reader will still be able to read the XML file. The reader will read the contents of the XML file and produce objects for whatever classes, attributes, and association ends are still in your model.

The user interface tier consists of an implementation of a plug-in for the Eclipse Rich Client Platform, where model can be created, navigated and OCL invariants and multiplicities can be checked.

As a proof of concept for the MDA vision Octopus generates a complete prototype application from UML/OCL model but still it needs even better transformation tools. As source code is available it can be extended and integrated easily with the existing frameworks.

3.5.5 MMT

The **M**eta-**modeling T**ool (MMT) has been designed and constructed to support the testing of UML 2.0. MMT is a virtual machine that understands the textual version of constrained class diagrams and the construction of languages using templates. MMT supports the testing of language definitions at a number of levels. At a simple level it is able to check the definition to ensure it is syntactically correct (the importance of this in a definition the size of UML 2 should not be underestimated). MMT is also able to check that constraints hold within the definition to ensure that models are well formed. Most importantly, MMT is reflexive which enables the building of new languages described using existing languages. Consequently, MMT can be used to build UML 2 models and check that our understanding of UML2 (encapsulated in the definition) is correctly defined. A programmer's guide is available at following URL.
http://www.dcs.kcl.ac.uk/staff/tony/docs/ProgrammersGuideToMMT.pdf

4 Transformation in MDA

4.1 Introduction

Transformation can be defined as the generation of a target model from a source model. This means that transformations are purely processes. The process is described by a transformation definition, which consists of a number of transformation rules, and is executed by a transformation tool. In an MDA approach desirable features of the transformation process are [2]:

1. **Tunability**, which means that although the general rule has been defined in the transformation definition, an application of that rule can be configured and tuned with additional control parameter; for example, when transforming a UML String to a database String (VARCHAR) in an entity-relationship model, you might want the length of the VARCHAR to differ for each occurrence of a UML String.

2. **Traceability**, Transformations may record links between their source and target elements. These links can be useful in performing impact analysis (i.e., analyzing how changing one model would affect other related models), synchronization between models, model-based debugging (i.e., mapping the stepwise execution of an implementation back to its high-level model), and determining the target of a transformation.. A preferable approach is to store a GUID in each model element and store the traceability information separate from the source and target.

3. **Incremental consistency**, which means that when target-specific information has been added to the target model and it is regenerated, the extra information persists and newly added information will get merged.

4. **Bidirectionality**, Most rules are applied in one direction by binding the LHS in the source and expanding the RHS in the target model. In some cases, a declarative rule (i.e., one that only uses declarative logic and/or patterns) can be applied in the inverse direction, too. This property seems attractive in the context of synchronization between models. An alternative approach is to define two separate rules, one for each direction.

4.1.1 Metamodeling

Model is a description of (part of) a system written in well-defined language. A well-defined language is a language which is suitable for automated interpretation by a computer.

In the past, languages were often defined using a grammar in Backus Naur Form (BNF), which describes what series of tokens is a correct expression in a language. This method is suitable and heavily used for text-based languages. But modeling languages do not have to be text based, and often aren't (they can, for example,

have a graphical syntax, like UML), we will need a different mechanism for defining languages in the MDA context. This mechanism is called metamodeling.

A model defines what elements can exist in a system. A language also defines what elements can exist. It defines the elements that can be used in a model. So, we can describe a language by a model: the model of the language describes the elements that can be used in the language.

Every kind of element that a modeler can use in his or her model is defined by the metamodel of the language the modeler uses. In UML you can use classes, attributes, associations, states, actions, and so on, because in the metamodel of UML there are elements that define what is a class, attribute, association, and so on. If the metaclass Interface was not included in the UML metamodel, a modeler could not define an interface in a UML model.

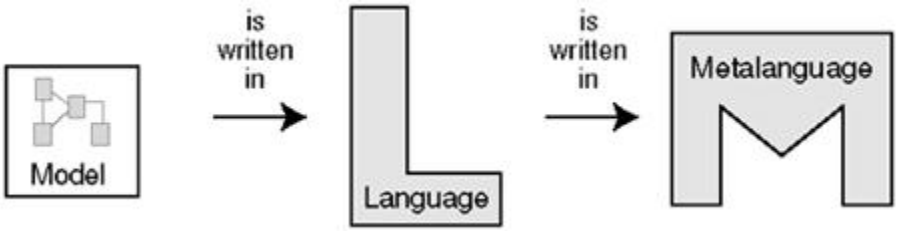

Figure 4.1 Relation between model, modeling language and metalanguage

Because a metamodel is also a model, a metamodel itself must be written in a well-defined language. This language is called a metalanguage.

Because a metalanguage is a language itself, it can be defined by a metamodel written in another metalanguage. In theory there is an infinite number of layers of model–language–metalanguage relationships. The standards defined by the OMG use four layers, as explained in the next section.

4.1.2 The Four Modeling Layers of the OMG

The OMG uses a four-layered architecture for its standards. In OMG terminology these layers are called M0, M1, M2, and M3 [4].

Layer M0: The Instances

At the M0 layer there is the running system in which the actual ("real") instances exist. These instances may exist in various incarnations, such as data in a database, or as an active object running in a computer.

While modeling a business and not software, the instances at the M0 layer are the items in the business itself, for example, the actual people, the invoices, and the products. While modeling software, the instances are the software representations of the real world items, for example, the computerized version of the invoices or the orders, the product information, and the personnel data.

Layer M1: The Model of the System

The M1 layer contains models, for example, a UML model of a software system. In the M1 model, for instance, the concept Customer is defined, with the properties name, street, and city.

There is a definite relationship between the M0 and M1 layers. The concepts at the M1 layer are all categorizations or classifications of instances at the M0 layer. Likewise, each element at the M0 layer is always an instance of an element at the M1 layer. Instances that do not adhere to their specification at the M1 layer are not feasible.

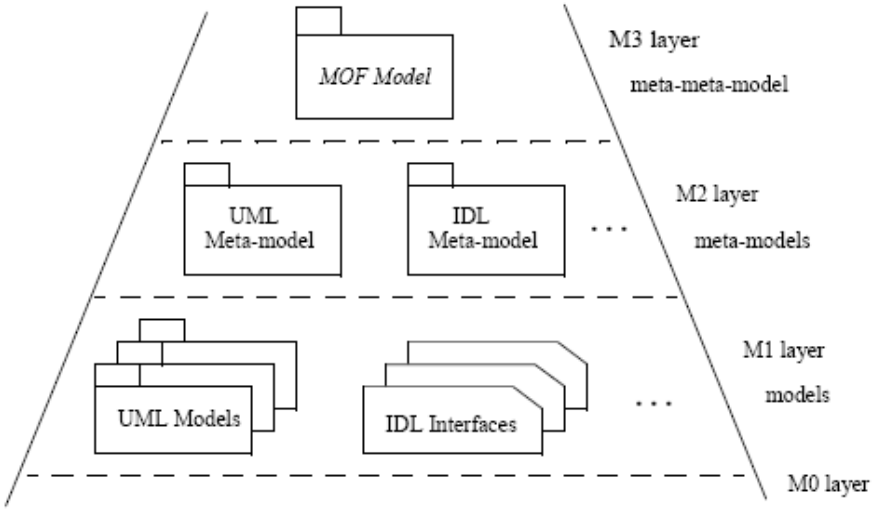

Figure 4.2 The four modeling layers of the OMG

Layer M2: The Model of the Model

The elements that exist at the M1 layer (classes, attributes, and other model elements) are themselves instances of classes at M2, the next higher layer. An element at the M2 layer specifies the elements at the M1 layer. The same relationship that is present between elements of layers M0 and M1, exists between elements of M1 and M2.

The model that resides at the M2 layer is call*ed* a metamodel. In fact, the concepts used at the M1 and M2 layers are identical. An M1 class defines instances at the M0 layer, an M2 class defines instances at the M1 layer.

Layer M3: The Model of M2

Along the same line, we can view an element at the M2 layer being an instance of an element at yet another higher layer, the M3 or metameta layer. Again, the same relationship that is present between elements of layers M0 and M1, and elements of layers M1 and M2, exists between elements of M2 and M3. Every element at M2 is an

instance of an M3 element, and every element at M3 categorizes M2 elements. Layer M3 defines the concepts needed to reason about concepts from layer M2.

Within the OMG, the MOF is the standard M3 language. All modeling languages (like UML, CWM, and so on) are instances of the MOF.

QVT (Query / Views / Transformations)

MOF QVT RFP (Request for Proposal) [4] is one of a series of RFPs related to developing the next major revision of the OMG Meta Object Facility specification, which will be referred to as MOF 2.0. Some of the RFPs pertain to specifying the technology neutral MOF itself, while others pertain to mapping the MOF to specific implementation technologies. MOF QVT RFP addresses a technology neutral part of MOF and pertains to:

1. Queries on models.
2. Views on metamodels.
3. Transformations of models.

In summary, the intent of this proposal is, to address the need of a QVT form to be used for standardization, and to establish an open standard, allowing the combination of legacy tools, MOF based technologies, a variety of modeling languages, and a large set of transformation modeling approaches, as well as allowing an easy and wide accessibility.

4.2 Metamodeling and Transformation

Within MDA we define languages through metamodels. On the other hand, the transformation rules that constitute a transformation definition describes how a model in a source language can be transformed into a model in a target language. These rules use the metamodels of the source and target languages to define the transformations.

Figure 4.3 [2] shows how the transformation and metamodeling works in MDA framework.

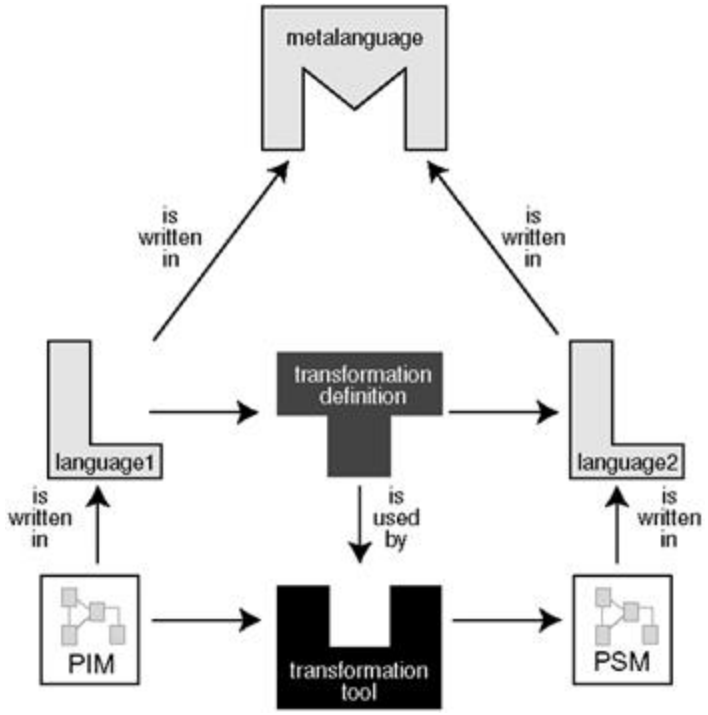

Figure 4.3 The extended MDA framework, including the metalanguage

4.3 Model Transformation Categories

At the top level, model transformation can be categorized into model-to-code and model-to-model transformation approaches. Transforming models to code is actually a special case of model-to-model transformations; it would need only a metamodel for the target model language. However, for practical reasons of reusing existing technologies, model is often generated simply as text, which is then fed into an existing system. For this reason, distinction is needed between model-to-code transformation (also known as model-to-text) and model-to-model transformation. Several tools offer both model-to-model and model-to-code transformations (e.g. ATL [23], MTF [22]).

When bridging large abstraction gaps between PIMs and PSMs, it is easier to generate intermediate models rather than go straight to the target PSM. For example, when going from a class diagram to an EJB implementation, tool would generate an intermediate EJB component model, which contains all the necessary information to produce the actual Java code from it. This makes the transformations more modular and maintainable. Also, intermediate models may be needed for optimization and tuning, or at least for debugging purposes. In addition to PIM-to-PSM transformation, model-to-model transformations are useful for computing different views of a system model and synchronizing between them, which is difficult in case of model-to-text transformation.

4.4 Model Transformation Approaches

There are many approaches noticed to achieve model transformation described above. Different MDA tools follow combination of approaches to achieve the functionality. Some of the approaches are shortly described as follows [17]:

4.4.1 Visitor-Based Approaches

A very basic code generation approach consists in providing some visitor mechanism to traverse the internal representation of a model and write code to a text stream. Example of this approach is Jamda [8]. Jamda is an object-oriented framework providing a set of classes to represent UML models, an API for manipulating models, and a visitor mechanism (so called CodeWriters) to generate code. Jamda does not support the MOF standard to define new metamodels; however, new model element types can be introduced by subclassing the existing Java classes that represent the predefined model element types.

4.4.2 Template-Based Approaches

Many of currently available MDA tools support template-based model-to-code generation, e.g., openArchitectureWare (oAW) [6], FUUT-je [46], Codagen Architect [39], AndroMDA [33], ArcStyler [37], OptimalJ [7][40] and XDE [12] (the later two also provide model-to-model transformations). AndroMDA reuses existing open-source template-based generation technology, namely Velocity [11] and Xdoclet [34]. A template usually consists of the target text containing slices of metacode to access information from the source and to perform code selection and iterative expansion.

In short, the LHS uses executable logic to access source; the RHS combines untyped, string patterns with executable logic for code selection and iterative expansion; and there is no syntactic separation between the LHS and RHS. Template approaches usually offer user-defined scheduling in the internal form of calling a template from within another one. The LHS logic accessing the source model may have different forms. The logic could be simply Java code accessing the API provided by the internal representation of the source model (e.g., JMI [5]), or it could be declarative queries (e.g., in OCL or Xpath [14]). The oAW Framework propagates the idea of separating more complex source access logic (which might need to navigate and gather

information from different places of the source model) from templates by moving them into user-defined operations of the source-model elements.

Compared to a visitor-based transformation, the structure of a template resembles more closely the code to be generated. Templates lend themselves to iterative development as they can be easily derived from examples. Since the template approaches operate on text, the patterns they contain are untyped and can represent syntactically or semantically incorrect code fragments. On the other hand, textual templates are independent of the target language and simplify the generation of any textual artefacts, including documentation.

A related technology is frame processing, which extends templates with more sophisticated adaptation and structuring mechanisms (Bassett's frames, XVCL [16], FPL [44], ANGIE [45]). FPL and ANGIE have been applied to generate code from models.

4.4.3 Direct-Manipulation Approaches

These approaches offer an internal model representation plus some API to manipulate it. It is usually implemented as an object-oriented framework, which may also provide some minimal infrastructure to organize the transformations (e.g., abstract class for transformations). However, users have to implement transformation rules and scheduling mostly from scratch using a programming language such as Java. Examples of this approach include Jamda and implementing transformations directly against some MOF-compliant API (e.g., JMI).

Direct manipulation is obviously the most low-level approach. It offers the user little or no support or guidance in implementing transformations. Basically all work has to be done by the user.

4.4.4 Relational Approaches

This category groups declarative approaches where the main concept is mathematical relations. The basic idea is to state the source and target element type of a relation and specify it using constraints. In its pure form, such specification is non-executable. However, declarative constraints can be given executable semantic, much like in the case of logic programming. All of the relational approaches are side-effect-free. They often support backtracking and, in contrast to the imperative direct manipulation approaches. Relational specifications can be interpreted bi-directionally. Logic programming-based approaches also naturally support bi-directionality. But some approaches fix the direction for executable transformations. Logic programming-based approaches require strict separation between source and target models (i.e., they do not allow in-place update).

Relational approaches seem to strike a well balance between flexibility and declarative expression. They provide flexible scheduling and good control of nondeterminism.

4.4.5 Graph-Transformation-Based Approaches

This category of model transformation approaches draws on the theoretical work on graph transformations [19]. In particular, these approaches operate on typed, attributed, labelled graphs, which is a kind of graphs specifically designed to represent UML-like models. Examples of graph-transformation approaches to model transformation include ATOM [47], GreAT [48], UMLX [50], and BOTL [53].

Graph transformation rules consist of a LHS graph pattern and a RHS graph pattern. The graph pattern can be rendered in the concrete syntax of its respective (source or target) language or in the MOF abstract syntax. The former is preferred since for complex syntaxes (like UML) the later may result in huge patterns even for relatively small transformations. The LHS pattern is matched in the model being transformed and replaced by the RHS pattern in place. The LHS often contains conditions in additional to the LHS pattern (e.g., negative conditions). Some additional logic (e.g., in string and numeric domains) is needed in order to compute target attribute values (such as element names). In most approaches, scheduling has external form and the scheduling mechanisms include nondeterministic selection, explicit condition, and iteration.

Graph-transformation-based approaches are inspired by heavily theoretical work in graph transformations. These approaches are powerful and declarative, but also the most complex ones. The complexity stems from the nondeterminism in scheduling and application strategy, which require careful consideration of termination of the transformation process and the rule application ordering (including the property of confluence). There is a large amount of theoretical work and some experience with research prototypes. However, experience with practical applications of these approaches is still limited. It remains to be seen how well the complexities of these approaches will be received in practice.

4.4.6 Structure-Driven Approaches

Approaches in this category have two distinct phases: the first phase is concerned with creating the hierarchical structure of the target model, whereas the second phase sets the attributes and references in the target. The overall framework determines scheduling and application strategy; users are only concerned with providing the transformation rules. An example of the structure-driven approach is the model-to-model transformation framework provided by OptimalJ. The framework is implemented in Java and provides so-called incremental copiers that users have to subclass to define their own transformation rules. The basic metaphor is the idea of copying model elements from the source to the target, which then can be varied to achieve the desired transformation effect. The framework uses reflection to provide a declarative interface. A transformation rule is implemented as a method with an input parameter whose type determines the source type of the rule, and the method returns a Java object representing the class of the target model element. Rules are not allowed to have side effects and scheduling is completely determined by the framework.

The structure-driven category groups pragmatic approaches that were developed in the context of (and seem particularly well applicable to) certain kinds of applications such as generating EJB implementations and database schemas from UML models.

These applications require a strong support for transforming models with a 1-to-1 and 1-to-n (and sometimes n-to-1) correspondence between source and target elements. Also, in this application context, there is typically no need for iteration (and in particular fix pointing) in scheduling, and the scheduling can be system-defined. It is unclear how well these approaches can support other kinds of applications.

4.4.7 Hybrid Approaches

Hybrid approaches combine different techniques from the previous categories. The Transformation Rule Language (TRL) [52] is a composition of declarative and imperative approaches. It could be also classified in the relational category, but its better to classify it separately because of its stronger imperative component. Similar to QVT Proposal, it distinguishes between specification and implementation. A mapping rule in TRL declares a relationship between source and target elements that is constrained by a set of invariants. They are similar to relations and fit into the relational category. Operational rules in TRL represent executable transformation rules. In contrast to mapping rules, operational rules explicitly state whether a rule creates, update, or deletes elements. Scheduling is explicit in internal form, where a rule explicitly calls other rules in its body. Rule inheritance is supported. Rules can be organized into modules (called *units*). Inheritance between modules (with overriding) is also supported.

Rational XDE is an example of a highly hybrid approach. XDE supports model-to-model transformation through its pattern mechanism. Patterns can be associated with JSP-like code templates (so-called scriptlets) in order to perform model-to-code transformation.

The Atlas Transformation Language (ATL) is also a hybrid approach, which has some similarities to TRL. A transformation rule in ATL may be fully declarative, hybrid, or fully imperative. The LHS of a fully declarative rule (so-called source pattern) consist of a set of syntactically typed variables with an optional OCL constraint as a filter or navigation logic. The RHS of a fully declarative rule (so-called target pattern) contains a set of variables and some declarative logic to bind the values of the attributes in the target elements. In a hybrid rule, the source and/or target pattern are complemented with a block of imperative logic, which is run after the application of the target pattern. A fully imperative rule (so-called procedure) has a name, a set of formal parameters, and an imperative block, but no patterns. Rules are unidirectional and support rule inheritance. ATL strictly separates source and target models; however, in-place transformation can be simulated thanks to an automatic copy mechanism. ATL provides both implicit and explicit scheduling. The implicit scheduling algorithm starts with calling a rule that was designated as an entry point, which may call further rules. After completing this first phase, it automatically checks for matches on the source patterns and executes the corresponding rules. Finally, it executes a designated exit point. Explicit, internal scheduling is supported by the ability to call a rule from within the imperative block of another rule.

Hybrid approaches allow the user to mix and match different concepts and paradigms depending on the application. Practical approaches are very likely to have the hybrid character.

4.4.8 EMF Ecore model based Approaches

In fact the approaches undertaken in this category can be described as one or more of the above approaches, but because of the fact that EMF models are not the extension of the OMG MOF models in any way, its better to consider as a different approach.

The OMG MOF Model has influenced the design of the EMF Ecore model. The Ecore model evolved in parallel with the MOF 1.4 model. Implementation experience in integrating a number of tools led to an optimized implementation (focused on tool integration as opposed to the original MOF focus of metadata repositories) that uses a subset of the modeling concepts in MOF 1.4. For example, Ecore does not support 'first class' Associations. Associations are mapped to a pair of Ecore References. To minimize confusion.

One of the key goals of EMF is to use simple visual models to allow easy integration of Java and XML tools. To accomplish this one can use UML class models as input to the EMF framework. This model is then used to drive Java interface and implementation generation for EMF instances. These java interfaces define a consistent programming model for tools built using EMF. The same EMF model is also used to generate the XML serialization (XMI 2.0 format) for EMF instances. Essentially EMF supports the key MDA concept of using models as input to development and integration tools which produce multiple programming language (Java in the case of Eclipse EMF itself) or data interchange format (XML) representations. The code generation or XML serialization is 'driven' from the same model.

As part of its involvement in the QVT standardization, IBM has developed a prototype model transformation toolkit, MTF. MTF implements some of the QVT concepts and is based on the EMF. It provides a simple declarative language for defining mappings between models, along with a transformation engine that can interpret mapping definitions in order to perform transformations. The aim of MTF is to simplify ability to develop transformation tools by supporting incremental update, round-tripping, reconciliation, and traceability.

An MTF transformation results in a set of *mappings* that relate objects among different models; Transformations in MTF are defined in a declarative way: you specify a set of relations between model classes, and then let the MTF engine perform the transformation actions using these relations as input. The transformation engine proceeds in two stages called *mapping* and *reconciliation*. During the mapping stage, it evaluates the relations and generates mappings by iterating through the relevant model instances. At the end of this stage, some mappings may be inconsistent with respect to the relation. In other words, not all the imposed constraints of the relation are satisfied.

Reconciliation tries to satisfy the relations by creating missing elements, modifying existing elements, or deleting elements. In some cases, reconciliation may not be needed (when the models are already consistent, or if you only want to check the consistency of models without changing them).

Similar to a rule-based system, it is possible to invoke one relation from another, and propagate the execution of the mapping to related model classes. This mechanism is called *correspondence* and allows MTF to traverse a whole model by applying all the

correspondences reachable (directly or transitively) from the top-level relation. Relations are expressed in a language called the Relation Definition Language (RDL). MTF is intended specifically for transforming Eclipse modeling Framework models, although you can work with other Java object models by using the MTF model extension mechanism to create EMF wrappers and also supports generation of text via an EMF model of document templates. MTF supports extensions and custom constraints, which allow one to extend the MTF mapping definition language. MTF is based on recording mappings between elements. Mappings may be persisted, re-loaded, and used for reconciling changes, and they can accessed from your own code. But the only restriction is: MTF is useful and simple to use only in the case of EMF Ecore Model transformation and not others like OMG MOF.

4.4.9 Other Model-To-Model Approaches

At least two more approaches [2] should be mentioned for completeness: the transformation framework defined in the OMG's Common Warehouse Metamodel (CWM) Specification and transformation implemented using XSLT.

The CWM transformation framework provides a mechanism for linking source and target elements, but the derivation of the target elements has to be implemented in some concrete language, which is not prescribed by CWM. Effectively, CWM gives a general model, but no actual mechanism to implement model transformations.

Since models can be serialized as XML using the XML Metadata Interchange (XMI), implementing model transformations using XSLT, which is a standard technology for transforming XML, seems very attractive. Unfortunately, this approach has severe scalability limitations. Manual implementation of model transformations in XSLT quickly leads to non maintainable implementations because of the verbosity and poor readability of XMI and XSLT. A solution to overcome this problem is to generate the XSLT rules from some more declarative rule descriptions. However, even this approach suffers from poor efficiency because the copying required by the pass-by-value semantics of XSLT and the poor compactness of XMI.

Following is the result [18] of comparing four main kinds of transformation approaches:

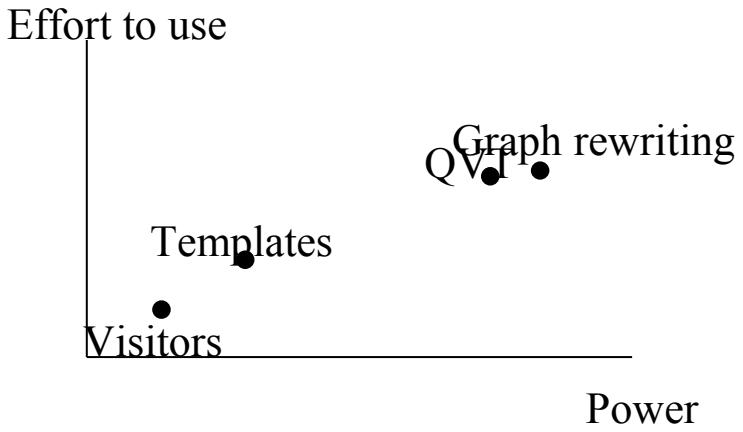

Figure 4.4 Comparison of four main kinds of transformation approaches

4.5 ATLAS Transformation Language

4.5.1 Introduction

ATL is the ATLAS INRIA & LINA research group answer to the OMG MOF/QVT RFP. It is a metamodel-based transformation DSL (Domain Specific Language). The scope of ATL is however not limited to the OMG set of recommendations as the intention is to cover other technical spaces as well. It is a model transformation language specified both as a metamodel and as a textual concrete syntax. It is a hybrid of declarative and imperative. The preferred style of transformation writing is declarative, which means simple mappings can be expressed simply. However, imperative constructs are provided so that some mappings too complex to be declaratively handled can still be specified. Once complex mappings patterns are identified, declarative constructs can be added to ATL in order to simplify transformation writing.

An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. A program in ATL is considered as a model, taking a model as input and producing a model as output. Multiple input and output parameters are supported as well.

The work on ATL is a collaboration between the University of Nantes and INRIA and initially with TNI company. ATL has been chosen as the model transformation technology for the "ModelWare" IST European project in collaboration with SINTEF (Norway). It is currently being used by several research groups working in different domains and also for teaching.

4.5.2 The ATL execution engine architecture

A model-transformation-oriented virtual machine has been defined and implemented to provide execution support for ATL while maintaining a certain level of flexibility. As a matter of fact, ATL becomes executable simply because a specific transformation from its metamodel to the virtual machine bytecode exists. Extending ATL is therefore mainly a matter of specifying the new language features execution semantics in terms of simple instructions: basic actions on models (elements creations and properties assignments).

This flexibility is important for two main reasons: ATL will need to be aligned with the QVT standard when it is adopted in 2005 and, as a research project, it can this way easily benefit from newly defined features. In the same way the Java Virtual Machine (JVM) instruction set can directly deal with objects, the ATL Transformation Virtual Machine (TVM) directly handles model elements. The present prototype version is a stack machine with less than twenty instructions. It is built on top of a Java-based model repository abstraction layer, which is the key to the current version’s lower level adaptability.

Lower level adaptability of ATL has already been proven by porting the engine from the Sun MDR/NetBeans to the Eclipse/EMF environment. Taking into account other

underlying platforms should not be a big challenge. This means that ATL may be easily integrated in various open MDD platforms.

4.5.3 Available developing tools for ATL

An IDE has been developed for ATL on top of Eclipse: ATL Development Tools (ADT). It uses EMF, the Eclipse Modeling Framework, to handle models: to serialize and deserialize them, to navigate and to modify them. A specific code editor, including syntax highlighting and an outline view of the program, is implemented as a convenience.

This IDE also includes a specific ATL extension of the Eclipse debugging framework enabling source-level debugging of transformation programs. Single step, step over and breakpoints support makes it possible for the developer to precisely control the execution of the transformation program being written. When the execution is suspended, it is possible to navigate into source and target models from the current context as well as into user-defined variables. ADT is about to be released as part of the Eclipse GMT project under the EPL (Eclipse Public License).

5 CxPT – A matured MDA tool prototype

5.1 CxPT - CORYX Platform Technology

CxPT is a Framework, which makes a fast development possible of enterprise applications. The platform is based on the criteria of the Model Driven Architecture (MDA) and uses the consistent separation of the Business logic from architectural details. Automatic code generators support the fast development of enterprise applications on a large scale. The majority of the experts assumes such beginnings will considerably determine the future of the software development. The CxPT converts these beginnings for the development of applications of business to basis of J2EE innovatively. The application developers do not have to control the complex technical concepts of the J2EE platform. The data structures and other system specification are defined in XML and the application logic is programmed in Java. A generator takes over then the production of the necessary J2EE artifacts as well as all model classes, Registries and Metaobjects necessary for application programming.

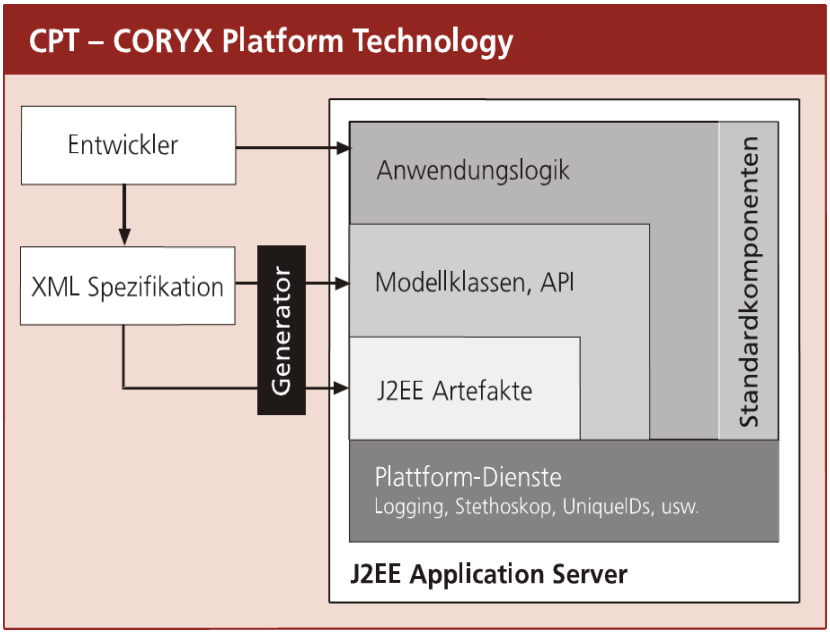

Figure 5.1 Basic concept of the CxPT

5.1.1 CxPT Framework

The CxPT is to serve business processes, which include business practices on the server side in business services. It lets the Clients or the services communicate by models. This communication is completely transparent for the participants. The Client uses Business Delegate Objects for inquiries to the services. On the server side the services communicate directly with one another - without delegation overhead.

On the other hand, service Locator and storage mechanism (persistance objects) for business services are not visible to the developers. For the fast use of the middleware technology represented above the platform has a GUI package which is based on Swing. Its most important characteristics are the illustration of the model into the user interface control, making complex basis functionalities, standardized dialogue types available, user Experience of element and panel Technology, report engine and Action set interpreter.

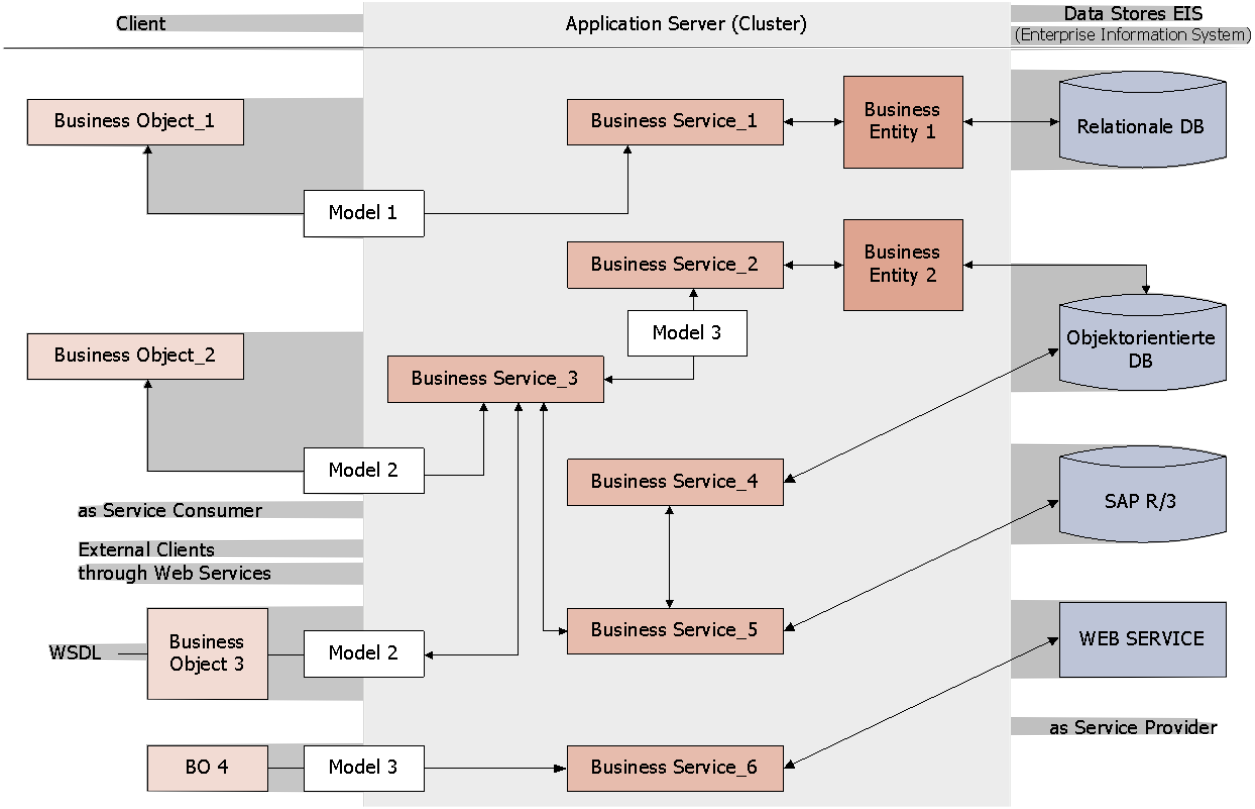

Figure 5.2 Enterprise system architecture generated by CxPT

The functional requirements are described best by applications (use cases), which will help to identify well isolatable and clear functions. These functions can be grouped and different views (e.g. similar functions; Functions, on the same participant or on a workflow etc.) and to services can also be aggregated. The services are put to the client disposal, without he knows something about their contents.

Models represent the static entities of application, which is combined by the business entities according to the requirements of application. The services represent the dynamic functions of application. The models are Java objects. They are collected by the business and domain entities with the help of the middleware and transferred by a business service method to the client. The client shows the model on the user level and permits it to modify it. Subsequently, it is back handed over another method of same service to the middleware. These steps can be repeated more arbitarily, in order to realize an entire workflow. The services are disposed to the client by the business objects, which works like a proxy and which make possible location-transparent remote access including fail over and load balancing.

5.1.2 Structure of the generated service artefacts

In this part some information about how the source code is structured within services is given. The example in the following illustration shows a service, which is responsible for the treatment of customized data.

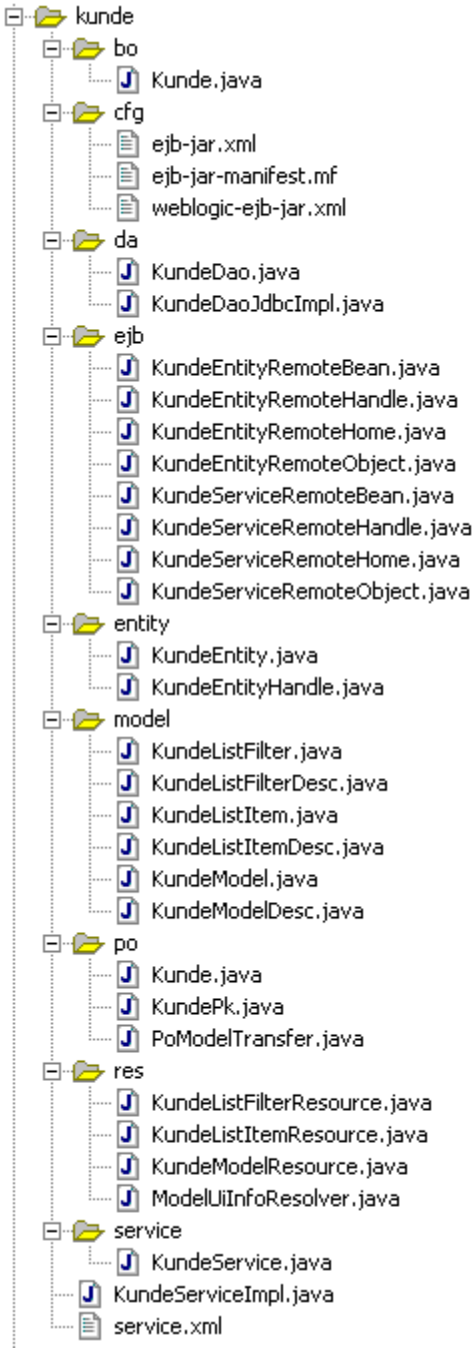

Figure 5.3 Generated service structure (java file listing)

The developers must provide a file *"service.xml"*, in which he name the Business services, model information based on database tables, identification of the user interface, data access mechanism and transfer requirements. It requires to write the business methods in Java. In this case the method will work on the specified models. Then through calls of the Generator tool all other objects are generated automatically. That means, all models and model descriptor objects are written into the *model* sublist as seen in the Figure. (note: a model descriptor is an object, which describes the fields of the model, e.g. the maximum length of a string or the accuracy or attributes of numeric fields etc.). Under the *po* sublists persistable object classes are generated, which used as a transfer objects while reading from or writing into a relational database and/or into another storage medium such as XML or OO databases, implemented in the CxPT generator). Into the *bo* sublist the business objects for the Client are generated. With the help of this object the client communicates with the server. In the *ejb* and *cfg* sublists the J2EE platform-specific classes are generated. In the *res* sublist language dependant label and error/warning messages are managed.

An example service.xml can be seen as below:

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2
3  <service name="Kunde" type="Stateless">
4     . . .
5     <meta-model>
6        <model name="KundeModel">
7           <attributes> . . . </attributes>
8           <labels> . . . </labels>
9        </model>
10    </meta-model>
11
12    <data-access>
13       <data-access-object name="KundeDao" data-source="CoryxRdbDs"/>
14    </data-access>
15
16    <entities>
17       <entity name="KundeEntity">
18          <model>KundeModel</model>
19          <persistable-object data-source="CoryxRdbDs">CptKunde</persistable-object>
20          . . .
21          <field-mapping>. . .</field-mapping>
22          . . .
23       </entity>
24    </entities>
25
26    <business-methods>
27       <entity-selector name=". . .">. . .</entity-selector>
28       . . .
29       <business-method name=". . .">. . .</business-method>
30       . . .
31    </business-methods>
32
33    <business-objects>
34       <business-object name="KundeManager">
35          <served-methods>. . .</served-methods>
36          <served-entities>. . .</served-entities>
37          <served-entity-selectors>. . .</served-entity-selectors>
38       </business-object>
39       . . .
40    </business-objects>
41 </service>
42
```

Now after understanding about service generator and business logic definition the point is how to assist them in a project more efficiently. The file "application.xml" is of central importance, which is provided only once for a project. Therein all necessary data objects including data source are defined for the support of the business logic, and also mention required "services" - in XML notation. Each business service is defined and described in separate "service.xml". The individual services are referenced in "application.xml" and assigned to the project. From this file, other "application.xml" files can also be referenced.

A snippet of an example application.xml can be seen as follows:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <application name="KundeApplication">
4    <namespace>de.coryx.apps.kunde</namespace>
5    ...
6    <data-sources>
7      <jdbc-data-source name="CoryxRdbDs">
8        ...
9        <persistable-objects>
10         <persistable-object name="Kunde" table="CPT_KUNDE"/>
11         ...
12       </persistable-objects>
13       ...
14     </jdbc-data-source>
15     <hibernate-datasource name="...">
16       <mapping>...</mapping>
17     </hibernate-datasource>
18     <jdo-datasource name="...">
19       <mapping>...</mapping>
20     </jdo-datasource>
21     <generic-ra-datasource name="...">
22       <data-access-object>...</data-access-object>
23     </generic-ra-datasource>
24   </data-sources>
25
26   <services>
27     <service>
28       <location>service/org/service.xml</location>
29     </service>
30     ...
31   </services>
32
33   <ext-applications>
34     <application name="extApp2">srctest/data/application2.xml</application>
35     ...
36   </ext-applications>
37
38   <ext-services>
39     <service>UniqueSequenceGenerator</service>
40   </ext-services>
41 </application>
42
```

5.1.3 Benefits

With the employment of the Coryx Platform Technology, EJB related objects, which make the business logic and the data base access possible, are hidden and must be called only over methods. Whether the platform uses Enterprise Java Beans or other methods, is perfectly uninteresting. The interfaces for the access always remain the

same. The software developer does not notice therefore a possible change of this technology. This independence from future technological change, the fact that a frontend programmer does not have to know the functionality of components being under it, are advantages with the employment of the CxPT.

It is no matter, which data base management system is used, Informix, Oracle or MySQL, object-oriented or relational DBMS, Web services, file systems, mixed applications or the parallel access Enterprise integration of systems e.g. on SAP/R3. A conversion is possible also in the future at small expenditure - and thereby, the Business logic remain untouched. With the GUI programmed in Java, the application is independent of the operating system.

The developer has more time for the realization of the technical requirements, since the manual tasks of programming can be reduced. Work procedures can be saved and the development of the software be accelerated.

Productivity rises substantially: One does not have to write the J2EE-specific code. One does not have to look for J2EE-specific error (that can be quite difficult to solve, if one has little experience). The J2EE training expenditure can be avoided for CxPT which requires substantially smaller !

INTEGRATION: The software system provided with the MDA and the CxPT has a higher measure of interoperability, i.e. the exchange of information in heterogeneous systems is facilitated by the generation of suitable interfaces.

MAINTENANCE: The higher quality of the automatically produced source code facilitates the maintenance work. It generates the code based on number of design patterns available.

STANDARDISATION: The components are standardized and coupled loosely with one another. They can be more flexibly used thereby and built up for new software.

EXPANDABILITY: The business process can be extended at small expenditure by new functionalities.

USER FRIENDLINESS: Pre-defined dialogue elements (filter, report...) improve the operability of the software.

INVESTMENT SECURITY: The concentration on the Design models, which are basis for the generation, leads to an completely documented and clearly structured architecture. The investment security of customers increases.

5.1.4 Missing Links

There are some missing links in CxPT as an matured MDA tool. Links where consistency must be checked at the PIM level and then application of transformation rules to transform PIM to PSM, which then can be used to generate J2EE based middleware solution as before in CxPT. Filling this gap will make CxPT more consistent, time effective, more maintenable and stable in development of its solutions.

The eXtensible Markup Language (XML) has gained acceptance as a configuration and specification language for different software artefacts. It is also used as a mechanism for bridging data heterogeneity problems. XML has simplified the creation of domain-specific markup languages. It is accomplished by a set of powerful technologies like Xpath which supports the selection of sets of elements from XML documents by standardising a language for paths in trees. XLink is the linking language for XML. An XLink consists of a set of locators which identify the resources connected by the link. XLink greatly improves the linking facilities available for hypertext authors over those available in HTML anchors: it can link more than two documents, links do not have to be inside the documents being linked and link traversal behaviour can be specified. When combined with a language like Xpath, XLink can be used at a fine-grain level to relate elements rather than simple documents.

So constraint based consistency checking of xml documents are becoming an area of concern.

Schematron enables the specification of assertions about the structure of documents and uses XSLT to evaluate the assertions. Schematron is a widely used, lightweight approach to semantic document validation. It does however not posses the expressive power of the language since, by using pure XPath expressions, it essentially builds on a boolean logic. It also does not provide support for checking inter-document relationships and does not eases the task of finding out the cause of broken constraints.

Xlinkit leverages open standards such as XML, Xlink and Xpath in order to bridge heterogenety problems and allow internet scale distribution of development activities. Xlinkit enables checking simple consistency relationships of elements in XML documents, and the formal specification of a semantics also enables the generation of hyperlinks between inconsistent elements.

The Object Constraint Language can be used to specify static constraints specifically on UML based models. OCL is more expressive than xlinkit, allowing the definition of functions and permitting the use of infinite sets such as the integers in constraints.

5.1.5 Impose constraints on CxPT specification

It would be better that CxPT xml specification files (application.xml and service.xml) metamodeled in uml and then ocl constraints can be imposed on the instances later on using any of the tools supporting ocl. I will use Octopus tool for the same purpose to demostrate the prototype solution.

Currently CxPT xml specification files are metamodeled by XSD files, will be used now to metamodel uml specification. For better uml metamodeling its better not to make it automatised and use heuristics instead. Following is the snippets of uml metamodel in OctopusUML's syntactic format derived from application.xml.

Example part of application.xml to be converted into uml:

```
<xs:element name="application">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="namespace" type="xs:string"/>
      <xs:element name="data-sources" type="DataSourcesType" . . ./>
       ...
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:complexType name="DataSourcesType">
  <xs:sequence>
    <xs:element name="jdbc-data-source" type="JdbcDataSourceType" . ./>
    <xs:element name="hibernate-datasource" type="HiberDatasourceType"/>
    <xs:element name="jdo-datasource" type="JdoDatasourceType" . . ./>
    <xs:element name="generic-ra-datasource" type="GenRaDsourceType"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="JdbcDataSourceType">
  <xs:sequence>
    <xs:element name="namespace" type="xs:string"/>
    <xs:element name="persistable-objects" type="PObjectsType" . . ./>
    <xs:element name="table-registry" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="HibernateDatasourceType">
  <xs:sequence>
    <xs:element name="mapping" type="xs:string" . . ./>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="JdoDatasourceType">
  <xs:sequence>
    <xs:element name="mapping" type="MappingType" . . ./>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="GenericRaDatasourceType">
  <xs:sequence>
    <xs:element name="data-access-object" type="xs:string" . . ./>
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
</xs:complexType>

<xs:complexType name="PObjectsType">
  <xs:sequence>
    <xs:element name="persistable-object" type="PObjectType" . . ./>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="PObjectType">
  <xs:attribute name="name" type="NameType" use="required"/>
  <xs:attribute name="table" type="xs:string" use="required"/>
</xs:complexType>
```

Derived uml specification from the above xml schema:

```
<package> application

 + <class> Application
<attributes>
 + name: String;
 + path: String;
 + namespace: String;
 + serviceNamespace: String;
<endclass>

+ <abstract><class> DataSource
<attributes>
 + name: String;
<endclass>

 + <class> JdbcDataSource <specializes> DataSource
<attributes>
 + namespace: String;
 + tableRegistry: String;
<endclass>

 + <class> HibernateDataSource <specializes> DataSource
<attributes>
 + mapping: String;
<endclass>

 + <class> JdoDataSource <specializes> DataSource
<attributes>
 + mapping: String;
<endclass>

 + <class> GenericRaDataSource <specializes> DataSource
<attributes>
 + dataAccessObjectName: String;
<endclass>

...
 + <class> PersistableObject
<attributes>
 + name: String;
 + table: String;
<endclass>
...
<associations>
 - Application.application [1..1] -> + DataSource.dataSources [0..*];
 - JdbcDataSource.jdbcDataSource [1..1] -> +
             PersistableObject.persistableObjects [1..*];

 . . .
<endpackage>
```

As application.xml aggregates many service.xmls we dont need separate metamodel for service.xml we can incorporate that into the single metamodel in a following way:

```
+ <class> Service
<attributes>
 + location: String;
 + name: String;
 + serviceType: ServiceType;
<endclass>

+ <enumeration> ServiceType
<values>
 Stateless;
 Stateful;
<endenumeration>
...

<associations>

 - Application.application [1..1] <->  + Service.services [1..*]  ;
```

And then it will be the responsibility of transformation to produce separate xml instances.

In Octopus apart from defining uml metamodel, ocl constraints can be defined in separate *.ocl* files.

```
context Application

-- application name must not be blank
inv application_name_must_not_be_blank:
      self.name.size() > 0

-- application path must not be blank
inv application_path_must_not_be_blank:
      self.path.size() > 0

-- application namespace must not be blank
inv application_namespace_must_not_be_blank:
      self.namespace.size() > 0

-- application service namespace must not be blank
inv application_service_namespace_must_not_be_blank:
      self.serviceNamespace.size() > 0

-- application name must be unique
inv application_name_must_be_unique_in_all_applications:
      allInstances()->select(name=self.name)->size() = 1

-- datasource name must be unique locally to the application
inv datasource_name_must_be_unique_locally_to_application:
      dataSources->collect(e | e.name)->size() = dataSources->collect(e
| e.name)->asSet()->size()
```

```
-- referenced external service name must be unique locally to the
application
inv
referenced_external_service_name_must_be_unique_locally_to_application:
      refExtServices->collect(e | e.name)->size() = refExtServices-
>collect(e | e.name)->asSet()->size()

-- service name must be unique locally to the application
inv service_name_must_be_unique_locally_to_application:
      services->collect(e | e.name)->size() = services->collect(e |
e.name)->asSet()->size()

-- external applications name must be unique locally to the application
inv external_application_name_must_be_unique_locally_to_application:
      extApplications->collect(e | e.name)->size() = extApplications-
>collect(e | e.name)->asSet()->size()
```

complex invariants can be defined in ocl in simple way, few of them are as follows.

```
context Service

. . .

-- referenced service must not refer self
inv referenced_service_must_not_refer_self:
      refServices->collect(e | e.name)->size() = refServices->collect(e
| e.name)->excluding(self.name)->size()

-- reference enumeration of own service not needed
inv reference_enumeration_of_own_service_not_needed:
      refEnumerations->forAll(e:ReferenceEnumeration |
e.refService.name <> self.name)

-- dataAccessObject must refer to the datasource from the owned
application
inv
dataAccessObject_must_refer_to_the_datasource_from_the_owned_applicatio
n:
      dataAccessObjects->forAll(dao:DataAccessObject |
application.dataSources->collect(e | e.name)-
>includes(dao.dataSource.name))
```

XMI Import

Octopus also allows developers to import existing UML model developed in UML based tool like Poseidon into OctopusUML models using xmiimport plugin module. The XMI Import module is based on a set of XSL stylesheets for conversion of an UML XMI file to a set of Octopus UML files. It utilizes the eXtensible Stylesheet Language Transformation (XSLT) technology.

Limitations

There are some limitations to be consider before using xmi import in Octopus:

- Octopus only allows packages as top level UML modelelement, no other modelelements are allowed (such as classes).

- Octopus does not support 'package' visibility.
- Octopus does not support sequences of multiplicity ranges, such as [1..3,4..6].
- Since Octopus is an OCL tool, it only has the following basic predefined primitive types: Integer, Real, Boolean, String and OclVoid. However, Octopus allows the user to provide a mapping (using the typeconversions.properties file).

Octopus is able to create Java code from the UML model. Optionally, code implementing the OCL expressions is generated as well. The Java code generated by Octopus complies with Java 1.4. One may customize the code generation process by selecting or deselecting the options given in the project properties for Octopus Code Generation. Typically octopus generates 3 tiers of java code: the middle tier, storage layer and user interface tier.

The Middle tier

Ocl expressions

- **Invariants:** For each invariant a public method that checks the invariant is created. This method is called *invariant_X*, where X is the name of the invariant, or when the name is not given, X is the name of the class postfixed with a unique number. If the check fails an *InvariantException* is thrown. As a convenience an method is generated that checks all invariants of the class. This is called *checkAllInvariants*. It returns a list of *InvariantError* objects. The user of the generated code is free to check invariants whenever it is appropriate.
- **Initial Values and Derivation Rules**: OCL expressions that denote initial values of either an attribute, or association end, are implemented in the constructor(s) of the class. If the attribute or association end is static the initial value will be part of the declaration of the corresponding Java field. A derivation rule is transformed into a *get* method for the attribute or association end for which the rule was defined. Note that there will be no *set* method, nor field for a derived feature.
- **Bodies, Pre and Post conditions:** A precondition will be transformed into an assert statement. Postconditions are not transformed into code. A body expression will, of course, be transformed into the body of the method.
- **OCL Defined attributes and operations:** Operations defined by an OCL (def) expression are implemented in the same manner as other operations. Its expression will be implemented as body expression. Attributes defined by an OCL (def) expression are implemented as attributes with a derivation rule. There will be a *get* method only, no field or *set* method.
- Any incorrect ocl expression will simply be ignored.

Association Multiplicities

The multiplicity check is implemented in a separate method called *checkMultiplicities*, that can be called whenever it is appropriate to check the multiplicities of an object.

Other convenience methods

The following convenience methods can be generated by Octopus (ClassName stands for the name of the class in which these methods are generated):

- public ClassName getCopy(),
- public void copyInfoInto(ClassName copy),
- public String getIdString(),
- public String toString(),
- public Collection allInstances().

Generation of Visitor interfaces

Octopus is able to generate two kinds of visitor interfaces. The first visitor interface is able to visit every class in the project. This interface is named using the project name: I<projectname>Visitor. In every class an accept operation will be added that ensures that every attribute or association end in the instances of the class will also be visited. Note that some objects may therefore be visited more than once.

The second type of visitor interface is dedicated to visiting a certain class, its parts, and its subclasses. Only true parts are visited by visitors of this type.

The user may indicate the class for which to generate visitor interfaces of the second type in the special 'Visitors' tab in the properties page. Visitor interfaces of this type are named using the class name: I<classname>Visitor. All visitor interfaces will be placed in the *utilities* package.

None of the generated visitor interfaces will visit instances of association classes. Instances at both sides of the association, however, will be visited. One has to decide at what end of the association responsible to handle the instance of the association class, and implement this in the visit operation of the class on that end.
Generation of Interfaces and 'internal' Package

Octopus is able to generate a facade of interfaces for the UML model. For each class in a UML package, a Java interface in the corresponding Java package is generated. A new Java sub package, called *internal*, is created as well. This sub package contains the implementations of all the interfaces in the package.

In Two-way Navigable Associations is explained that for some associations two extra methods are created named *z_internalAddToX*, and *z_internalRemoveFromX*, where X is the name of the association end. The facade interface will not contain these methods, but it will contain the *addToX*, and *removeFromX* methods.

Using this option together with Post Generation Editing

Note that this option may be used together with the option to generate separate classes for post generation editing. In that case a package *P* that contains classes *Aa* and *Bb* will be transformed into a Java package *P* that contains interfaces *IAa* and *IBb* and a subpackage called *internal*. The *internal* package contains the empty placeholders for extra methods and fields called *A* and *B* and a subpackage called *generated*. The *generated* package contains the classes that hold the generated code. They are called *AaGEN* and *BbGEN*.

Generating User Interface

Within octopus it is possible to generate a user interface as a proof of concept (because it works only for small and less complex models). The user interface can be generated for plugin projects only.

The Navigator view

The application starts with the Navigator view opened. From the Navigator view new instances may be created and existing instances may be explored. In it all instances are sorted according to their class. Only classes that were selected in the project properties are included. Double clicking on one of the class names will refresh the Navigator view. An example of the navigator view is shown in the figure below.

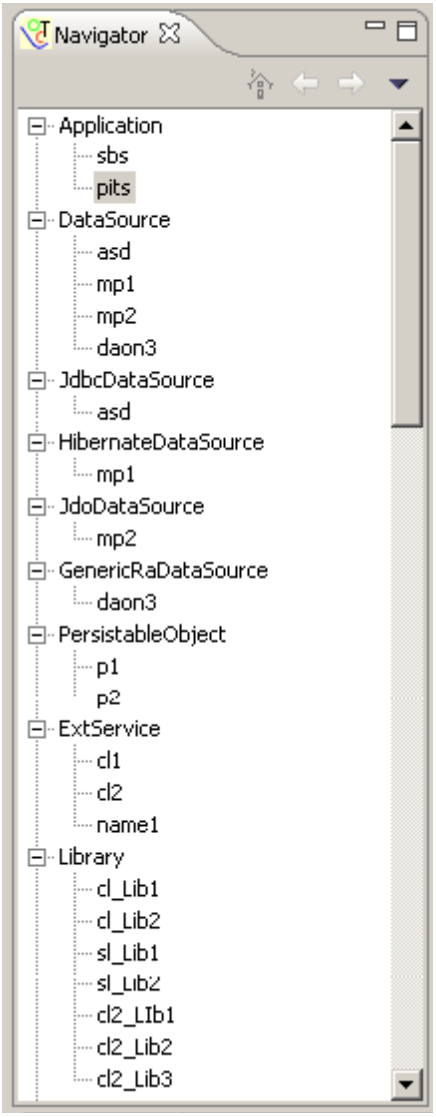

Figure 5.4 Navigator view

One may create a new instance of a certain class by using the context menu on the class name. Double clicking in the Navigator View on one of the instances will open an object detail window. For example Application object view can be viewed as follows:

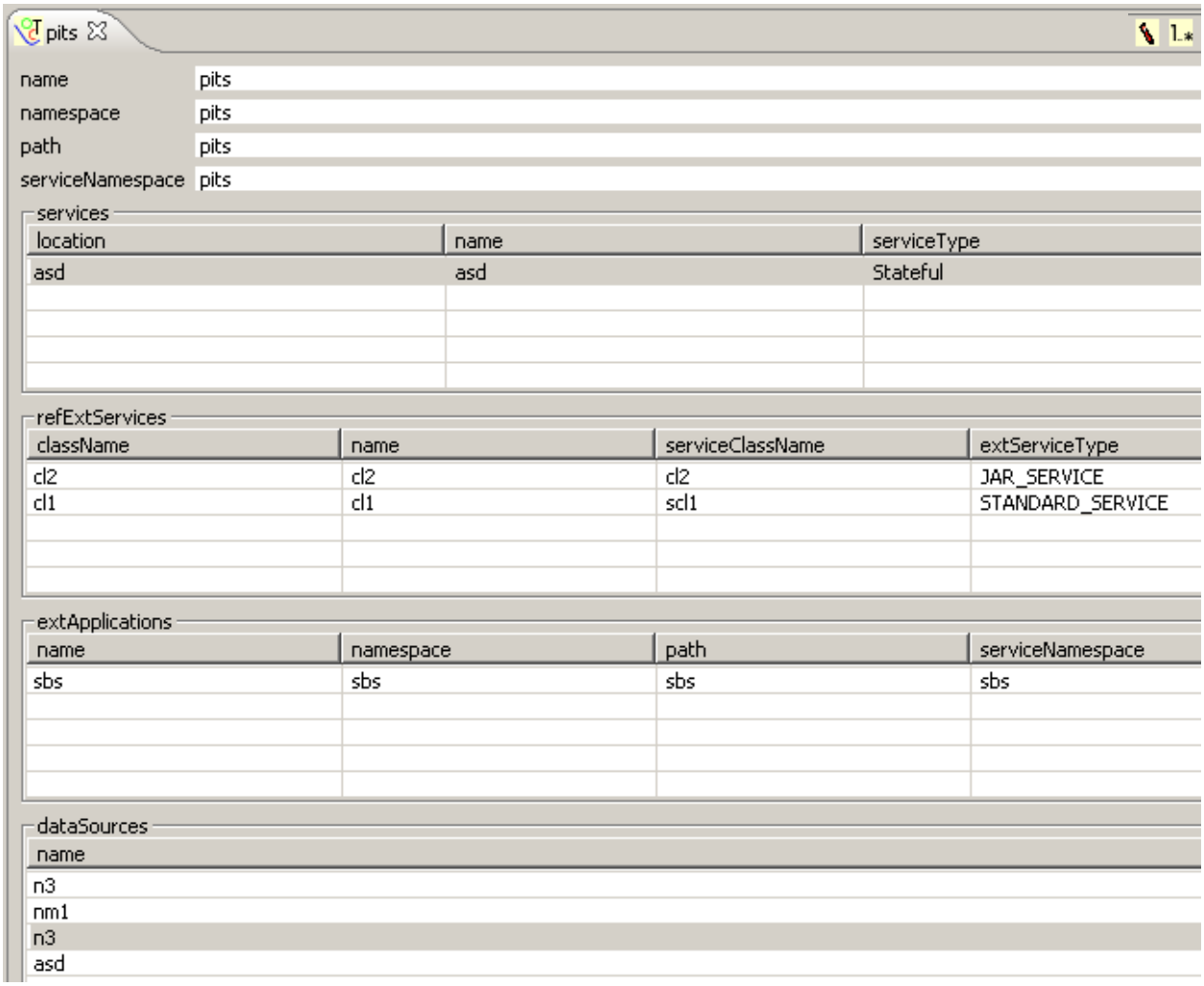

Figure 5.5 Object detail view

In the right upper corner of the Object Detail View there are two buttons:

- *Check Invariants* (on the left): checks all of the OCL invariants from the UML model for this object. The result is a new window showing all broken invariants.
- *Check Multiplicities* (on the right): checks all of the multiplicities from the UML model for this object.

The Actions in the toolbar

In the toolbar one will find three buttons. The left button opens an XML file in which instances of this model are stored. The middle button saves the current instacnes to file. The third button performs the *Check Invariants* and *Check Multiplicities* actions on all instances in the system. And all broken invariants and association multiplicity violations will be reported in a invariants view. Example can be seen in a following figure 5.6:

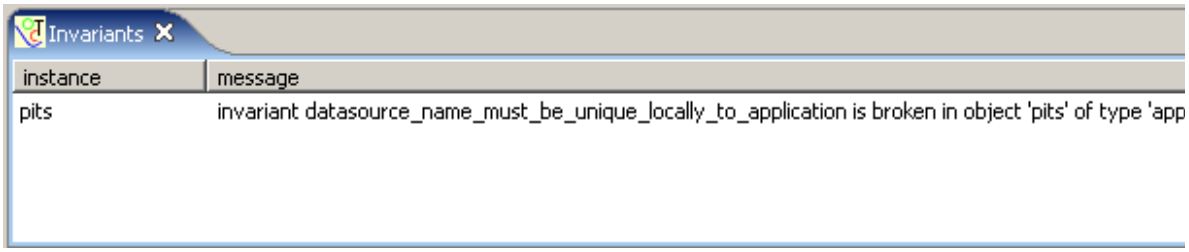

Figure 5.6 Broken invariants view

Using invariants view objects which are violating invariants can be navigated easily and corrected in the object detail view then.

XMLStorage

By default octopus will generate xml storage layer (consists XMLReader and XMLWriter) so that it can store the created objects in a xml format. Storage related code generated under *xmlstorage* package. Octopus persists all xml data according to *octopus-storage.xsd* shema file:

```
<xs:schema . . .>
  <!-- storage ::= instance* -->
  <xs:element name="storageRoot">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="instance" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
      <xs:attribute ref="project" use="required"/>
      <xs:attribute ref="createdOn"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="instance">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="assocclass" . . ./>
        <xs:element ref="attribute" . . ./>
      </xs:sequence>
      <xs:attribute ref="id" use="required"/>
      <xs:attribute ref="class"/>
    </xs:complexType>
  </xs:element>

. . .
```

Following is the snippet of the xml file produced by octopus which conforms to above schema file:

Output.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
- <storageRoot xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="file://c:/eclipse4DAwork/plugins/com.k
  lasse.octopus.codegen_2.0.0/octopus-storage.xsd"
  project="octoAppTester" createdOn=" . . .">
  - <instance id="id1" class="application.Application">
      <attribute name="name" value="sbs" />
          <attribute name="path" value="sbs" />
      <attribute name="namespace" value="sbs" />
      <attribute name="serviceNamespace" value="sbs" />
    </instance>
 . . .
     . . .
```

As this is octopus specific format can only be read by octopus, and for better integration with other tools with octopus it needs to produce a model based on a standard metamodel. Model based on some metamodel like this can be transformed to any other required format based on other metamodel or text based models (e.g. XML, HTML etc.). XML.ecore is a ecore metamodel for xml and can be seen as follows:

```
xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore">
 <ecore:EPackage name="PrimitiveTypes">
  <eClassifiers xsi:type="ecore:EDataType" name="Boolean"/>
  <eClassifiers xsi:type="ecore:EDataType" name="Integer"/>
  <eClassifiers xsi:type="ecore:EDataType" name="String"/>
 </ecore:EPackage>
 <ecore:EPackage name="XML">
  <eClassifiers xsi:type="ecore:EClass" name="Node" abstract="true">
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="name" ordered="false" lowe
   <eStructuralFeatures xsi:type="ecore:EAttribute" name="value" ordered="false" lowe
   <eStructuralFeatures xsi:type="ecore:EReference" name="parent" ordered="false" e
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Attribute" eSuperTypes="/1/Node"/>
  <eClassifiers xsi:type="ecore:EClass" name="TextNode" eSuperTypes="/1/Node"/>
  <eClassifiers xsi:type="ecore:EClass" name="Element" eSuperTypes="/1/Node">
   <eStructuralFeatures xsi:type="ecore:EReference" name="children" upperBound="-1"
   eOpposite="/1/Node/parent"/>
  </eClassifiers>
  <eClassifiers xsi:type="ecore:EClass" name="Root" eSuperTypes="/1/Element"/>
 </ecore:EPackage>
</xmi:XMI>
```

5.1.6 Integrating ATL with Octopus

The Atlas Transformation Language (ATL) is a hybrid language (a mix of declarative and imperative constructions) designed to express model transformations as required by the MDA. A transformation model in ATL is expressed as a set of transformation rules. The recommended style of programming is declarative. Transformations from Platform Independent Models (PIMs) to Platform Specific Models (PSMs) can be written in ATL to implement the MDA.

A prototype transformation engine, named ATL v0.1, has been developed to validate some of the ideas included in the language. every model and metamodel needed (typically: input model plus input and output meta-models) is read from its XMI / Ecore definition. The transformation is then executed, rule after rule and the resulting model is eventually serialized to an XMI / text file. ATL also support user to query a view over any model using special OCL Query module.

Eclipse is going to be used as an IDE for ATL, with advanced code edition features (syntax highlighting, auto-completion, etc.). ATL will provide a context in which transformation-based MDA tools can be designed and implemented for Eclipse.

Now we will see how can we use output from Octopus to transform into the desired PSM, which will be CORYX Platform specific xml based specification here, using ATL.

now the generated output.xml cant be used directly in ATL, because we need to specify xml metamodel in ecore or xmi form. And that would be XML.ecore as seen before.

So a desired example octopus output model for the metamodel XML.ecore would be as follows.

Desired example octopus output model

```
<?xml version="1.0" encoding="iso-8859-1"?>
<Root xmlns="XML" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi="http://www.w3.org/2
   <children xmlns="" xsi:type="Element" name="application.Application">
      <children xsi:type="Attribute" name="id" value="id1"/>
      <children xsi:type="Attribute" name="name" value="sbs"/>
      <children xsi:type="Attribute" name="path" value="sbs"/>
      <children xsi:type="Attribute" name="namespace" value="sbs"/>
      <children xsi:type="Attribute" name="serviceNamespace" value="sbs"/>
      <children xsi:type="Element" name="services">
            <children xsi:type="Attribute" name="ref-id" value="id21"/>
            <children xsi:type="Attribute" name="ref-id" value="id20"/>
      </children>
      <children xsi:type="Element" name="dataSources">
         <children xsi:type="Attribute" name="ref-id" value="id3"/>
      </children>
   </children>

   <children xsi:type="Element" name="refExtServices">
      <children xsi:type="Attribute" name="ref-id" value="id9"/>
      <children xsi:type="Attribute" name="ref-id" value="id10"/>
   </children>

   <children xsi:type="Element" name="extApplications">
      <children xsi:type="Attribute" name="ref-id" value="id1"/>
   </children>

   <children xsi:type="Element" name="dataSources">
      <children xsi:type="Attribute" name="ref-id" value="id6"/>
      <children xsi:type="Attribute" name="ref-id" value="id5"/>
      <children xsi:type="Attribute" name="ref-id" value="id4"/>
      <children xsi:type="Attribute" name="ref-id" value="id3"/>
   </children>

   <children xmlns="" xsi:type="Element" name="application.JdbcDataSource">
      <children xsi:type="Attribute" name="id" value="id3"/>
      <children xsi:type="Attribute" name="name" value="asd"/>
      <children xsi:type="Attribute" name="namespace" value="asd"/>
      <children xsi:type="Attribute" name="tableRegistry" value="asd"/>
      <children xsi:type="Element" name="persistableObjects">
         <children xsi:type="Attribute" name="ref-id" value="id7"/>
         <children xsi:type="Attribute" name="ref-id" value="id8"/>
      </children>
   </children>
```

The first dirty way to achieve this desired output is to directly modify the src/xmlstorage/..write.java and make it write a proper format as you require. But that will create a problem when you regenerate the java code, well then one will think I would better write my own writer and use that writer from the generated class, that would be slightly better but then when you change your .uml model in octopus and regenerate the java code your writer class has to be modified manually to match the new model. There are many ways to solve this problem, and I have one to show here:

Whatever uml model you use to generate java code in octopus, the resulted output stored by the octopus will always conform to octopus-storage.xsd, and you always need to convert that to match the general xml metamodel given above in the xmi. So a simple xslt transformation will do the job here, because we don't need any heavy

or complex transformations, we can use xslt for normal xml2xmi or xml2ecore conversion.

Partial listing of the xslt solution

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:xsi="ht
xmlns:xmi="http://www.omg.org/XMI" xmi:version="2.0">
<xsl:output method="xml" version="1.0" encoding="iso-8859-1" indent="yes"/>
  <xsl:template match="/">
   <Root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xmi="http://www.omg
   name="Root" value="">
      <xsl:apply-templates select="//instance"/>
   </Root>
  </xsl:template>

  <xsl:template match="instance">
     <children xsi:type="Element">
        <xsl:attribute name="name"><xsl:value-of select="@class"/></xsl:attribute>
        <children xsi:type="Attribute">
           <xsl:attribute name="name"> <xsl:text>id</xsl:text></xsl:attribute>
           <xsl:attribute name="value"><xsl:value-of select="@id"/></xsl:attribute>
        </children>
        <xsl:apply-templates/>
     </children>
  </xsl:template>

  <xsl:template match="attribute">
     <xsl:variable name="value" select="@value"/>
     <xsl:variable name="idref" select="@idref"/>
     <xsl:variable name="idrefs" select="@idrefs"/>

     <xsl:choose>
        <xsl:when test="$value != ''">
           <children xsi:type="Attribute">
              <xsl:attribute name="name"> <xsl:value-of select="@name"/> </xsl:attribute>
              <xsl:attribute name="value"><xsl:value-of select="@value"/></xsl:attribute>
           </children>
        </xsl:when>
        <xsl:when test="$idref != ''">
           <children xsi:type="Element">
              <xsl:attribute name="name"><xsl:value-of select="@name"/></xsl:attribute>
              <children xsi:type="Attribute">
                 <xsl:attribute name="name"> <xsl:text>ref-id</xsl:text></xsl:attribute>
                 <xsl:attribute name="value"><xsl:value-of select="@idref"/></xsl:attribute>
              </children>
           </children>
        </xsl:when>
        <xsl:when test="$idrefs != ''">
           <children xsi:type="Element"> <xsl:attribute name="name"><xsl:value-of select="@n
```

This will work to convert octopus output to desired .xmi or .ecore format which will be a valid model of the XML.ecore metamodel. This transformation enables octopus to integrate with any transformation tool easily and so eases octopus' further extensions and/or integration with other software frameworks.

Now to automize this conversion, when you are done with generation of java code in octopus just add a line of code in src/xmlstorage/..writer.java/write(..) as follows:

```
public void write(File output) {
   Map map = allocateIDs();
   List allElements = storeInstances(map);
   writeDoc(output, allElements);
   org.apache.xalan.xslt.Process.main(
      new String[]
         {"-IN",  "octopus out file name",
          "-XSL", "xsl file name",
          "-OUT", "desired out file name with extension .xmi/.ecore"});
}
```

or one can also do this conversion before applying ATL rules, so that one don't even need to change any auto generated file.

Now that we have xml model based on Ecore metamodel we can apply transformation rules using transformation language, I chose ATL for the purpose. One of the advanced feature from ATL will be used here: ATL Query to generate text.

5.1.7 ATL Queries and Generation of Text

Queries for service.xmls

```
query XML2Service = XML!Root->allInstances()->asSequence()->first().startTransform(0);

helper context XML!Root def: startTransform(i : Integer) : String =
    XML!Element->allInstances()->select(x | x.name='application.Application')
        ->collect(a | a.transformServices('services', 'octopus_storage\\apps\\'
            + a.getAttrValue('path') + '\\server\\service\\', 0));

helper context XML!Element def: transformServices(ref : String, path : String, i : Inte
    if self.exist(ref) then
        self.getChildren(ref)->first().getRefIds()->
            collect(rs | self.getElement(rs.getValue()).transformService(i)
                .writeTo(path + self.getElement(rs.getValue()).getAttrValue('location')
    else ''
    endif;

helper context XML!Element def: exist(element : String) : Boolean =
    self.getChildren(element)->size()>0;

helper context XML!Element def: getChildren(eName : String) : Sequence(XML!Element) =
    self.children->select(d | d.oclIsTypeOf(XML!Element))
                        ->select(e | e.name=eName)->asSequence();

helper context XML!Element def: getRefIds() : Sequence(XML!Attribute) =
    self.children->select(d | d.oclIsTypeOf(XML!Attribute))
                        ->select(a | a.name='ref-id');
```

Above is the ATL Query snippet which will generate set of service.xml which will be then integrated using application.xml which will be generated by another query; snippet of which can be seen as follows:

```
query XML2Application = XML!Root->allInstances()->asSequence()->first().startTransform(0);

helper context XML!Root def: startTransform(i : Integer) : String =
    XML!Element->allInstances()->select(x | x.name='application.Application')->collect(a |
        a.transformApplication(i).writeTo('octopus_storage\\apps\\' + a.getAttrValue('path')

helper context XML!Element def: exist(element : String) : Boolean =
    self.getChildren(element)->size()>0;
```

After generating xml specification files for CxPT now those can be directed to CxPT as input and then can be generated software artefacts as before.

The whole solution prototype can be visualized as follows:

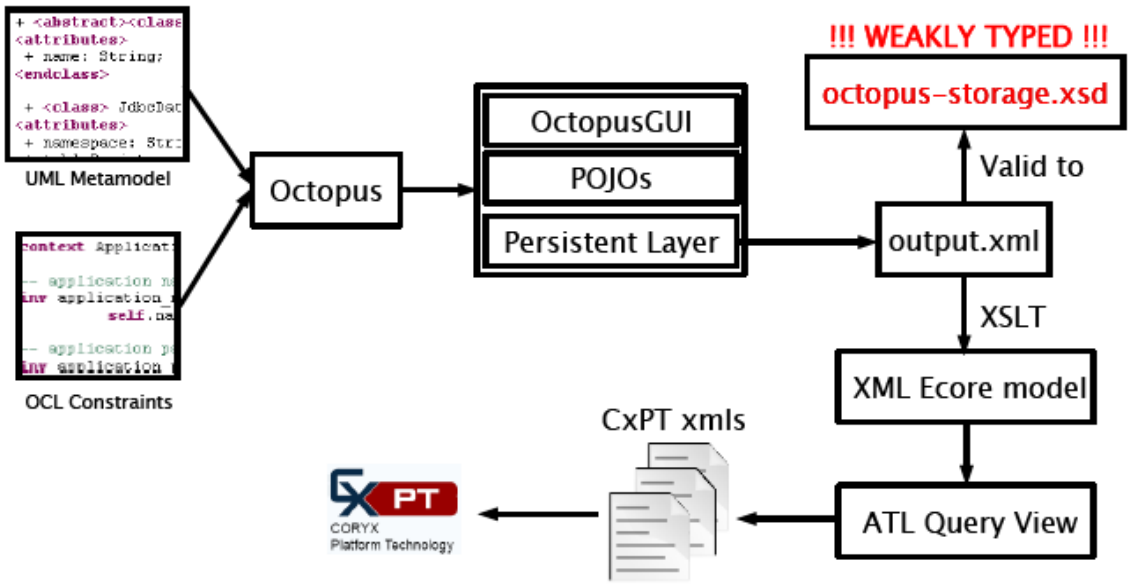

Figure 5.7 Extended CxPT Prototype

But as you can see Octopus persist xml instances based on octopus-storage.xsd which is weakly typed, meaning all instances are stored with same xml elements and it would be better if it could persist instances valid to schema which is derived from uml metamodel. XML instances then can also be checked for inconsistencies using Xqueries which could be derived from OCL constraints. Details for this approach to extend Octopus can be seen in following chapter.

6 Extending Octopus

6.1 OctopusUML to XSD transformation

The rubber meets the road when using UML in the development of XML schemas. A primary goal of this mapping is to allow sufficient flexibility to encompass most schema design requirements, while retaining a smooth transition from the conceptual vocabulary model to its detailed design and generation. A related goal is to allow a valid XML schema to be automatically generated from any UML class diagram [54,55,56,57], even if the modeler has no familiarity with the XML schema syntax. Having this ability enables a rapid development process and supports reuse of the model vocabularies in several different deployment languages or environments, because the core model is not overly specialized to XML structure.

The same mapping from UML to XML schema can be reversed to support reverse engineering XML schemas into UML. There are several different strategies for implementing this mapping into UML. For example, the model might reflect a hierarchical structure similar to the XML parent/child relationships, or the mapping might emphasize a conceptual, object-oriented structure in UML. The description on the mapping from UML to XML schema is not the intent of this document.

Another purpose of this mapping being implemented is to extend Octopus tool to serialize xml instances in strong typed way instead of weak typed as it is now. Strong typed and weak typed xml snippet can be seen as follows:

Weak typed XML instance

```
<instance id="001" type="Person">
   <element name="name" value="James"/>
   <element name="birth-city" value="Hamburg"/>
</instance>

<instance id="002" type="Person">
   <element name="name" value="Roger"/>
   <element name="birth-city" value="Coburg"/>
</instance>
```

Strong typed XML instance

```
<Person id="001">
   <name>James</name>
   <birth-city>Hamburg</birth-city>
</instance>

<Person id="002">
   <name>Roger</name>
   <birth-city>Coburg</birth-city>
</instance>
```

To achieve this strong typed support, XMLReader and XMLWriter will also be needed as a proof of concept to support new schema definition, implementation of which will be in the end of this chapter.

Octopus offers two important functionalities:

- Octopus is able to statically check OCL expressions. It checks the syntax, as well as the expression types, and the correct use of model elements like association roles and attributes.

- Octopus is able to transform the UML model, including the OCL expressions, into Java code.

Octopus is able to generate a complete 3-tier prototype application from your UML/OCL model.

- The middle tier consists of plain old Java objects (POJOs). These POJOs include code for checking invariants and multiplicities from the model.

- The storage tier consists of an XML reader and writer dedicated to your UML/OCL model. It stores any data content in your prototype application in an XML file, and can also read back in. All instances are stored in a weak typed way in the XML file for now.

- The user interface tier consists of an implementation of a plug-in for the Eclipse Rich Client Platform. From a Navigator view that shows you all instances in the system, you are able to create and examine instances of your UML/OCL model. Of course, the invariants or multiplicities of an instance can be checked by pushing a single button.

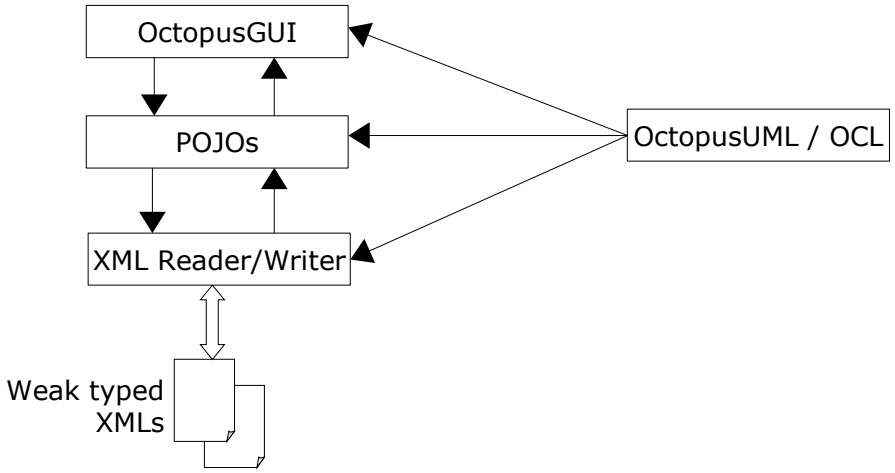

Figure 6.1 Existing Octopus generated artefacts using weak typed XMLs

The remaining of this chapter is about the details on extending Octopus to transform OctopusUML (.uml) to XML Schema XSD model, Which then can be extended further to transform OCL expressions to XQueries to make it complete. The complete extension can be visualized in figure 6.2.

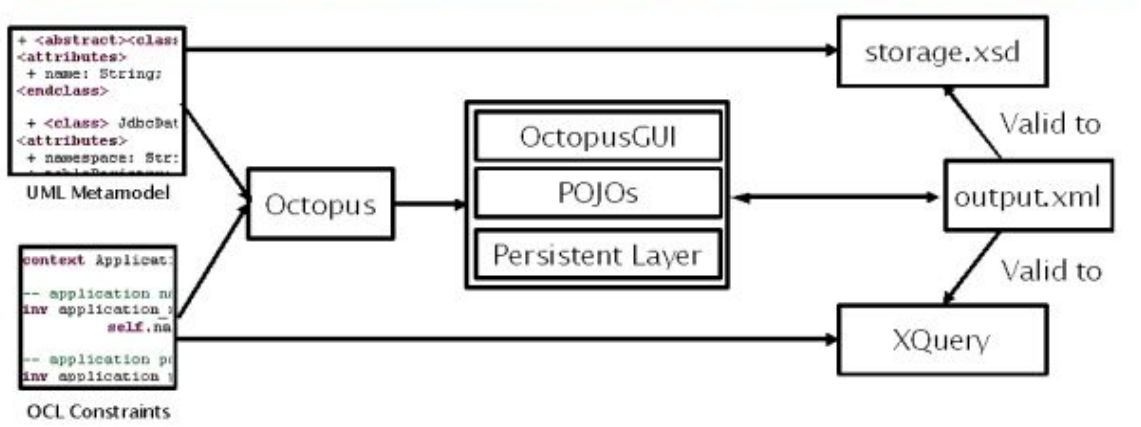

Figure 6.2 Extended Octopus with UML/OCL mapped to XSD/XQuery

6.1.1 Representing Associations

The most crucial issue of the transformation algorithm is the treatment of UML associations. There are different procedures how to represent associations in an XML Schema but all of them result in some loss of information regarding the source model. There are four approaches which are as following:

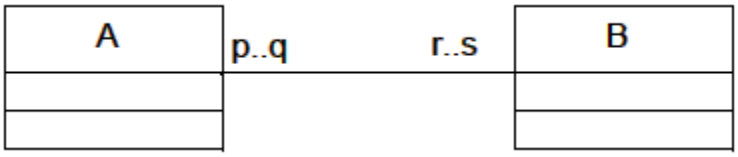

Figure 6.3 Example relation between two classes

- Nested elements (hierarchical relationship)
- Key/Keyref references of elements
- References via association element
- References with XLink and XPointer

Hierarchical relationship

The hierarchical relationship is the "natural" relationship in XML because it corresponds with the tree structure of XML documents. Elements are nested within their parent elements which implies some restrictions. The main obstacle for the nesting of elements is the creation of redundancies in case of many-to-many relationships. Regarding hierarchical representation it is also difficult to deal with recursive associations or relationship cycles between two or more classes. The XML documents have a document tree of indefinite depth.

Key/Keyref references

The Key/Keyref relationship is expressed by adding an ID attribute to referenceable elements and a key that contains a selector and a field which includes an XPath expression each. The selector selects all elements of a class and the field selects the ID attribute of each selected element. The references are implemented by reference elements with an attribute of type IDREF and a keyref (key reference). This keyref references the key of the target class. Additionally, the keyref selects all reference elements with the selector XPath expression and the field XPath expression selects the IDREF attribute of each selected reference element. So the schema validator compares the IDREF attribute of all the reference elements with the ID attribute of the target class element. This approach guarantees type safety.

References via association elements

For each association an association element is introduced that references both participating elements using IDREF attributes (analogous to relations for many-to-many relationships in RDBMS). The association elements are included as subelements of the document root. There are no references in the class elements. The association element gets the name of the association, the references are labeled according to the association roles. The approach produces XML documents with minimal redundancy because every instance needs to be stored only once within the document. The multiplicity values cannot be expressed adequately by association elements.

References with XLinks

XLinks have been invented for hyperlink documents that are referencing each other which makes it possible to reference different document fragments. We consider the extended features provided by XLinks. The association element is represented as *extended* link. A *locator* element is needed for each associated element to identify it. The association itself is established by *arc* elements that specify the direction. However this approach has no type safety.

6.1.2 Representing Association classes

An association class is an association with class features. So the transformation has to consider the mapping of both a class and an association. Therefore, the four mapping approaches for associations, as sketched above, apply to association classes as well: The association class is mapped to an association element that is nested inside the parent element in the hierarchical approach (for functional relationships only). The association attributes and the child element in the hierarchical approach are added to the association element.

Using Key/Keyref references requires the introduction of two references to consider bidirectional relationships. Thus the attributes of the association class would be stored twice. It could not be guaranteed that those attributes are the same in two mutually referencing elements. Hence, the mapping has to be enhanced by association elements. The association elements contain the attributes of the

corresponding association class. Associations of each multiplicity are dealt with the same way.

Regarding the last approach that uses extended XLinks is comparable with association elements one can draw the same conclusion as mentioned above.

It is also possible to resolve the association class and represent it as two separate associations. Note that the semantics of bidirectional associations cannot be preserved adequately with that mapping.

6.1.3 Associations - Limitations

Each end of an association can be assigned the *{ordered}* property to determine the order of the associated instances. It is not possible to define the order of element instances in an XML Schema.

The direction of an association cannot be preserved by mapping approaches that represent just bidirectional associations. This applies to: hierarchical relationships, association elements and extended XLinks.

6.1.4 Mapping of Generalization

There is no generalization construct in the XML Schema. The most relevant aspect of generalization is the inheritance of attributes of the superclass. There are two reasonable approaches to represent the inheritance in the XML Schema: the type inheritance by type extension and the reuse of element and attribute groups. This approach supports the substitution relationship between a superclass and its subclasses, but it supports only single inheritance.

Alternatively, an element and an attribute group can be defined for the subelements and attributes of each class element which can be reused in the complex type of the corresponding class element. Additionally, the element and attribute groups of all superclasses of a class are reused in the complex type of this class. So all elements and attributes of the superclasses are assigned to the subclasses. This approach supports multiple inheritance, but doesn't support the substitution relationship between a superclass and its subclasses. To express the substitution relationship between a superclass and its subclasses, the use of a superclass element is substituted by a choice list that contains the superclass element and all its subclass elements.

6.1.5 Further mapping issues

The *aggregation* relationship of UML embodies a simple part-of semantics whereas the existence of the part does not depend on the parent. Therefore aggregations are treated like associations. *Compositions* can be mapped through hierarchical relationships according to a previous proposal for associations, because nested elements are dependent on the existence of their parent elements and therefore represent the semantics of compositions.

6.1.6 Transformation rules

Transformation rules can be seen as follows:
Table 6.1 Transformation rules – OctopusUML to XML Schema

OctopusUML	***XML Schema***
Package: `<package>`	Root element, complex type without attributes consisting all types(e.g. Class, datatype, enumeration etc.); and name will be used as a namespace for children e.g. Package.class
Abstract types: `<abstract><class> / <interface>`	Abstract complex type, with ID attribute
Normal types: `<class> / <datatype> / <enumeration>`	Element, complex type (with extension if defined so,), with ID attribute, and in case of enumeration type will be xsd:enumeration
Extended types: `<implements> / <specializes>`	Complex type of the subclass is defined as an extension of the complex type of the superclass
Attributes: `<attributes>`	Optional subelement of the corresponding class complex type, derived attribute (/) not supported in the xml context, transform multiplicity in minoccurs and maxoccurs attributes
Operations: `<operations>`	<<information will be added as a comment>>
Visibility: `# + $ -`	<<information will be added as a comment>>
Behavioral states: `<states>`	<<loss of information>>
-	**-**
Associations / Aggregations: `<associations> / <aggregate>`	Reference element, with IDREF attribute referencing the associated class and keyref for type safety references
Composition: `<composite>`	Reference element, with subordinated class elements (hierarchical rel.), or reference element with dangling checks to impose cascade relationship
Qualified associations: `<ordered> / <notUnique>`	<<information will be added as a comment>>
Association Class: `<associationclass>`	Complex type with ID, and an additional IDREF references to the association class element and a keyref in the corresponding reference elements in the associated classes

For more clear idea of mapping between OctopusUML and XSD following example (based on Royal and Loyal class model) will be helpful.

Royal and Loyal Class model

An UML class diagram of Royal and Loyal can be seen in figure 6.4 and an OctopusUML of Royal and Loyal can be refered from appendix A.

Abstract Syntax Trees (ASTs) used in example are produced using debug mode of octopus' plug-ins for more clear view of the Octopus in-memory representation of object model.

Octopus has different visitor classes which will help to visit through OctopusUML model and OCL expressions. While visiting different artefacts one can also get branches related to that artefact from the in-memory AST. For transformation from OctopusUML model to XSD only packages and classifiers (classes, datatypes and enumerations) and interfaces are visited.

In this algorithm there will always be one root package, which will contain all other packages as elements in a complex type. This root package named 'model' is equivalent to the octopus hidden root package named '_system'.

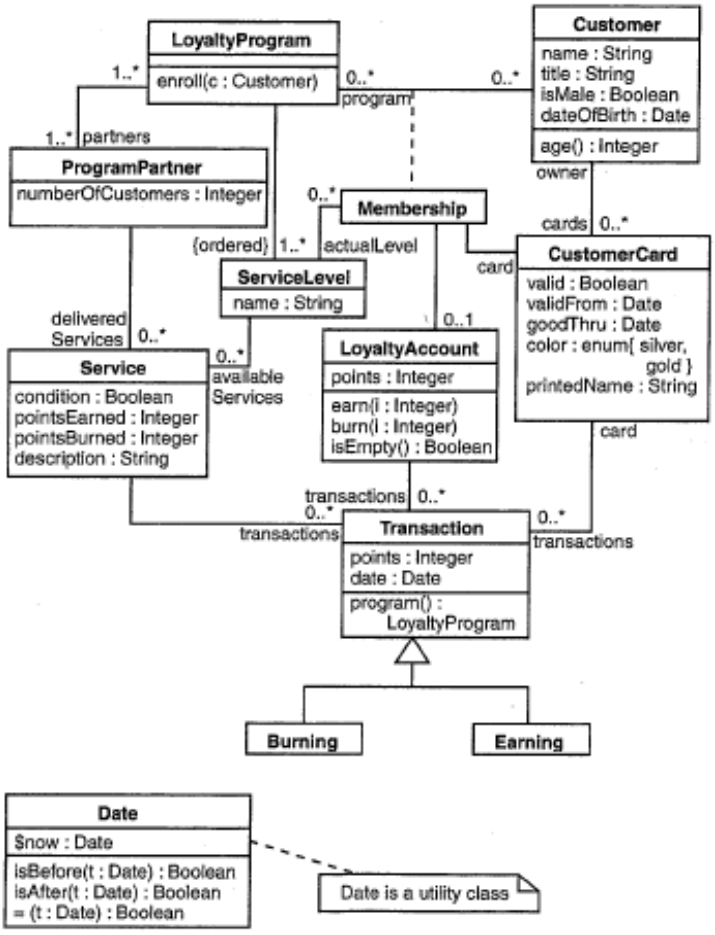

Figure 6.4 Royal and Loyal class diagram

Package

For each package, a complexType will be created having same name as package. This complexType will have a sequence of all classifiers and interfaces contained in that package including any subpackages as elements (same name as in uml model) of complexType (fully qualified name like in java). A complexType for each will be created when it will be visited. All elements here will be optional, subpackage elements can be allowed once at max and other elements are allowed with no max limit. For each classifier element (except enumeration) one key also be defined here with the fully qualified name and having key attribute 'id', which will be refered from keyRef definition using attribute 'idref'.

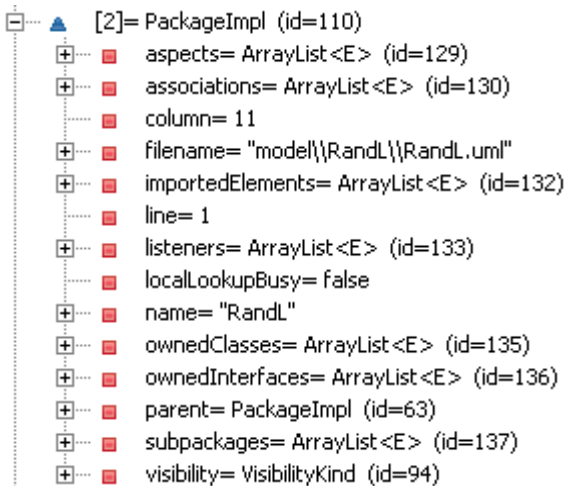

Figure 6.5 AST view of Package

Following is an example snippet of xml schema generated from the AST shown above.

```
- <xsd:complexType name="RandL">
  - <xsd:sequence>
    + <xsd:element name="Burning" type="RandL.Burning" minOccurs="0" maxOccurs="unbou
    + <xsd:element name="Customer" type="RandL.Customer" minOccurs="0" maxOccurs="un
    + <xsd:element name="LoyaltyProgram" type="RandL.LoyaltyProgram" minOccurs="0" m
    + <xsd:element name="ProgramPartner" type="RandL.ProgramPartner" minOccurs="0" m
    + <xsd:element name="Membership" type="RandL.Membership" minOccurs="0" maxOccur
    + <xsd:element name="Service" type="RandL.Service" minOccurs="0" maxOccurs="unbou
    + <xsd:element name="ServiceLevel" type="RandL.ServiceLevel" minOccurs="0" maxOccu
    + <xsd:element name="Transaction" type="RandL.Transaction" minOccurs="0" maxOccurs
    + <xsd:element name="CustomerCard" type="RandL.CustomerCard" minOccurs="0" maxO
    + <xsd:element name="LoyaltyAccount" type="RandL.LoyaltyAccount" minOccurs="0" ma
    + <xsd:element name="Earning" type="RandL.Earning" minOccurs="0" maxOccurs="unbou
    + <xsd:element name="Date" type="RandL.Date" minOccurs="0" maxOccurs="unbounded"
      <xsd:element name="RandLColor" type="RandL.RandLColor" minOccurs="0" maxOccurs=
      <xsd:element name="Gender" type="RandL.Gender" minOccurs="0" maxOccurs="unbour
    + <xsd:element name="TransactionReport" type="RandL.TransactionReport" minOccurs=
    + <xsd:element name="TransactionReportLine" type="RandL.TransactionReportLine" mi
    </xsd:sequence>
  </xsd:complexType>
```

Class, attributes and navigable associations

For each classifier, complex type will be created with a fully qualified name of the classifier to avoid name conflicts. Here we see how class will be transformed. Interfaces will be treated same as an abstract classes.

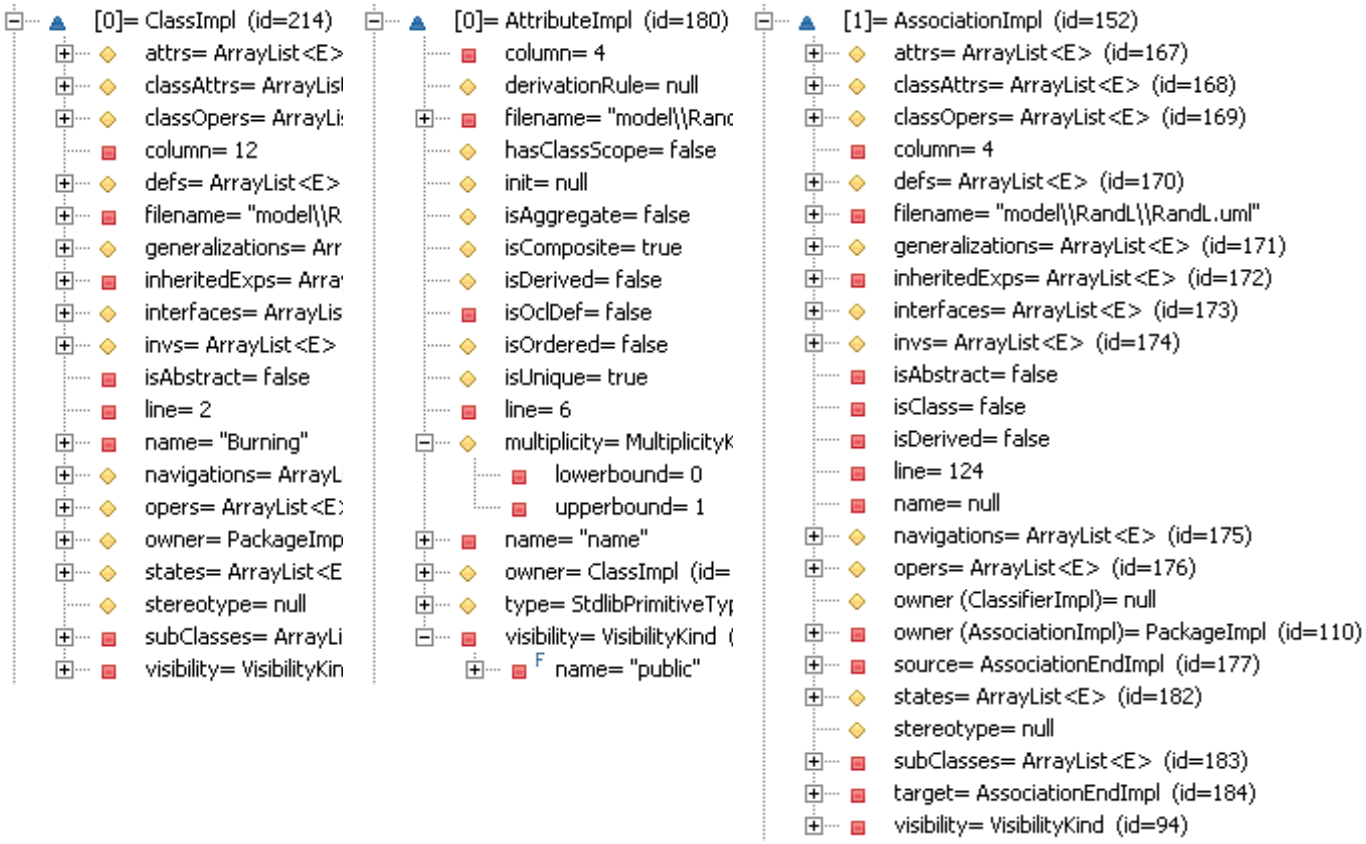

Figure 6.6 AST view of Class, Attribute and Association

While visiting classes we also visit contained attributes, operations and navigable associations (navigations). Operations will be ignored in the context of XSD and will be added as XML comments. Attributes will be added as subelements (same name as in uml model) in a single complexType/sequence. By default all these elements will be kept as optional and allowed only once at max, otherwise if specified multiplicityKind will be used to set lowerBound and upperBound of element occurance. Visibility of attribute will be ignored in the context of XSD. Actually here only non-transient attributes should be serialized but as there is no way of defining attributes transient in octopus, all attributes will be treated as non-transient. Primitive type of the attributes will be transformed to an XSD primitive type, e.g. String to xsd:string, boolean to xsd:boolean etc., all other types will be treated as complexType and assumed to be declared in the given uml model. Collection types like Set and Sequence will be treated as complexType of collection item type but with according upperBound. Aggregation and composition will be treated same here. So dangling artefacts then can be removed or ignored using appropriate XQuery.

Then all navigations will be added as an element with an attribute of type 'xsd:IDREF' and appropriate keyRef will be defined with selectors and field. Multiplicity will be used to set lowerBound and upperBound of element occurance.

For all abstract classes and interfaces one more attribute will be added in the complexType: `abstract="true"`.

Following is the XSD snippet of the AST of class, attribute and association shown in figure 6.6.

```
- <xsd:complexType name="RandL.Transaction" abstract="true">
  - <xsd:sequence>
      <!-- attribute : + points : Integer  -->
      <xsd:element name="points" type="xsd:integer" minOccurs="0" maxOccurs="1" />
      <!-- attribute : + date : Date  -->
      <xsd:element name="date" type="RandL.Transaction.date" minOccurs="0" maxOccurs="1" />
      <!-- attribute : + amount : Real  -->
      <xsd:element name="amount" type="xsd:real" minOccurs="0" maxOccurs="1" />
      <!-- method : + program() : LoyaltyProgram  -->
      <!-- AssociationEnd : + account : LoyaltyAccount  -->
    - <xsd:element name="account" minOccurs="1" maxOccurs="1">
      - <xsd:complexType>
          <xsd:sequence />
          <xsd:attribute name="idref" type="xsd:IDREF" use="required" />
        </xsd:complexType>
      - <xsd:keyref name="RandL.Transaction.account" refer="RandL.LoyaltyAccount">
          <xsd:selector xpath="." />
          <xsd:field xpath="@idref" />
        </xsd:keyref>
      </xsd:element>
      <!-- AssociationEnd : + generatedBy : Service  -->
    + <xsd:element name="generatedBy" minOccurs="1" maxOccurs="1">
      <!-- AssociationEnd : + card : CustomerCard  -->
    + <xsd:element name="card" minOccurs="1" maxOccurs="1">
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="required" />
  </xsd:complexType>
```

Following is an XSD snippet for an extended class. It should be noted here that multiple inheritance is not supported and interfaces will be treated as abstract classes.

```
- <xsd:complexType name="RandL">
  - <xsd:sequence>
    - <xsd:element name="Burning" type="RandL.Burning" minOccurs="0" maxOccurs="unbounded">
      - <xsd:key name="RandL.Burning">
          <xsd:selector xpath="RandL/Burning|.//Burning" />
          <xsd:field xpath="@id" />
        </xsd:key>
      </xsd:element>
- <xsd:element name="Transaction" type="RandL.Transaction" minOccurs="0" maxOccurs="unbounded">
  - <xsd:key name="RandL.Transaction">
      <xsd:selector xpath="RandL/Transaction|.//Transaction|.//Burning|.//Earning" />
      <xsd:field xpath="@id" />
    </xsd:key>
  </xsd:element>
```

Note here that there is no key attribute 'id' defined for this complexType because extended types will always inherit all fields from the base type and so key attribute 'id' also been inherited and used.

It has been said that key will be defined while visiting package, for each classifier. While defining key we also need to specify that how it will be searched using appropriate XPath in selector. Note below that how selector includes inherited types also in the base types.

Enumeration

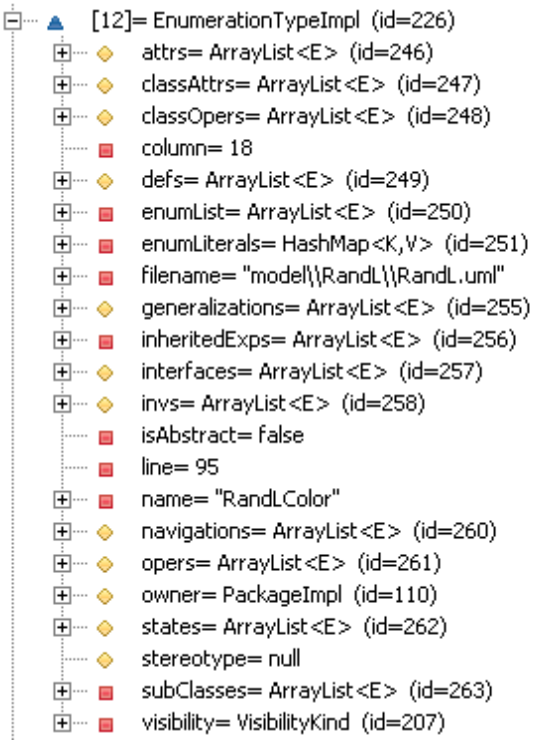

Figure 6.7 AST view of Enumeration

Even though enumeration is treated as classifier in octopus AST, it will be treated differently in the context of XSD. For each enumeration classifier simpleType (with fully qualified name of classifier) with restriction of string values will be created. String values will be retrieved using enumLiterals from the AST. All other transformation details will be same as other classifiers as before.

The transformed snippet from the XSD can be seen as follows :

```
- <xsd:simpleType name="RandL.RandLColor">
  - <xsd:restriction base="xsd:string">
      <xsd:enumeration value="silver" />
      <xsd:enumeration value="gold" />
    </xsd:restriction>
  </xsd:simpleType>
```

Association class

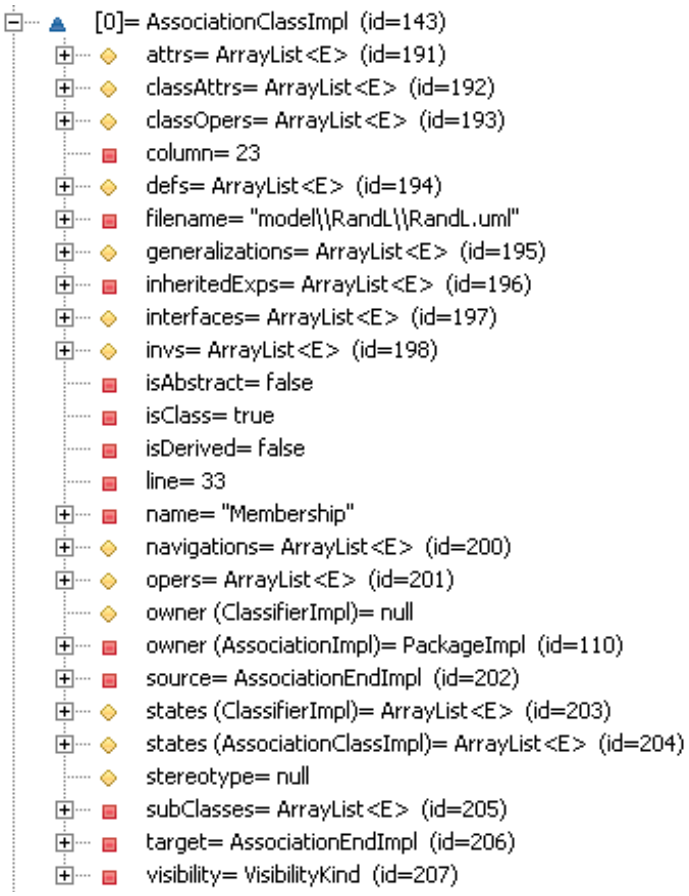

Figure 6.8 AST view of Association Class

Association classes are treated differently in the XSD context. For each association class complexType (with the fully qualified name) will be created with all attributes as subelements as in other classifiers. Additionally here two keyRef will be added each for each associated end. Here it should be noted that associated class can only refer two ends at most and those are mandatory references. Transformed snippet of XSD can be referred as follows:

```
- <xsd:complexType name="RandL.Membership">
  - <xsd:sequence>
      <!-- AssociationEnd : + currentLevel : ServiceLevel  -->
    + <xsd:element name="currentLevel" minOccurs="1" maxOccurs="1">
      <!-- AssociationEnd : + account : LoyaltyAccount  -->
    + <xsd:element name="account" minOccurs="0" maxOccurs="1">
      <!-- AssociationEnd : + card : CustomerCard  -->
    + <xsd:element name="card" minOccurs="1" maxOccurs="1">
      <!-- AssociationEnd : + participants : OrderedSet(Customer)  -->
    + <xsd:element name="Customer" minOccurs="1" maxOccurs="1">
      <!-- AssociationEnd : + programs : Set(LoyaltyProgram)  -->
    + <xsd:element name="LoyaltyProgram" minOccurs="1" maxOccurs="1">
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="required" />
  </xsd:complexType>
```

It should also be noted the way association class being referred by associated classes. Associated classes will have keyRef to other associated class and the associationclass, to avoid name conflicts attribute name will be same as name of associationclass instead of normal 'idref'. Example snippet can be seen as follows:

```
- <xsd:complexType name="RandL.LoyaltyProgram">
  - <xsd:sequence>
      <!-- attribute : + name : String  -->
      <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs="1" />
      <!-- method : + enroll( c: Customer )  -->
      <!-- method : + getServices() : Set(Service)  -->
      <!-- method : + getServices( pp: ProgramPartner ) : Set(Service)  -->
      <!-- method : + addTransaction( accNr: Integer, pName: String, servId: Integer
      <!-- method : + selectPopularPartners( d: Date ) : Set(ProgramPartner)  -->
      <!-- method : + addService( p: ProgramPartner, l: ServiceLevel, s: Service )
      <!-- method : + enrollAndCreateCustomer( n: String, d: Date ) : Customer  -->
      <!-- AssociationEnd : + participants : OrderedSet(Customer)  -->
    - <xsd:element name="participants" minOccurs="0" maxOccurs="unbounded">
      - <xsd:complexType>
          <xsd:sequence />
          <xsd:attribute name="idref" type="xsd:IDREF" use="required" />
          <!-- participants <-> programs  -->
          <xsd:attribute name="Membership" type="xsd:IDREF" use="required" />
        </xsd:complexType>
      + <xsd:keyref name="RandL.LoyaltyProgram.participants" refer="RandL.Customer">
      + <xsd:keyref name="RandL.Customer.Membership" refer="RandL.Membership">
      </xsd:element>
      <!-- AssociationEnd : + partners : Set(ProgramPartner)  -->
    + <xsd:element name="partners" minOccurs="1" maxOccurs="unbounded">
      <!-- AssociationEnd : + levels : OrderedSet(ServiceLevel)  -->
    + <xsd:element name="levels" minOccurs="1" maxOccurs="unbounded">
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID" use="required" />
  </xsd:complexType>
```

6.2 Proof of Concept

Now as we have seen how OctopusUML to XSD transformation rules are implemented, we need a proof of concept that octopus can still support reading and writing of XML instances valid to newly generated strong typed XSD schema. To understand how new reading and writing capability is implemented in the original structure shown in figure 6.1, we first need to understand two important things which are Java reflection API and Java XML Binding.

6.2.1 Java Reflection API

Reflection enables Java code to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts on objects, within security restrictions. The API accommodates applications that need access to either the public members of a target object (based on its runtime class) or the members declared by a given class.

Beginning with J2SDK 1.4.0, certain reflective operations, specifically `java.lang.reflect.Field`, `java.lang.reflect.Method.invoke()`, `java.lang.reflect.Constructor.newInstance()`, and `Class.newInstance()`, have been rewritten for higher performance. Reflective invocations and instantiations are several times faster than in previous releases.

6.2.2 Java XML Binding (JAXB)

JAXB simplifies access to an XML document from a Java program by presenting the XML document to the program in a Java format. The first step in this process is to bind the schema for the XML document into a set of Java classes that represents the schema. JAXB also supports unmarshalling and marshalling of XML instances.

Unmarshalling an XML document means creating a tree of content objects that represents the content and organization of the XML document. The content tree is not a DOM-based tree. In fact, content trees produced through JAXB can be more efficient in terms of memory use than DOM-based trees. The content objects are instances of the classes produced by the binding compiler. In addition to providing a binding compiler, a JAXB implementation must provide runtime APIs for JAXB-related operations such as marshalling, means creating a XML document that represents the objects. The APIs are provided as part of a binding framework.

6.2.3 Extended Octopus using strong typed XML reader/writer

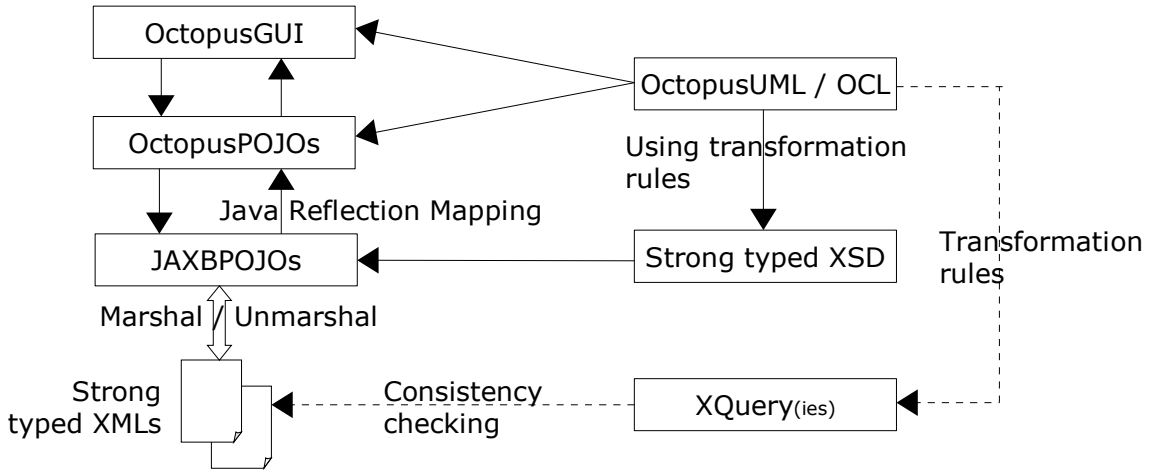

Figure 6.9 Extended Octopus using strong typed XML reader/writer

Please compare figure 6.1 and figure 6.9 to have more clear idea about what changes are made to support strong typed XML reading and writing capability. In the new architecture Octopus generated XML reader/write are not used as they support only weak typed XML instances. OctopusGUI and Octopus generated POJOs (in the figure 6.9 – OctopusPOJOs) are used as they were. Strong typed XSD is transformed from OctopusUML and then from that XSD JAXBinding classes (here refered as JAXBPOJOs) are generated which will be used to marshal and unmarshal the strong typed XMLs. Now as OctopusPOJOs are tightly coupled with OctopusGUI to support JAXBPOJOs with the OctopusGUI we need a mapping between OctopusPOJOs and JAXBPOJOs. That mapping bridge here is implemented using Java Reflection APIs. As this mapping is done using the generation pattern of Octopus and JAXB we don't need to generate classes for mapping everytime we change our metamodel, OctopusUML in this case. The part of the figure connected with dotted lines is not implemented with this work but can be implemented as shown to support external consistency checking using XQuery.

Appendix A

OctopusUML For Royal and Loyal

```
<package> RandL

 + <class> Burning <specializes> Transaction
<endclass>

 + <class> Customer
<attributes>
 + name: String;
 + title: String;
 + isMale: Boolean;
 + dateOfBirth: Date;
 + /age: Integer;
 + gender: Gender;
<operations>
 + age(): Integer;
 + birthdayHappens();
<endclass>

 + <class> LoyaltyProgram
<attributes>
 + name: String;
<operations>
 + enroll(<inout> c: Customer);
 + getServices(): Set_Service;
 + getServices(<inout> pp: ProgramPartner): Set_Service;
 + addTransaction(<inout> accNr: Integer, <inout> pName: String, <inout>
servId: Integer, <inout> amnt: Real, <inout> d: Date);
 + selectPopularPartners(<inout> d: Date): Set_ProgramPartner;
 + addService(<inout> p: ProgramPartner, <inout> l: ServiceLevel,
<inout> s: Service);
 + enrollAndCreateCustomer(<inout> n: String, <inout> d: Date):
Customer;
<endclass>

 + <class> ProgramPartner
<attributes>
 + numberOfCustomers: Integer;
 + name: String;
<endclass>

 + <associationclass> Membership
 + Customer.participants [0..*]  <ordered>    <->  +
LoyaltyProgram.programs [0..*]
<endassociationclass>

 + <class> Service
<attributes>
 + condition: Boolean;
 + pointsEarned: Integer;
 + pointsBurned: Integer;
 + description: String;
 + serviceNr: Integer;
```

```
<operations>
 + upgradePointsEarned(<inout> amount: Integer);
 + calcPoints(): Integer;
<endclass>

 + <class> ServiceLevel
<attributes>
 + name: String;
<endclass>

 +  <abstract> <class> Transaction
<attributes>
 + points: Integer;
 + date: Date;
 + amount: Real;
<operations>
 + program(): LoyaltyProgram;
<endclass>

 + <class> CustomerCard
<attributes>
 + valid: Boolean;
 + validFrom: Date;
 + goodThru: Date;
 + color: RandLColor;
 + /printedName: String;
 + myLevel: ServiceLevel;
<operations>
 + getTransactions(<inout> from: Date, <inout> until: Date):
Set_Transaction;
<endclass>

 + <class> LoyaltyAccount
<attributes>
 + number: Integer;
 + points: Integer;
 + totalPointsEarned: Integer;
<operations>
 + earn(<inout> i: Integer);
 + burn(<inout> i: Integer);
 + isEmpty(): Boolean;
 + getCustomerName(): String;
<endclass>

 + <class> Earning <specializes> Transaction
<endclass>

 + <class> Date
<attributes>
 +  $ now: Date;
 + year: Integer;
 + month: Integer;
 + day: Integer;
<operations>
 + isBefore(<inout> t: Date): Boolean;
 + isAfter(<inout> t: Date): Boolean;
 + <infix> = (<inout> t: Date): Boolean;
 + fromYMD(<inout> y: Integer, <inout> m: Integer, <inout> k: Integer):
Date;
```

```
<endclass>

 + <enumeration> RandLColor
<values>
silver;
gold;
<endenumeration>

 + <enumeration> Gender
<values>
male;
female;
<endenumeration>

 + <class> TransactionReport
<attributes>
 + from: Date;
 + until: Date;
 + /name: String;
 + /balance: Integer;
 + /number: Integer;
 + /totalEarned: Integer;
 + /totalBurned: Integer;
<endclass>

 + <class> TransactionReportLine
<attributes>
 + /partnerName: String;
 + /serviceDesc: String;
 + /amount: Real;
 + /points: Integer;
 + /date: Date;
<endclass>

<associations>
 + CustomerCard.cards [0..*]<-> + Customer.owner [1..1];
 + ServiceLevel.currentLevel [1..1]<-> + Membership.<noName> [0..*];
 + ProgramPartner.partners [1..*]<-> + LoyaltyProgram.programs [1..*];
 + Service.deliveredServices [0..*]<-> + ProgramPartner.partner [1..1];
 + LoyaltyAccount.account [0..1]<-> + Membership.<noName> [1..1];
 + CustomerCard.card [1..1]<-> + Membership.<noName> [1..1];
 + Transaction.transactions [0..*]<-> + LoyaltyAccount.account [1..1];
 + Service.generatedBy [1..1]<-> + Transaction.transactions [0..*];
 + Transaction.transactions [0..*]<-> + CustomerCard.card [1..1];
 + Service.availableServices [0..*]<-> + ServiceLevel.level [1..1];
 + ServiceLevel.levels [1..*]<ordered><-> +LoyaltyProgram.program[1..1];
 + Service.usedServices [0..*]<- + LoyaltyAccount.<noName> [0..*];
 + TransactionReportLine.lines[0..*]<-> +TransactionReport.report[1..1];
 + TransactionReport.<noName> [0..*]-> + CustomerCard.card [1..1];
 + Transaction.transaction[1..1]<-+TransactionReportLine.<noName>[0..*];
<endpackage>
```

Bibliography

[1] Model Driven Architecture: An Introduction, Richard Mark Soley, OMG Chairman and CEO, 2001

[2] Anneke Kleppe, Jos Warmer, and Wim Bast, MDA Explained; The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003.

[3] http://www.sdmagazine.com/documents/s=826/sdm0406e/ (browsed on 7.09.2005)

[4] Object Management Group, MOF 2.0 Query / Views / Transformations RFP, OMG Document: ad/2002-04-10, revised on April 24, 2002

[5] Java Metadata Interface 1.0, July 2002, http://java.sun.com/products/jmi

[6] ArchitectureWare, Generator Framework, http://www.architectureware.de

[7] OptimalJ 3.0, User's Guide, 2003,
http://www.compuware.com/products/optimalj/default.htm

[8] Jamda: The Java Model Driven Architecture 0.2, May 2003,
http://sourceforge.net/projects/jamda/

[9] Strategies for Program Transformation, http://www.stratego-language.org

[10] The Unified Modeling Language 1.5, March 2003, http://www.omg.org/cgibin/doc?formal/03-03-01

[11]Velocity 1.3.1, The Apache Jakarta Project, March 2003,
http://jakarta.apache.org/velocity/

[12]Rational XDE, http://www.rational.com/products/xde

[13] OMG, OMG-XML Metadata Interchange (XMI) Specification, v1.2, OMG
Document formal/02-01-01, http://www.omg.org/cgi-bin/doc?formal/2002-01-01

[14] XML Path Language Version 1.0, November 1999, http://www.w3.org/TR/xpath

[15] W3C, XSL Transformations (XSLT) Version 1.0, November 1999,
http://www.w3.org/TR/xslt

[16] XML-based Variant Configuration Language, http://fxvcl.sourceforge.net/

[17] Classification of Model Transformation Approaches
by Krzysztof Czarnecki and Simon Helson
http://www.swen.uwaterloo.ca/~kczarnec/ECE750T7/czarnecki_helsen.pdf

[18] Dr. Jon Whittle, Templates & Transformations : http://www.sse.cs.tu-bs.de/teaching/ss05/transformations/docs/Templates.ppt

[19] Dr. Jon Whittle, Graph Transformations : http://www.sse.cs.tu-bs.de/teaching/ss05/transformations/docs/GraphTransformations.ppt

[20] J. B´ezivin, F. Jouault, and P. Valduriez. First Experiments with a ModelWeaver. In *Proceedings of OOPSLA & GPCE Workshop*, October 2004.

[21] MOFScript, http://www.modelbased.net/mofscript/index.html

[22] MTF, http://www.alphaworks.ibm.com/tech/mtf

[23] ATL, http://www.sciences.univ-nantes.fr/lina/atl/

[24] ModelWare, http://www.modelware-ist.org/

[25] UMT, http://umt-qvt.sourceforge.net/

[26] MTL, http://modelware.inria.fr/

[27] MDR, http://www.netbeans.org/mdr

[28] EMF, http://www.eclipse.org/emf

[29] ModFact, http://modfact.lip6.fr/ModFactWeb/index.jsp

[30] GMT, http://www.eclipse.org/gmt

[31] KMF, http://www.cs.kent.ac.uk/projects/kmf/index.html

[32] OpenMDX, http://www.openmdx.org/index.html

[33] AndroMDA, http://www.andromda.org/

[34] XDoclet, http://www.xdoclet.org/

[35] Middlegen, http://boss.bekk.no/boss/middlegen/index.html

[36] OMELET, http://www.eclipse.org/omelet

[37] ArcStyler, http://www.io-software.com/

[38] MCC, http://www.inferdata.com/products/mcc/mdac.html

[39] Codagen Architect, http://www.codagen.com/

[40] OptimalJ, http://www.compuware.com/products/optimalj/default.htm

[41] Xactium XMF, http://www.xactium.com/

[42] SosyInc, http://www.sosyinc.com/

[43] Model-in-Actioin, http://www.mia-software.com/

[44] Frame-Processing-Language, http://sourceforge.net/projects/fpl

[45] Frame Processor ANGIE, Delta Software Technology, http://www.d-s-t-g.com/neu/pages/pageseng/et/common/techn_angie_frmset.htm

[46] FUUT-je, hosted at the Eclipse Generative Model Transformer (GMT) project website, http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmthome/download/index.html

[47] ATOM3: A Tool for Multi-Paradigm modeling, http://atom3.cs.mcgill.ca/

[48] A. Agrawal, G. Karsai and F. Shi. Graph Transformations on Domain-Specific Models. Under consideration for publication in the Journal on Software and Systems Modeling, 2003

[49] A. Rensink (Ed.) Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, University of Twente, Enschede, The Netherlands, June 26-27, 2003, CTIT Technical Report TR–CTIT–03–27, University of Twente, 2003, http://trese.cs.utwente.nl/mdafa2003

[50] E. D. Willink. UMLX: A graphical transformation language for MDA. In [49], pp. 13-24

[51] OMG, The Object Constraint Language Specification 2.0, OMG Document: ad/03-01-07

[52] Alcatel, Softeam, Thales, TNI-Valiosys, Codagen Corporation, et al. MOF Query/Views/Transformations, Revised Submission. OMG Document: ad/03-08-05

[53] F. Marschall and P. Braun. Model Transformations for the MDA with BOTL. In [49], pp. 25-36

[54] Rule-Based Generation of XML Schemas from UML Class Diagrams
by Tobias Krumbein, Thomas Kudrass; Leipzig University of Applied Sciences

[55] Conceptual Modeling XML Schemata Using UML
by Rainer Eckstein, Humboldt University zu Berlin; Silke Eckstein, Technical University Braunschweig;

[56] Towards A Framework for Mapping Between UML/OCL and XML/XQuery
by Sherif Sakr, University of Konstanz; Ahmed Gaafar, Cairo University;

[57] Practical Usage of W3C's XML-Schema and a Process for Generating Schema Structures from UML Models
by Mario C. Jeckle, DaimlerChrysler Research and Technology, Ulm, Germany