General Architecture and Architectural Comparison of OMG Meta Object Facility Repository Management Systems

Ilia Petrov
Databases and Distributed Systems Group, Technische Universität Darmstadt,
Hochschulstr. 10, 64289 Darmstadt, Germany
petrov@dvs.tu-darmstadt.de

Alejandro Buchmann
Databases and Distributed Systems Group, Technische Universität Darmstadt,
Hochschulstr. 10, 64289 Darmstadt, Germany
buchmann@dvs.tu-darmstadt.de

Received May 12 2008; Accepted February 23 2009

Abstract: Metadata repository systems store metadata in the form of models and meta-models. In this paper we introduce a general architecture of a OMG MOF (Meta Object Facility) repository system and describe its modules. In addition, we examine the architectures of several existing MOF repositories such as MDR (MetaData Repository), EMF (Eclipse Modeling Framework), dMOF (product name, not abbreviation) and iRM (integration Repository Manager) and illustrate how these related to the proposed general architecture.

Keywords: Metadata, Metadata Repository Systems, Architecture

1. Introduction

Repository systems are “shared databases about engineered artifacts” [4]. They store metadata describing class or data-type definitions, models, as well as structured or unstructured descriptions. They facilitate integration among various tools and applications, and are therefore central to an enterprise.

Logically, the repository metadata is organized into a set of stacked metadata layers called levels (briefly described in Table 1). We call this organization layered metadata architecture. Existing metadata related specifications such as ISO IRDS (Information Resource Dictionary System) [8-10], CDIF (Case Data Interchange Format) [5,6], MDC OIM which is merged with OMG MOF (Meta Object Facility) [13], etc. exhibit very similar layered meta-data architectures. They serve as a blueprint of the logical organization of repository data and together with the meta-meta-model describe the conceptual structures and entities a repository system is managing.

We use the term Repository Management System (RMS) to refer to the system for managing structured collections of metadata, usually called repositories. This is analogous to the term Database Management System (DBMS) that manages structured collections of data called databases. From engineering point of view different repository management systems tend to exhibit common design structure and provide similar metadata management functionality.

In contrast to the layered metadata architecture, which is conceptual metadata architecture, the repository system architecture describes the way the repository management system is engineered and acts as a blueprint in terms of system modules and interfaces. In the present paper we focus mainly on the system architecture of MOF repository management systems and analyze their functionality.

The contributions of this paper are:

a. We introduce and describe a general architecture of MOF RMS. There have been only sparse research efforts describing the general organization of MOF repository management systems mainly focusing on a concrete RMS. A general architecture has not been published.

This paper is invited as an outstanding presentation from iiWAS 2008 and and modified by the authors.
Ilia Petrov and Alejandro Buchmann

2. General Architecture of a MOF Repository System

The general architecture of a MOF repository system is illustrated in Figure 2, which presents a detailed view on Figure 1. In this section take a concrete look at every module of the architecture starting with the repository API.

Repository applications access the metadata and the repository management system through the RMS API. The basic version of MOF RMS API comprises different parts, the most important of which are: the JMI (Java Metadata Interface) Reflective API, the transactional support API, as well as the package management and entry point API (Figure 3).

The JMI Reflective API [15] provides a uniform access to repository artifacts (repository objects) on levels M1 through M3. The MOF API provides access to the MOF meta-model; it can also be done through the reflective API however in many of the existing RMS it is also provided as generated interfaces since MOF is immutable and self-descriptive. We use the term metadata API to refer to the combined JMI and MOF API, which is the basis for accessing the bulk data instance data (e.g. objects, records).

Table 1. Layers in OMG MOF Metadata Architecture

<table>
<thead>
<tr>
<th>Layer</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3</td>
<td>Meta-meta-model (MOF)</td>
<td>A standardized language, in terms of which, definitions of underlying metamodels are expressed.</td>
</tr>
<tr>
<td>M2</td>
<td>Meta-model</td>
<td>Language for defining the structure (syntax) of a whole set of application model definitions. Structural definitions may be extended with the semantics of application domain definitions.</td>
</tr>
<tr>
<td>M1</td>
<td>Model</td>
<td>Application Model (Application classes, Table definitions etc.). Alternative term is “information model” [4].</td>
</tr>
<tr>
<td>M0</td>
<td>Data</td>
<td>Instance Data (e.g. objects, records)</td>
</tr>
</tbody>
</table>

b. We provide an architectural comparison between architectures of different RMS. This is to the best of our knowledge is the first one.

c. We provide a summary of features required by the MOF repository in different scenarios such as: embedded, client-server, JEE server etc. In addition we discuss different features with an enterprise grade MOF repository must have.

The paper is organized as follows. In the rest of the present section we introduce a bird’s eye view on a repository system. In Section 2 we present the general architecture of a MOF repository system and a set of classification criteria for MOF RMS. In Section 3 we discuss the architectures of various repository management systems and discuss the related work in Section 4.

On a general level repository systems are said to be a layer of software on top of a relational database system offering various services for object (or entity) and relationship management [2] through a repository API. The components of the general repository architecture (Figure 1) are: the repository programmatic interface (API); the repository system itself RMS (Repository Management Systems); the persistence layer.

The repository interface (Figure 1) comprises at least three groups of functions:

a. The most important is the programmatic API. It exposes functionality for storage and retrieval of metadata (model element management like object, (entity) or relationship management), model transactions, lifecycle management and optionally version control.

b. The query interface for executing queries against the repository data formulated in a declarative query language.

c. An import/export interface for repository metadata interchange. Some repository systems implement querying functionality directly on top of the data store (database). Others implement the query facility on top of the repository API, with the advantage that it is portable and uses the repository API to perform certain operations faster and simpler.
of repository metadata.

The XMI (XML Metadata Interchange) [14] Import/Export API allows to import MOF compatible M1 and M2 models into the repository as well as to serialize repository data on the respective levels into XMI files, to facilitate the exchange with CASE tools, or other RMS or backend systems.

In addition we introduce a simple Transactional API which encapsulates transaction processing functions such as BeginTransaction(), EndTransaction(), Commit() and Rollback() or alternatively Save(), depending on the transaction module.

The RMS API supports a set of functions serving as entry points to the repository. In other words the repository applications will start working with the repository using these functions as the first RMS API calls. These are functions such as retrieving a M2 model package or the set of all M2 packages as well as functions for retrieving package and extent names. Last but not least the repository API may expose various utilities and helper tools such as static interface generators, search functions, partitioning API, etc. Those are mostly repository system specific.

Repository applications perform metadata access through standardized metadata API (Figure 3); although the API is generic and model-independent it is complex to use. To simplify the metadata access and manipulation applications generate static interfaces for a specific metamodel, called JMI Generated Interfaces (Figure 2). The static interfaces (Figure 2) are simple static wrappers of the reflective functionality; they use internally the reflective API and provide slightly higher level of CRUD (Create, Read, Update, Delete) operations (setter/getter methods for every attribute, factory functionality for creating M1 “instance classes” and removal) for every metamodel model element. If it can be safely assumed that the respective metamodel is immutable and can be changed at no application-lifecycle phase by no party the static interfaces are the better choice. This assumption is made by many repository systems. Alternatively an application may interface with the repository over the base reflective API gaining the full flexibility but also adding a degree of code complexity. The portioning API is an optional part (it is barely specified in MOF) it may however greatly improve the performance of an RMS. The goal is to be able to define arbitrary groups of repository objects (called partitions) and be able to read and write them altogether.

The repository management system (Figure 2) comprises a number of modules such as the metadata manager, the transaction manager, the object cache and the storage managers.

The metadata manager provides the implementation of the metadata API. Roughly speaking the metadata manager alone would represent the most elementary MOF repository (embedded and in-memory, without transactional support). The metadata manager may also implement some of the MOF consistency constraints in absence of a dedicated consistency manager (Section 3.3).

The transaction manager (Figure 2) is in charge of the transactional functionality the RMS provides. Depending on the mode it supports it either exposes the four general transaction management operations BeginTransaction(), End-Transaction(), Commit() and Rollback(), or exposes the single operation Save().
Although on the architecture diagram it is depicted as a separate module in practice its functionality is shared among the metadata manager the persistence provider the transaction manager and the object cache. The transaction manager includes a sub-module managing locks on repository objects handled by the metadata manager. Typical operations are: 

- `addLock(lock, object)`;
- `checkLock-Compatibility(object)`;
- `hasLock(lock, object)`;
- `releaseLock(lock, Object)`;
- `releaseLocks(object)`.

The Object Manager (Figure 2) handles repository object retrieval requests on behalf of the metadata manager; it implements on demand fetching and caching of repository objects. Whenever the metadata manager requests a repository object (because the repository application has navigated to it) it directs the fetch request to the object cache manager, which interacts directly with the storage manager to read the respective object. However the all other objects the repository object may reference are not loaded immediately, although metadata manager is provided with the illusion that they are (by creating “proxies”). If the metadata manager requests one of them the object cache manager loads them transparently. Repository objects are highly interconnected with each other (due to the metamodel and the JMI data model), hence such a mode of operations trades “performance” for minimizing memory consumption in a beneficial balance. Typical operations on the object cache manager interface are:

- `getObject(ID)`;
- `storeObject(obj)`.

To the best of our knowledge none of the existing MOF repositories implements buffer management techniques known form object oriented database systems. Hence especially for enterprise scenarios this is an open issue.

Partitioning is an important part of an RMS which is underspecified in the MOF specification. It allows to partition the repository object space into portions called partitions and perform load and store operations on them. Partitioning not only improves the performance but also forces meta-model designer and application developers to think of grouping metadata. It is part of the transaction processing.

The RMS must ideally be able to work with multiple persistence providers: a relational database, a flat files or index files. To abstract from the specifics of each provider the so-called metadata storage manager is introduced, which play the role of persistence SPI (Service Provider Interface). In general specific implementation of the storage manager interface are required for every persistence provider. The typical persistence manger interface would consist of lifecycle functions, the typical transaction management operations (begin, commit, rollback) as well as CRUD operations on object level for the different repository objects.

The bootstrapper is a very important module ensuring that the self-describing MOF model is always loaded in a repository. It resolves the problem of starting the repository for first time after the installation, when the presence of the MOF model is expected but it is physically not present.

A MOF repository is expected to have a Constraint manager, which is an engine for model constraints expressed in certain constraint language. In the realm of MOF a natural choice (inspired by UML) for a constraint language is OCL (Object Constraint Language). MOF makes provision for a MOF Constraint modeling element, related to precisely such expressions, which need to be evaluated (immediately or deferred) by the RMS and the constraint manager in particular. The constraint manager offers the following the operation `evaluate()` or `evaluateDeferred()` and lifecycle operations.

Last but not least an RMS must have a notification manager, which notifies repository applications upon
metadata changes. A repository application A subscribes for model element Me. If repository application B modifies Me then the notification manager notifies application B in the desired way. Notification services are very useful in design time of several CASE (or modeling) applications are performing complementary operations on the same metadata like \{create | update | remove\}Object() etc.

2.1 Classification

Depending on the architecture we distinguish embedded repositories and shared repositories. Embedded repositories run as part of the enclosing application, promote little metadata sharing compared to client/server ones, and are mainly indented for design-time use. As a general rule of thumb these run in single user mode, exhibit simpler architecture, less complicated transactional facilities and typically use file-based persistence. In contrast the shared repositories are intended to be multi-user, promote data sharing and run independent of the repository application. They are generally client/server based. As a general rule of thumb these exhibit more complex transactional architecture and typically rely on a database system as a persistence provider.

Another classification criterion is the RMS runs locally or in a server infrastructure such as JEE Application Server. Local RMS typically run locally on a computer but can be accessed remotely over some remoting technology such as Java RMI, RPC, CORBA remote calls etc. Local RMS do not utilize application server infrastructure or services. Server RMS, in contrast, are specially designed to run on an application server, conform the expected component or library model and provide metadata services to the applications within the applications server.

Besides the above two classification categories there may be several others such as (a) single-user vs. multi-user; (b) design-time vs. multi-phase; or (c) in-memory vs. persistent.

3. Architectures of Existing MOF Repository Systems

In this section we describe the architectures of existing MOF metadata repository products and compare them to the general architecture described in Section 2. Two general issues need to be pointed out in this context: (a) the comparison relies on the product documentation and on the code (where available); and (b) it is very difficult to compare commercial products such as SAP MMR [21], SAP MOIN [22] or Adaptive [19], mainly because the documentation stresses features and usability whereas technical blue prints are naturally not published. As a result we compare mainly open source products, however we also try to consider the commercial products. For the latter we can only make marginal statements. A summary of the comparison results can be seen in Section 6, Table 3.

3.1 Sun NetBeans Metadata Repository (MDR)

The repository applications access MDR through the metadata API comprising the JMI API and the MOF API. Additional services are exposed through the utilities API. For example, MDR provides notification services, XMI import/export services, static JMI interface generation utilities, indexing services, etc.

In order to achieve the best flexibility MDR generates the implementation of the static interfaces dynamically (on-the-fly) they utilizes the so-called handlers to access persistent metadata through the persistence API. This approach allows to load any meta-model on the fly and receive the generated interfaces allowing for meta-model specific meta-data access, while still allowing generic access to metadata instance of other metamodels. In addition the event notification functionality is also implemented by using handlers. Conceptually speaking the handlers represent the internal API of MDR. For example,
there dedicated are handlers for the most relevant parts of the JMI reflective API.

All repository metadata in MDR is persisted using the persistence SPI. The primary storage structure assumed by the SPI is a key-value index. The key is the repository object identifier, while the value contains the repository object in a serialized form. It can be realized by different structures such as b-Trees, Hash-tables, database tables with indices etc. The default realization is a file based b-tree store.

MDR supports in addition attribute based indexing. If an attribute from a meta-class in the metamodel is tagged as indexed then MDR will generate an index for all M1 values of the respective attribute. Using indices provides better performance for load and search operations. Indexing is used as an add-on to the default B-tree storage.

MDR supports a simple query facility on a programmatic API level. It does not include support for a declarative query language but rather provides a selection filtering support.

At present MDR does not include support for OCL constraints itself. There are however multiple projects (such the Dresden OCL-Toolkit) that do implement it. Hence the standalone MDR does not include a constraint manager, there are however projects implementing it. MDR also does not support versioning; therefore version manager is not present.

MDR implements the consistency support for all MOF constraints directly in the handlers. The transaction manager functionality is shared between the handlers and the persistence managers. MDR provides a simple transaction API distinguishing between read and write transactions; demarcation is done by means of two methods beginTrans( mode ) and endTrans().

3.2 DSTC dMOF

dMOF [7] is a MOF repository developed by DSTC (Distributed Systems Technology Centre). dMOF is one of the first (if not the first) MOF repository implementation. Historically dMOF is a pre-JMI implementation therefore many of the existing “metadata repository application lifecycle phases” are different. dMOF is shared, server-based MOF repository. The metadata in a dMOF repository can be shared among multiple repository applications. dMOF is CORBA based and can run as a server.

The meta-model is expressed in a text based neutral format called MODL file. Next its MOF representation is generated (tool called modl2mof). Next the IDL and the Java generated interfaces are generated (by the tools mof2idl and mof2moflet). These are the counter part of the generated JMI interfaces.

From the user guide only a few aspects of the dMOF architecture can be reconstructed. dMOF supports the generated and reflective interfaces. The RMS lacks modules such as constraint or consistency manager, versioning is not supported. Initially dMOF has been designed as main-memory repository, in version 1.1 a database persistence server has been added. Its transaction model is JDBC based and was introduced with the persistence service – therefore we cannot speak of transaction manager.

The dMOF repository API (Figure 5) comprises several sets of interfaces. One set are the repository interfaces. These interfaces are used to handle naming and different M1 model views with respect to the metamodels: for example the “MultiRepository” [7] interfaces allows an application to view the models as conforming to many meta-models; the “SimpleRepository” [7] interface provides a single meta-model view of the M1 metadata. The Repository interfaces also handle naming and naming contexts. dMOF repository API also offers operations for deletion or freezing of a repository. The term repository in dMOF is logically associated with a well defined combination of M1 models and M2 meta-model(s); and is technically associated with a set of generated interfaces and dMOF runtime libraries.

3.3 iRM-Integrated Repository Manager

iRM is a shared, local repository system: it can handle multiple repository applications accessing and sharing simultaneously their metadata; although it has client/server architecture it does not offer any application server or middleware integration facilities hence we classify it as local. The key parts of the log-
The metadata manager is the logical module of the iRM RMS that is directly associated with the MOF meta-data architecture. It organizes the repository objects into the four layers, it organizes them in MOF compatible model structures, e.g. models, packages, classes etc., it exposes a set of manipulation operations over the RMS API.

The consistency manager is the module which automatically checks and enforces the structural consistency of the repository data. Enforcing consistency is one of the advantages of iRM to other MOF repositories. The consistency manager makes sure, for example, that every repository object has a type, or that every association has two association ends, or that not an instance of an attribute is created without being assigned to an instance of a class.

The transaction manager is a logical entity, whose functionality is shared among several modules such as the meta-data manager, the lock manager the session manager and the persistence manager. The lock manager ensures that a repository object and its dependent objects are not accessed by multiple repository applications in an incompatible manner. It implements a locking protocol, developed for iRM, that disallows incompatible read/write (create, delete, modify) patterns. In the current version the lock manager contains a special purpose lock manager client. Operations on the lock manager are addLock (lock, object); checkLockCompatibility (object); hasLock (lock, object); releaseLocks (object) and lock<RepositoryObject> – for the different repository object types; lockModelElement() – for M2 repository objects; or lockM1InstObject() – for M1 objects; releaseLocksOnObject(). Details of the iRM locking protocol can be found in [28].

The persistence layer in iRM is realized in terms of storage managers implementing a well defined storage manager interface. Every storage manager is associated with a storage approach and a concrete data store (e.g. database such as Oracle, DB2 or MySQL). Moreover, iRM has storage manager for storing “metadata”, i.e. repository objects on M1 through M3.

The second major sub-system of the iRM project is the iRM/mSQL engine. It processes queries formulated in a declarative query language called mSQL.
against the repository data. A repository application may use both mSQL queries and RMS API; it would typically implement its entry points into the repository using mSQL queries while successive modification (creation, modification or deletion) operations would be implemented using RMS API calls.

The iRM/mSQL module (Figure 6) consists of a number of the following sub-modules: (i) mSQL parser which includes partial semantic checker and a logical query plan generator; (ii) a query plan execution module; (iii) mSQL wrapper containing the implementation of the different operators and mSQL functions; (iv) miscellaneous other parts such as mSQL connection or the mSQL result set, which is used by the repository application to process the results of the executed mSQL query.

Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF) [20] is developed in the frame of the Eclipse Project. EMF is a MOF-like repository, however not a pure MOF implementation like the rest of the products considered here. EMF uses a model called Ecore as a metamodel instead of MOF. Having said this we still choose to consider EMF for several reasons: (i) EMF is very widespread; (ii) EMF is the centerpiece of the Eclipse MDA technology conglomerate; (iii) EMF supports all relevant standards and technologies; (iv) regardless of the differences between Ecore and MOF the two have a significant overlap.

EMF is generally not to be considered as a stand-alone product but rather an integral part of a group of technologies that are used as MDA (Model-Driven Architecture) tool framework. EMF provides the model/metadata management functionality and transformation functionality, while other tools are used to develop graphical CASE (Computer-Aided Software Engineering) applications. EMF reuses the Eclipse platform functionality.

According to our classification EMF is a local, shared, multi-user, multi-phase repository. To the best of our knowledge EMF does not have a server support, it is only used in local, Eclipse based scenarios therefore it is a local repository. It is also shared in the application and not a stand-alone repository. EMF can be used by multiple Eclipse based applications simultaneously. One of the major goals of EMF is to provide support different design phases: modeling, code generation, packaging, and the respective transformation therefore it can be classified as a multi-phase repository as well.

EMF offers a combination of change notification support and adapter framework. It allows EMF repository objects to react to notification events by being adapted in a controlled way. If object A references object B1 and object B1 has to be replaced with B2 then the setter method of A will notify B1 and B2 and if necessary the respective adapter will be called performing for example some housekeeping on B1. The adapters have to be implemented and cannot be automatically generated. The notification mechanism is also transitive – change notification events are propagated dependent classes along navigational associations. If there is a bidirectional self-association then all instances will be notified.

The persistence mechanism implemented in EMF is easy to extend and flexible. It allows for on-demand loading of resources in the cases where resource sets are used. By default it uses XMI for maximum portability but allows custom implementations for other persistent stores.

EMF implements a reflective API very similar to the JMI reflective API; in EMF terms it is called Dynamic EMF. Add-ons such as EMF.Edit and EMF’s command framework rely on dynamic EMF support.

An EMF add-on includes support for modeling transactions, which are based on undo/redo functionality provided by the standard EMF’s change model. An EMF transaction belongs to a transactional editing domain and encapsulates/spans one or more EMF commands that can be undone/redone as whole automatically upon an error. Consistency is a topic traditionally discussed in the context of transactions and hence an integral part of the repository management system. In EMF validation is delegated to a separate framework, which can be optionally used. EMF developers implement the custom validation methods for specific meta-model and specific constraint set which are invoked by the EMF validation framework. The constraint set may involve meta-model/domain spe-
pecific consistency constraints as well as standard consistency checks such as multiplicities respected, closure rules, references, proxies resolved, values of valid data types etc. Although validation requires custom constraint implementation OCL (Object Constraint Language) may also be used.

The EMF Query is a programmatic API for formulating and executing queries as well as for ResultSet processing; it also includes OCL support utilizing OCL as query language. The EMF Query framework is available in EMF 2.2 or higher versions. It includes three classes SELECT, FROM and WHERE, each of which can be parameterized respectively. EMF queries also support update functionality by using the UPDATE class; update class passes a user defined update function the classes satisfying the condition. Compared to SQL and OQL the EMF Query API does not allow for projections and lacks object construction features. In terms of joins only object-level equi-joins are supported. The WHERE conditions can be logical conjunctive or disjunctive expressions; it also includes equality and XOR operators.

Last but not least EMF provides a powerful code generation framework – generating text/code corresponding to a model. It is based on its own metamodel that is an additive extension of Ecore and allows storing the generation-specific metadata in the same time relating it to the original modeling artifact; the generation meta-model artifacts are specializations/subclasses of the original Ecore metamodel. The EMF code generation framework supports regeneration and merge functionality combining generated snippets with custom implemented code fragments. What differentiates the EMF code generator from generators in MOF repositories is that it is designed to support concurrent editing (model and code), as well as it can work with templates and be extended to a certain degree.

4. Additional Scenarios

According to the classification introduced in Section 2.1 a MOF repository is local or server. One very interesting case are server repositories offering metadata services in JEE applications servers. To the best of our knowledge the only two JEE implementations are SAP MMR [21] and SAP MOIN [22]. A shared server application promotes scenarios such as metamodel integration, meta-model integration, model matching and composition, versioning and constraint integration. They require a support for active or inactive areas of metadata, which serve for merging and metadata and meta-models.

Versioning metadata and versioning models is another little explored area. There are generally two types of solutions: (i) custom, repository-specific versioning; and (ii) the repository uses model-to-text transformations and versions the models and metamodels traditionally in an SCM/versioning system. The advantage of the latter approach is that it can be bound to certain SCM systems and depending on the build utility and project configuration be used for metadata deployment and generation. On the other hand SCMs are available across organization and considered stable and predictable. Technically they introduce a transaction model suitable for modeling transactions and generally a simple API.

Another feature an enterprise-scale MOF repository should have is the ability to import meta-data from various sources. The simplest case is there that there are meta-data stores of different kinds. These have to be integrated in order to allow at least for referencing metadata artifacts. Ideally the creation of combined metadata artifacts will be allowed. This metadata interchange will work bi-directionally, which poses significant challenges to the transformation infrastructure and the metamodel integration. One of the aspects of that bidirectional transformation not sufficiently investigated is the information loss when mapping models between different meta-models. If solved reliably the process may lead to the creation of central modelers, which import meta-data from third party metadata repositories and allow for central artifact design.

5. Related Work

To the best of our knowledge there is no directly related work. We are not aware of other research describing the architecture of a MOF repository. There are several works describing the architectures repository systems [3, 4, 9, 18, 23]. Since MOF is came to life in the context of CORBA and distributed computing there are works like [7, 23] describing metadata repositories in that context. In this paper we considered several MOF repository products. There are however some we did not include because they are representative of other categories [16, 17, 26, 27].

The paper concentrates on MOF 1.4 metadata repositories. MOF 2.0 implementations are still in the early prototype phase [24] and the full standard is yet
to be finalized we do not consider them in the present paper. We believe however that major part of the proposed general system architecture will also hold for the MOF 2.0 case as well.

In addition there are several known architectures of DataDictionary systems [11] that are generally considered predecessors of repository systems. Yet we do not discuss them in detail.

Under architecture we understand a representation of the structure of a software system in terms of its components and the way they are connected expressed in a particular architectural style. Shaw and Garlan [1] consider software architecture to consist of components, connectors, and configurations. The components are computational elements, connectors are descriptions of the interactions between these components, and configurations are the resulting structure of topology of the architecture, which can reflect the attributes of a particular architectural style description of the design domain.

The trade-off over having a general-purpose repository system or one with custom implemented API and set of services tied to as specific meta-model has been discussed in a number of works and experimentally exemplified in [25].

### 6. Conclusions

In this paper we discussed the architecture of MOF repositories. We presented the general architecture of a MOF repository and investigated and compared several existing projects. We summarize the results of our study in Table 3.

Single-user embedded, local repositories have limited support for transactions. Hence modules such as lock manager and consistency manager may be missing. In addition the persistence layer may comprise flat files instead of transactional stores such as databases. Backup and archival functionality is missing or implemented in its most rudimentary form – XMI export. Most of the repositories do not implement special buffer management techniques known from object-oriented database systems. On the one hand this is acceptable for small, embedded RMS because the volume metadata created and needed by the average application is limited. On the other hand for enterprise shared, server RMS this feature is indispensable. Majority of the repositories support generated JMI interfaces and XMI import/export. Extended features such as querying and constraint/OCL support are either unavailable or are implemented as third-party projects.

### 7. Limitations and Future Research
In its capacity of metadata storage system a metadata repository may have numerous additional extensions or features that we failed to discuss in the present paper. Two such extensions are security and semantic web support.

One of the observations we made in this paper is that there are numerous extensions to the core architecture coming from industrial implementations. Many of them have the capacity to augment the MOF standard and should also be considered in paper such as the present one. Yet the majority of them are not publicly available or published in any form making their analysis challenging for the future research.

Another limitation is the degree of abstraction we had to introduce in order to compare the existing repository systems. Some of the underlying technical details had to be ignored.

References

Biographies Ilia Petrov  is a senior researcher at the Databases and Distributed Systems group at the Technische Universität Darmstadt, which he joined in 2008. Ilia holds MSc Degrees from the University of Erlangen-Nuernberg, Germany and the Technical University of Sofia, Bulgaria. He received his Ph.D. from the University of Erlangen-Nuernberg in 2005. Before joining the Technische Universität Darmstadt Ilia was with the SAP working as a software developer in the field of BI. His research interests are generally in the field of database systems, metadata and virtualization.

Alejandro Buchmann  is Professor in the Department of Computer Science of Technische Universität Darmstadt since 1991 and is responsible for the area of Databases and Distributed Systems. He received his first degree from Universidad Nacional Autónoma de Mexico and his MS and PhD degrees from the University of Texas, Austin, in 1980. He was an Assistant/Associate Professor at IIMAS/UNAM and held positions as a senior researcher at Computer Corporation of America, Cambridge Mass. and GTE Laboratories, Waltham Mass. before joining TUD. Alejandro’s current research interests are in the areas of event-based and reactive systems, performance evaluation, middleware, peer-to-peer systems and new paradigms for data management and information processing in cyberphysical systems. Alejandro Buchmann has served on the editorial board of various journals and as general chair and program chair for many conferences in the areas of data management, event based systems, performance modeling and evaluation, and ambient intelligence. He is a member of ACM, GI and the IEEE Computer Society.