

# Component development: MDA based transformation from eODL to CIDL

Harald Böhme, Glenn Schütze, and Konrad Voigt

Humboldt-Universität zu Berlin

(boehme|schuetze|kvoigt)@informatik.hu-berlin.de

<http://www.informatik.hu-berlin.de/~boehme/~schuetze/~kvoigt>

**Abstract.** The development of software systems in general and software components in particular becomes a more and more challenging task. The key solution for handling the complexity in the development process is modeling of software systems and the transformation into implementation. The authors show an application of OMG's Model Driven Architecture (MDA) in the context of component development, where different languages as eODL, SDL, CIDL and C++ are involved. The application of model transformation is based on eODL as platform independent modeling language and CIDL as the platform specific modeling language. We used type based mapping rules to define the transformation. The paper shows the concrete implementation of these rules based on MOF repositories as model storage and the use of Java to perform the transformation actions. The Java technology Meta Data Repository (MDR) builds the base for an on-demand MOF repository creation in our approach. The handling of syntax based language is considered for integration purposes.

## 1 Introduction

The problem domain for today's software systems has broadened the scope in two dimensions. Software systems have to bridge different business areas and problems in single business fields become more complex. To cope with these new challenges different approaches were developed. Two of them are in the scope of this paper.

The component based software development is an approach for modular software development. It extends the concept of components from design and implementation, where modules are well known, to the binary software. The composition of components takes place during execution time.

Model Driven Architecture (MDA) as an initiative of the Object Management Group (OMG) centers the software development process on the models and tackle different problems of traditional software development processes. Starting point for activities in MDA are the models. This ensures that the models are always up-to-date; back propagation from implementation change to the design model is build-in.

With the technical specification for model storage an access MOF builds a good base for model transformation. This enables the automatic transformation of the design models into implementation (models). The gap between design and implementation known from traditional software development does not appear here.

But the successful development of new technologies can't be independent from existing tools and methods. So the step by step integration of MDA based methods is necessary. Conventional development tools are syntax based and therefore the integration has to be at syntax level. This means new tools have to start with syntactical representations of models or have to finish with it.

Therefore we present an example for a MDA based transformation. In the context of component based distributed software systems the CORBA Component Model (CCM) defined by the OMG is a concrete technology for distributed software systems and can be used as language for PSM. For the modeling of this kind of systems at abstract, platform independent level (PIM) this paper will use the language eODL defined by the ITU-T.

We will show the theoretical basis and appropriate languages/tools for an implementation of this kind of transformations. In the following section transformation rules at conceptual level are introduced by an example. Section 3 gives a detailed view on MOF and model repositories usage in Java. As stated the relation between syntax representation of models and repositories has to be defined. This is done in section 4. Based on the model repositories section 5 present a way how transformation rules can be implemented.

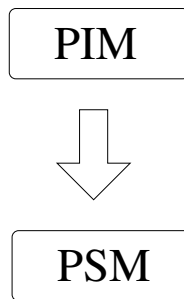
## 2 Transformation rules – from eODL to CIDL

The complexity of developing large distributed software systems can be reduced by usage of component technology and the MDA approach. With the usage of component technology the whole system is divided into parts, which reduces the complexity. Specifying and implementing the different parts becomes easier as the whole system at once. But the overall system behavior is a fixed constant in this process. It can be reduced by reusing software in terms of pre-developed software components. For doing so MDA is not needed. The MDA approach introduces a new dimension in software reuse. With component technology alone we are stuck to a concrete version/technology of software components in reuse. But as we all know technology is changing over the time. This leads to the situation, that at some point in future your software system consisting out of a set of software components implemented in a certain technology fits not to the current applied technology. Without MDA you now stop reusing your software components and are forced to redevelop the components. If we apply MDA here the transformation from the platform independent model/ design to the new component technology/ platform has to be defined only.

## 2.1 Model Driven Architecture (MDA)

The MDA is a new software engineering approach developed and published by the Object Management Group (OMG). One fundamental observation in the evolution of living software systems over the years is that their basic design models are mostly unchanged. Most changes to evolving software systems take place only at engineering level, forced by the introduction of new technologies and platforms.

MDA promotes simply the usage of models for the whole software system development. To capture the problem of technology evolution MDA defines two classes of models. The first one is for abstract modeling of the software systems at the design level. This model class is called Platform Independent Model (PIM). The second class is related to specific platforms and/or technologies. It contains mainly engineering aspects of the software system and is called Platform Specific Model (PSM). Between these two classes of models MDA defines a relation in form of several transformations, which insure the structural equivalence of PIM and PSM (see Fig. 1).



**Fig. 1.** Relation between PIM and PSM

There are three basic kinds of transformation specification:

- *Type based transformation:* Rules for transformation define a relation from concepts in the source Meta-Model to concepts in the target Meta-Model.
- *Instance based transformation:* Rules for transformation are defined on instances in the actual models, additional information for instance selection is needed.
- *Pattern based transformation:* Rules for transformation define a relation between patterns of instances in source and target model. Concepts of the Meta-Models can be used to formulate patterns.

Another key issue of MDA is a technology framework for different kinds of model handling (storage, exchange, mapping of models, etc.). The Meta Object Facility [4] is convenient for this purpose. Historically modeling languages are defined by abstract grammars. MOF instead defines modeling languages on the

base of so called Meta-Models. Meta-Models are models (instances) of built-in MOF concepts. Using this framework the developer can focus more on the definition of mappings between models rather than having to struggle with ordinary model handling. This is due to the fact that MOF comes with a method for the definition of model classes (Meta-Models) and for the exchange of models using the XML Metadata Interchange (XMI) [3]. In addition, MOF provides mappings of Meta-Models to repository interfaces as well. Such a repository holds all necessary information about model instances.

The above argumentation is correct for most of today's component technology. To show the real application we have to choose concrete Meta-Models for PIM and PSM. This also leads to the selection of appropriate Meta-Models and notations for PIM and PSM. One requirement for both is the support of the component concept as a first class concept. Moreover the Meta-Model for the PSM should be part of a well defined and established component technology. Because spreading out industrial usage is a task consuming several years the suitable technologies have traditional syntax based languages for component definition.

## 2.2 eODL

eODL is a modeling language, which contains components as first class concept. Other key concepts are interface, module, signal, data type and concepts for the description of distributed environments and deployment. Even though eODL utilizes the data type part from IDL (Interface Definition Language of OMG) it is not restricted to CORBA based platforms. The data type part from IDL is here used as an abstract data definition part, which has to be mapped for different platforms.

The definition of eODL is based on a MOF based Meta-Model. The defined concepts are structured on two dimensions. At first packages are used to group the concepts according to their origin. So all concepts taken from IDL are in one package. The second structuring is more on logical level. Inspired by the Open Distributed Processing (ODP) [6] in eODL all concepts are assigned to one of the following view points:

- *computational view point*,
- *implementation view point*,
- *deployment view point*,
- *target environment view point*.

Some view points are here used in the same way as in ODP, but eODL identifies also other view points. They are defined around the development cycle of software components. So the concepts from *deployment view point* are only meaningful for implemented software components and not for CO-Types in the design stage.

## 2.3 CCM and CIDL

The CORBA Component Model [5] is a standard published by the OMG. It provides the Meta-Model for CORBA components and the technology and runtime

environment for components developed using that Meta-Model. It is based on mature CORBA technologies like the GIOP protocol<sup>1</sup> and language bindings for implementation languages.

The component model of CCM defines two kinds of interactions for components. There is a RPC-like interaction with request/response and a signal-like one with events. For each of these interaction kinds components can declare the usage or the provision.

For the notation of models CCM extends the IDL2<sup>2</sup> syntax by rules for components. CCM also contains a mapping from IDL3 (IDL2 + components) to the older IDL2. This was introduced for a compatibility with older, not component-aware CORBA clients.

But IDL3 covers only the computational aspects of components, which are first class concept here. For the description of some implementation aspects the Component Implementation Definition Language (CIDL) was published by OMG. CIDL is an superset of IDL3 and therefore contains all computational concepts. Furthermore it introduce the grouping of interfaces for implementation purpose.

## 2.4 Apply MDA

After the identification of the development domain (distributed software components) we apply the MDA approach. This includes the full specification of computational objects and software components in the PIM level. But eODL only covers the structural aspects for this kind of specification. The full picture of involved model classes and languages is shown in figure 2. Here we see that in addition to eODL SDL is used to provide a mechanism for behavioral specification. It also illustrate the transformation between eODL and SDL model (**a** in figure 2).

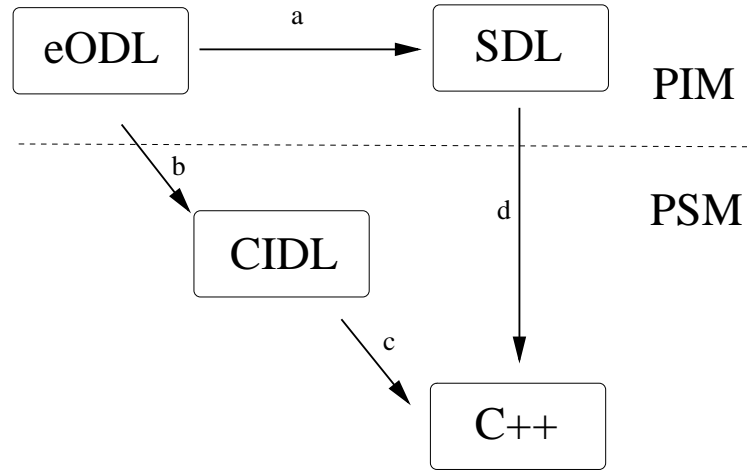
As target technology for the final software system we choose in this paper CCM as platform technology. The model class for structural aspects of software components is therefore CIDL. Also the aspects from implementation view point in eODL can be transformed into corresponding concepts in CIDL (**b** in figure 2). Only the behavioral aspect, which is expressed in the SDL model can't be transformed into the CIDL model. This is because there are no suitable concepts. Final goal of the development process is a running system and this means executable software. Most technologies defined by OMG are neutral regarding implementation language and base there mapping to implementation languages on existing mappings. CCM/ CIDL is no exception from this rule an the regular mapping from IDL2 can be applied here.

In this paper we will not focus on the full set of model classes involved in the development process but rather on the technical realization of one transfor-

---

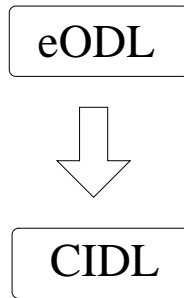
<sup>1</sup> The General Inter ORB Protocol defines the exchange of requests and replies for RPC interaction.

<sup>2</sup> IDL2 is the 2.x version of the Interface Definition Language standardized by the OMG.



**Fig. 2.** Models and related transformations

mation. Therefore we narrow the process to the **b** transformation (see figure 3).



**Fig. 3.** Focus on technical realization of the transformation

A more detailed description of the whole development process shown in figure 2 can be found in [2].

## 2.5 Example for a rule

The transformation from eODL models to CIDL models is defined by using natural language. Unfortunately there is no well established formalism<sup>3</sup> for transformation definition. In the natural language we establish a relation between

<sup>3</sup> The RFP for Query/View/Transformation of models in MOF 2.0 is still under progress at OMG.

concepts of source and target Meta-Model. Pure type based transformation rules are not possible in most cases. Only if the concept structure of source and target Meta-Model is nearly the same this can happen.

In the following example rule we show a mixture of type based and pattern base definition. The example uses this *font* for concepts in the Meta-Models and this *font* attributes of concepts.

For each *SignalDef* in the eODL model exist a *EventDef* in the CCM model with same name. Each pair of name and data type associated in *CarryField* from the eODL model is mapped to a *ValueMemberDef*, which is defined in the scope of the current *EventDef*. All created *ValueMemberDef* are public visible (*isPublicMember==true*).

The example rule transforms the concept *SignalDef* from the source Meta-Model in the concept *EventDef* of the target Meta-Model. This part of the rule is pure type based. But to transform the signal based interaction from eODL to CIDL we also need the correct relation between signal and signal parameters. Structure of the Meta-Models differs here and we have to use a "pattern" as instrument.

### 3 MOF based Repositories in Java

In this section we explain the relation between MOF and tools in the Java environment. This starts with a short overview of MOF and continues with related Java standards.

#### 3.1 MOF

MOF (Meta Object Facility) is a standard for modeling data on different levels of abstraction. MOF provides a framework that supports all kinds of metadata. The architecture of MOF is based on the four layers as shown in figure 4.

- M3: The Meta-Meta-Model defines a language (MOF) with/by which the underlying Meta-Models are specified. For example: Meta-class, Meta-attribute, Meta-operation.
- M2: The Meta-Model is an instance of the Meta-Meta-Model and it defines the language for the models. For Example: class, attribute, operation.
- M1: The Model is an instance of the Meta-Model and it defines the language for the domain. For example: class:book, class:author, operation:setAuthor (for class:book).
- M0: The Data are instances of the Model. For example: instance of the class:book with name "cook-book".

The MOF specification describes an abstract language for managing platform independent Meta-Models. For example: the Unified Modeling Language (UML), the Common Warehouse Metamodel (CWM) and the MOF itself.

The MOF-specification contains:

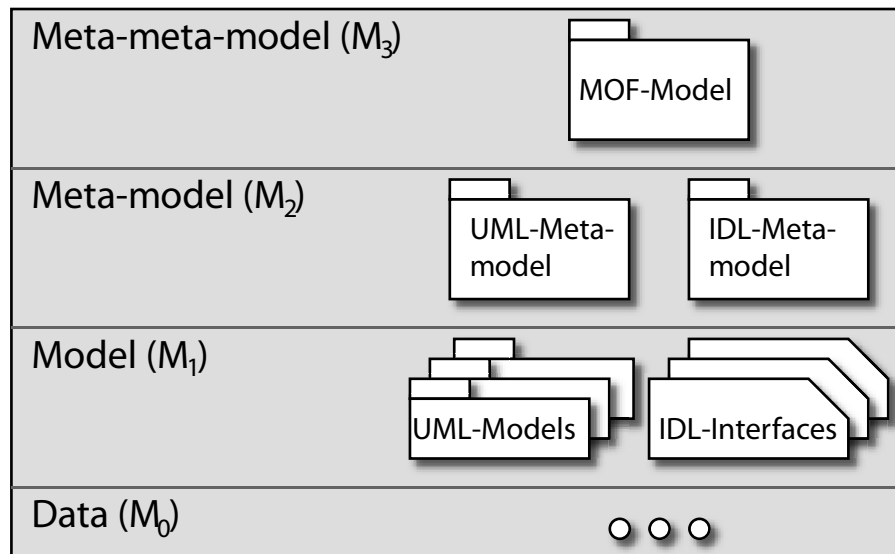


Fig. 4. Four-layered architecture

- a specification of the MOF Meta-Meta-Model, thus an abstract language for the specification of MOF Meta-Models
- a mapping of MOF Meta-Models to CORBA IDL (Interface Definition Language)
- a set of CORBA IDL interfaces for managing Meta-data, independent from the Meta-Model
- the XMI mapping (XML Metadata Interchange) for exchanging Meta-Models

A repository is a storage place in general. In this case we store all the data and Meta-data in the repository. Any repository object has an identity, a unique ID called MofID. That means that every repository element can be identified and accessed explicitly by its MofID.

**Extents** In the MOF specification statements about object locations have no place. Therefore the MOF assumes a conception of context in many areas:

- The classifier-scoped features of an M2-level Class are notionally common to "all instances" of the Class.
- Mapping typically allow a client to query over "all links" in an Association instance.

It is infeasible to define "all instances" or "all links" as meaning all instances or links in the known universe. For that reason, the MOF specification defines logical scopes of M1-level instances that are the base of these and other "for all" quantifications. These scopes of M1-level instances are called extents.

Every class instance belongs to precisely one class extent. These extents are part of package extents, depending on the structure of the Meta-Model. This means that extents are strictly hierarchical. Extents are related to the intrinsic container semantics of Meta-Objects.

The extent of a class is the entire set of M1-level instances of the class. The class instance will be created in connection with a class extent. The class persists within this extent for its complete lifetime. The same applies to associations. The extent of a package is a collection of class, association and other package extents.

### 3.2 JMI

JMI (Java Metadata Interface) provides a platform independent definition to describe Meta-data [9]. With JMI you can create, modify, access, exchange and store Meta-data. JMI is based on the Meta Object Facility (MOF) developed by the Object Management Group (OMG). JMI defines the standard Java interface to the modeling components which are part of MOF. JMI also provides for Meta-Model and Meta-data interchange via XML by using the standard XML Meta-data Interchange (XMI) specification[3]. For each model element in MOF an interface is defined in JMI to access it. Packages of MOF are accessed by the **Refpackage** interface of JMI, Association by **RefAssociation** and for each element a **RefClass** interface which manages the set of objects and the **RefObject** interface for access to the object itself. To access a concrete Meta-Model and instances of it the interfaces have to be specialized. That means that each interface for the specific model elements is a specialization of the Ref-interfaces named according to the model element. For the instances of the MOF-class two interfaces are created, one named according to the model elements name plus class and the second uses only the name. With the use of JMI as a mapping from MOF onto Java the implementation of a Meta-Model based mapping in Java is facilitated.

### 3.3 MDR

The repository we used for our implementation is called MDR (Meta Data Repository) [10], it is written in Java and developed by the netbeans community. It contains implementation of MOF repository including the persistent storage mechanism for storing the metadata. The interface of the MOF repository is based on and fully compliant with JMI [9]. The MDR provides the generation of JMI interfaces and a XMI Reader and XMI writer facility as well. It allows instantiation of any MOF compliant Meta-Model and models in the Meta-Models. We chose MDR as a repository because it is a Java and JMI based free available implementation and because of its built-in features.

## 4 Syntax based languages and MOF repositories

Our aim was to enable the transfer of textual notated eODL-programs to the MOF repository. Why do we want to transfer eODL source code files to the

repository? On the one hand thus we have another possibility of input apart from model input via XML, on the other hand we achieve a high integration, because there are tools, which create textual eODL notation from graphical eODL notation.

In this chapter we describe how textual notated programs in eODL can be transferred to the MOF repository. Here programs are the eODL source code files. eODL is a syntax based language. Its grammar is defined in Z.130. In the following part we give a short introduction to ANTLR which is the compiler generator tool we used. Apart from ANTLR we used MOF and JMI. Both are described in section 3.

#### 4.1 ANTLR

We used ANTLR (ANother Tool for Language Recognition) [11], a compiler generator tool, to manage the lexical and the syntactical analysis process and to construct the objects in the repository by means of semantical actions.

ANTLR is developed by Terence Parr. It constructs recursive descent parsers from LL( $k$ ) grammars, for  $k > 1$ .

ANTLR integrates the specification of lexical and syntactical analysis. A separate lexical specification is unnecessary. Lexical regular expression (token descriptions) can be placed in double quotes and used as normal token references in an ANTLR grammar.

ANTLR accepts grammar constructs in EBNF (Extended Backus Naur Form) notation. It provides facilities for automatic abstract syntax tree (AST) construction and modification.

ANTLR allows each grammar rule to have parameters and return values. It converts each rule to a Java function, a rule parameter is simply a function parameter. Additionally, ANTLR rules have multiple return values.

#### 4.2 From syntax to model creation in MOF repository

As we used ANTLR for building the parser we had to change some things concerning the grammar of eODL. For example: We eliminated the optional usage of a meta symbol which is optional itself. After we had made the corrections the parsing of eODL-source files was possible.

Beside the parsing the source code in the first pass we also generated an abstract syntax tree (AST). In contrast to generate code as we would do for building an usual compiler we had to create model elements in the repository.

We used local symbol tables for each node in the AST, so we can manage and resolve container-content relations. These relations we have to transfer to the repository.

We also built up two global symbol tables realized with two associative containers, meaning two hashmaps:

- The first hashmap contains the names of the types already found associated with the created model elements.

- The second one contains elements with missing references.

At this point we can start to create instances of Meta-Model-elements in the repository. We demonstrate this technique with the following example. It is an extract from Dining Philosophers shown in listing 1.1.

**Listing 1.1.** Extract from Dining Philosophers

```

module DiningPhilosophers {
  CO o_Philosopher{};
  CO o_Fork{};
  interface i_Fork;
  interface i_Philosopher;
  interface i_Observer;
  exception ForkNotAvailable {};
  exception NotTheEater {};
  enum e_ForkState { UNUSED, USED, WASHED };
  enum e_Pstate { EATING, THINKING, SLEEPING, DEAD, CREATED, HUNGRY };

  interface i_Fork {
    void obtain_fork ( in o_Philosopher eater )
      raises ( ForkNotAvailable );
    void release_fork ( in o_Philosopher eater )
      raises ( NotTheEater );
  };

  artefact a_ForkImpl {
    obtain_fork implements supply i_Fork::obtain_fork;
    release_fork implements supply i_Fork::release_fork;
  };

  CO o_Philosopher {
    implemented by a_PhilosopherImpl with Singleton;
    supports i_Philosopher;
    requires i_Fork, i_Observer;
    use l_observer observer;
    use i_Fork left;
    use i_Fork right;
  };

  valuetype Pstate {
    public e_PState mystate;
    public string name;
    public i_Philosopher philosph;
    factory create ( in e_PState mystate, in string name, in
      i_Philosopher philo );
  };

  signal PhilosopherState {
    PState carry_pstate;
  };

  interface i_Observer {
    consume PhilosopherState pstate;
  };

  artefact a_Observer {
    pstate_Impl implements supply i_Observer::pstate;
  };

  CO o_Observer {
    implemented by a_Observer with Singleton;
    supports i_Observer;
    provide i_Observer observer;
  };
};

```

In the given source code (listing 1.1) line 13, there is the following declaration:  
`void obtain_fork ( in o_Philosopher eater )`  
(within `interface i_Fork`).

Each element, which are to be associated with other elements are created in the repository. We also look up the first hashmap, whether the hashmap contains the element. If this is true, the element is associated. If not, the element have to be inserted in the second hashmap.

Types we find in the AST we create in the MOF repository. Also they will be inserted in the first hashmap. As we find `o_Philosopher` while traversing the AST, we have to look up the first hashmap, whether `o_Philosopher` already exists in the model. If not, we have to insert `o_Philosopher` in the second hashmap. For every type we find, at first we take a look at the second hashmap: If there is an item, which needs to be associated with the type we have found (this type has to be inserted in the first hashmap as a matter of course), we can complete the model element creation in the repository. In this case we have to delete all these items from the second hashmap. If we have traversed the whole AST and the second hashmap is empty the Model is transfered completely to the repository. We can work with the Model. If not, there occurd an error.

## 5 Implementing Rules in Java

### 5.1 Introduction

This section describes the concrete implementation of the mapping rules with MDR using JMI. The whole project is called `etoc` standing for eODL to CIDL. The transformation is based on the principle of recursive descend and the technique of the Fluxbox. `etoc` is implementing the mapping between an eODL extent and a CIDL extent, first the given extent is transformed into a new CIDL extent and second it is walked to produce a syntax based output into an IDL and CIDL file. Both parts of the mapping and walking respectively are using the recursive descend principle. In contrast to traditional compiler technology, there is no need for a symbol table or AST (abstract syntax tree). Each element could be identified through its MOF-id. Using the knowledge of these properties allows an easy to understand and straightforward implementation.

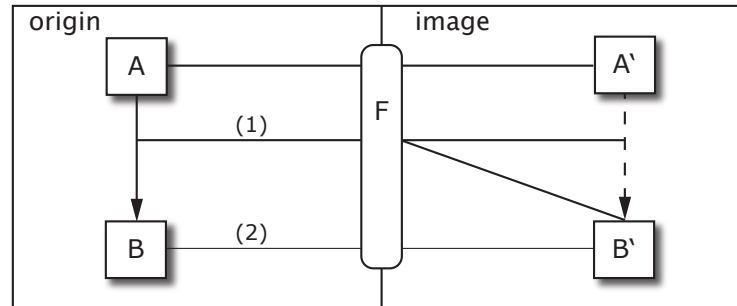
### 5.2 Architecture of `etoc`

The whole transformator consists of two parts due to the splitted process of transformation. There is the `Z130ComputationalViewTransformator` responsible for the walking of the eODL model in respect to the computational view of eODL and besides there is the `Z130ImplementationalViewTransformator` walking the implementational view allowing a splitted and selectional transformation. For further generation of syntactical output there is the `CcmModelWalker` walking the transformed model and generating the output. Besides this components the `FluxBox` has to be used for transformation.

### 5.3 One pass transformation with the Fluxbox

As for the problem of relating two objects in MOF without knowledge of the existence of the other one, a concept is introduced named Fluxbox.<sup>4</sup>

Assume we have two objects named A and B as you can see in figure 5 and F



**Fig. 5.** Fluxbox technique

is representing the Fluxbox. Both are connected through a directed association. The intention is to map A on A' and B on B' and to establish an association between A' and B'. While browsing the model within the repository an instance of A has to be handled. The first step is creating the instance A' with the corresponding properties, the second is to relate A with B. While browsing the first time (1), one cannot ensure if B is existing, so it would be impossible to associate with a nonexistent object. The Fluxbox allows to create an instance of B using the origin objects MOF-id for uniqueness and reference, without specifying B any further. The Fluxbox's base is creating a placeholder element, that can be accessed and changed by demand. By giving the Fluxbox the meta object defined by JMI for creating the image object paired with the key of the original object the image element is created and returned. In further model walking the element B is found (2), so B' is again accessed through the Fluxbox using the MOF-id of B and specified according to the mapping from B to B'. The Fluxbox is an container, to be more precise a `HashMap` with the original objects MOF-id as the key and the image model element as the contained element. The Fluxbox's only operation is getting the object according to the given key. If there is no object associated with the given key a new one is created, else the existing object is returned for manipulation. Thus the Fluxbox solves the problem of possible not existing objects and simplifies the whole process of transformation by giving the possibility of one pass transformation.

<sup>4</sup> It was originally developed by Markus Scheidgen (scheidgen@informatik.hu-berlin.de) for handling of such problems.

## 5.4 Transforming the model

The Transformator `etoc` supports two possible inputs, syntax based input files parsed by the eODL parser and model injector as described in section 4 or the XMI representation of the eODL model. Both methods result in an eODL extent within the repository, the first one by creating the eODL extent directly while parsing, the second by using the built-in XMI-Reader of MDR passing the extents target location as an argument while calling. Because OCL is not fully supported in MDR the formulated constraints are not considered, so it is possible to add them at any time. The concepts which are not supported by the mapping like *MediaDef* or the *deployment view* are not considered in the implementation. Despite that approach they are read and included within the repository. This preserves the implementations future extensibility.

Having the model instanced in the repository further actions are necessary. First a CIDL extent is created and named unique according to the given extent of eODL. Second the given eODL extent has to be walked recursively for transforming the model. The key for transforming the model are two techniques. The first is the *container-content* relation and the second is the Fluxbox using the MOF-id. Because of the *container-content* association the model is presenting itself in a tree structure meaning each model element is connected to his container. Therefore the entry points for walking can be determined by getting all top level objects. These elements are implicitly contained within the virtual global module forming also container for elements. Each of these container elements forms a separate tree of containment, so the whole model is representing itself as a set of trees. Every tree is a spanning-tree meaning it is covering all contained model elements, because of the containment restriction of eODL and CIDL respectively. This assures that every model element is reachable and can be walked. Using the Fluxbox technique bypasses the problem of not yet existing model elements and is allowing a one pass transformation. So every model element which has to be created during the process of transforming is generated out of the Fluxbox. The mapping is done in two steps, the first is the cloning of the common IDL core the second the mapping rules for the eODL specifics. Implementing the cloning is following a straightforward schema. For each common model element the elements attributes and references are copied to the new created element. Also the associations are copied by setting the according reference objects on the new created model elements. Implementing the mapping rules is done as straight forward as cloning the IDL core. Because that each element of eODL is mapped on one CIDL element the Fluxbox can be used with the original elements MOF-id. This allows the assignment of the image element to the origin model element. Afterwards the repository contains the eODL model and the CIDL model resulting of the transformation, so both of the models can be accessed and manipulated on demand.

## 5.5 Walking the transformed model

After the transformation has finished the extent is walked by the `CcmModelWalker` to produce a syntax based output in files. The principle of the walker is again

the recursive descend using the *container-content* relation. While passing each element the corresponding code is created by converting types to strings and resolving the namespace. The walk is not obligatory because there is the possibility of writing the result model in a XMI file using the built-in feature of MDR generating XMI output.

## 5.6 Using JMI

This section covers the more implementational aspects of the transformation. Accessing the model is done by using JMI and MDR which is implementing the provided operations. In the following the access is described by example. Handling of the package hierarchy of eODL and CIDL is done by providing elements of the type `RefPackage` which delivers the class proxy objects and association objects. The type `IdlPackage` grants access to all IDL model elements of eODL and the type `CCMMetamodel` to these of CIDL. Note that `z130` is an instance of `Z130Package` which is the start point of the extent and therefore the extent itself.

**Listing 1.2.** example for package handling

```
ComputationalViewPackage compView =
z130Extent.getAdvancedConcepts().getComputationalView();
Iterator signalDefIterator = compView.getSignalDef().refAllOfType().
iterator();
```

The iterator provides the browsing of the collection of instances each of the of the type *SignalDef*. For each model element there are two types of interfaces granting access. The first one is a specialized element of *RefClass* representing the proxy object which is managing the set of instances. The second one is a specialised *RefObject* object providing the access to the model elements instance itself. In the following example the former *SignalDef* element is walked and named `signal`. The *EventDef* in context of CIDL is created with the Fluxbox and named `newEvent`.

**Listing 1.3.** example for element mapping

```
EventDef newEvent = (EventDef) fluxBox.getObject(ccm.getComponentIdl
().getEventDef(), signal);
newEvent.setIdentifier(signal.getIdentifier());
newEvent.setRepositoryId(signal.getRepositoryId());
newEvent.setVersion(signal.getVersion());
newEvent.setDefinedIn(setContainer(signal.getDefinedIn()));
```

As you have seen the attribute were copied by using the get-functioncalls on the original element using their value on the set-functions of the new eventtype. The members of the signal are also traversed recursively. The setting of their *Idltype* is the setting of the reference object of the *TypedBy* association. Another way of handling the association instances is getting the `RefAssociation` objects allowing to add or remove instances taking part in the relation. Each association is specialized according to the related types. Besides this it is also possible to access them by setting or getting the proper reference attributes. The code below describes how to handle the association proxy object.

#### Listing 1.4. example for association handling

```
z130.idl.InterfaceDef interfaceBase ;
ccmmetamodel.baseidl.InterfaceDef interfaceDerived ;
...
InterfaceDerivedFrom derivedFromAss = z130Extent.getIdl().
    getInterfaceDerivedFrom();
derivedFromAss.add(interfaceBase, interfaceDerived);
```

Obviously the association is retrieved through the `z130Extent` both interface objects are created by using the Fluxbox. Then accessed by adding the participating objects to the association. The objects also grants functions like `remove`, `exist` and so forth.

## 6 Conclusions

As we showed in this paper the implementation language Java, the JMI standard and the JMI implementation MDR build a power-full combination for the realization of MDA related tools, like transformers. More generally, it demonstrates, that MDA is a real use case for software development, all base technologies are available as products. But the transition to new standards like MOF 2.0 from OMG is not supported by current Java tools.

## 7 Further work

Most of the work to build a transformer can be done by using tools. The lack of formal description methods of transformations is the main obstacle here. So the definition of such formalism has to be tackled in future. The activities in the Query/View/Transformation RFP by the OMG have a great potential. But only tools for automatic implementation derivation in Java will support the application of MDA.

Also the automatic or semi-automatic integration of traditional syntax based languages in the Meta-Model centered world is a field of future activities.

## References

- [1] ITU-T: Extended Object Definition Language (eODL), ITU-T SG17 standard, Z.130, (2003)
- [2] Böhme, H., Fischer J.: eODL and SDL in combination for components, Proceedings of the SAM work shop 2004, Ottawa, Canada, (2004)
- [3] Object Management Group: XML Metadata Interchange (XMI) version 1.1, OMG document, formal/00-11-02, (2000)
- [4] Object Management Group: Meta Object Facility, Version 1.3, OMG document, formal/00-11-02, (2000)
- [5] Object Management Group: CORBA Components, v3.0 full specification, OMG document, formal/02-06-65, (2002)
- [6] ITU-T: Open Distributed Processing, ITU-T/ISO, ITU-T Recommendation X.901 X.902 X.903 X.904, (1997)

- [7] Michael Piefel, Markus Scheidgen: Metamodelling SDL, Metamodelling Languages, Twelfth SDL Forum 2005, Ottawa, Canada, (2005)
- [8] Object Management Group: Object Constraint Language Specification (OCL), OMG document, ad/1997-08-08, (1997)
- [9] Java Community Process: The Java Metadata Interface(JMI) Specification(Final Release), JSR-000040, (2002)
- [10] netbeans.org: Metadata Repository (MDR), <http://mdr.netbeans.org/>
- [11] ANTLR: ANother Tool for Language Recognition, <http://www.antlr.org/>