On the Notion of Consistency in Metadata Repository Systems

Ilia Petrov, Stefan Jablonski, and Marc Holze

Chair for Database Systems, Department of Computer Science,
University of Erlangen-Nürnberg, Martensstrasse 3,
Erlangen, D-91058, Germany
{ilia.petrov, stefan.jablonski, marc.holze}@cs.fau.de
<http://www6.informatik.uni-erlangen.de/>

Abstract. Repository systems handle the management of metadata and meta-models. They act as data store with a custom-defined and dynamically adaptable system catalogue. This feature finds a useful application in systems such as process engines, collaborative and information systems, CASE tools and transformation engines, in which custom-defined catalogues are rarely available due to their complex nature. In this context repositories would improve those systems' ability to adapt and allow for dynamic information discovery. Preserving the consistency of the repository data is a major challenge. Repository consistency has several aspects, the most important of which is structural consistency. It is insufficiently specified in the metadata and repository standards, and is incompletely implemented in existing systems. In this paper we propose a novel approach to enforcing structural consistency in MOF-based repositories. We describe its implementation in iRM/RMS - a prototypical OMG MOF-based repository system [35]. We show how this algorithm overcomes the deficiencies of the existing approaches and products.

1 Introduction

Repository systems are “shared databases about engineered artifacts” [4]. They facilitate integration among various tools and applications, and are therefore central to an enterprise. Loosely speaking repository systems resemble data stores with a new and distinguishing feature – a customizable system catalogue. Metadata are data which refer to other data; they describe (a) certain aspects of the way the data are structured (structural metadata); (b) auxiliary or system-specific properties of the data (descriptive metadata). Metadata repositories are systems for handling metadata (some handle also applications' data). The organization of repository metadata (metadata architecture) is defined in [23, 15, 7, 11, 25]. It exhibits a typical layered, multi-level structure. Preserving consistency between the different layers (Table 1) is a major challenge specific to repositories. The relationship between artifacts on two adjacent levels is the type-instance relationship, i.e. definitions on a level are instances of definitions on the next higher level. While in theory this progression can be continued infinitely, in practice it is limited to four layers due to the self-description phenomenon.

A database system contains layers M_0 through M_2 , where M_2 is immutable. To provide a custom-defined and extensible system catalogue repository systems utilize an additional layer. Therefore the layer M_3 is introduced allowing for custom-defined M_2 . This however entails the specific problem of consistency between adjacent layers.

Table 1. Layers in OMG MOF Metadata Architecture

Layer	Name	Description
M_3	Meta-meta-model (MOF)	A standardized language, in terms of which definitions of underlying metamodels are expressed.
M_2	Meta-model	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.
M_1	Model	Application Model (Application classes, Table definitions etc.). Alternative term is “information model” [4].
M_0	Data	Instance Data (e.g. objects, records)

Like many other repository and meta-model standards, OMG MOF (Table 1) does not explicitly target the lowest layer containing application instance data. OMG MOF [23] is a meta-meta-model standard. The different layers it defines are shown in Table 1. Self-description here is defined by the fact that the meta-meta-model (e.g. MOF) is an instance of itself, i.e. M_3 -layer definitions are instances of themselves.

Consistency in repository systems has several aspects:

- Operational consistency – deals with the interaction between repository applications and the RMS (see Section 3) and is closely related to the notion of repository transactions. There are two sub-aspects: concurrent multi-client access; and cooperative atomicity [32], i.e. atomic, structurally consistent repository update operations spanning multiple elementary API operations.
- Metadata integrity – comprises the notions of well-formedness and structural integrity. It must be automatically enforced by the repository management system. Well-formedness ensures the syntactical correctness of the model definitions within a meta-layer. Structural integrity (structural consistency [23]) guarantees the conformance of objects on one level to type definitions on the adjacent higher meta-level. Structural integrity results from the strict enforcement of the type-instance relationship across meta-layers. Without structural integrity, repository applications might create or modify metadata artifacts on M_{n-1} inconsistent with respect to their meta-classes on M_n . For example, an application may read the value of an attribute of an object whose meta-object does not exist and is therefore invalid. Structural integrity violations may occur naturally in repository systems since they allow for dynamic modification of M_2 , M_1 and M_0 at run time. Other systems do not face this kind of issues because they assume that the catalogue is static at run time. More examples are discussed in detail in Section 4.

MOF provides several mechanisms for controlling metadata integrity. However none of them describes structural integrity, i.e. propagation of changes to underlying

layers when instances exist, and how it can be implemented. Firstly, MOF defines a set of MOF Model constraints. Secondly, MOF defines a set of closure rules and computational semantics for the abstract mapping, and JMI defines computational semantics for the Java mapping. Thirdly, MOF provides the MOF constraint model element for expressing domain rules. Last but not least, MOF (and JMI) defines a set of repository interfaces. All of the above contribute to well-formedness. The JMI computational semantics defines the so called “lifecycle semantics” for different model elements, describing create, delete or update operations when no instances exist.

This paper discusses various aspects of repository consistency. Its major contribution is the proposed approach to enforcing consistency in OMG MOF-based repositories. Here its implementation [35] is also discussed— the iRM/RMS module of the iRM project [33]. In addition, this paper dwells on some run-time aspects of consistency, which are not touched upon in the current MOF specification version 1.4 [23] such as repository transactions and changes to the MOF reflective package facilitating structural integrity.

The paper is organized as follows: the next section describes the related work; Section 0 describes the architecture of iRM/RMS outlining its key modules and their functionality. Section 0 presents an example motivating the need for consistency in repository systems. It discusses what actions must be taken upon different kinds of repository data modifications and motivates the need for transactional support. Section 0 describes how iRM/RMS implements repository transaction. Section 0 is dedicated to metadata integrity, defining the notions of structural integrity and well-formedness. It also presents the proposed structural integrity algorithm and provides a performance evaluation as well as some considerations. Section 0 draws conclusions from the work described in this paper.

2 Related Work

Repository systems (Unisys Universal Repository Manager [30], CA Platinum Repository, ASG Rochade etc.), metadata repositories (IBM Repository Manager/MVS, IBM AD/CycleManager, Digital CDD Cohesion etc.), and data dictionaries (IBM DB/DC DataDictionary, DataCatalogue etc.), are closely related terms for metadata management systems, among which a significant overlap exists. A key difference between repository systems and the rest is that repository systems store both data (M_0) and metadata. Data dictionaries store primarily structural metadata (type and schema definitions), whereas metadata repositories can handle both structural and descriptive metadata.

There are a number of metadata repository related standards: IRDS [15], PCTE [31], CDIF [6,7], MOF [23]. The ISO/IEC IRDS framework [15] defines a set of related standards [16, 17] defining a Data dictionary system (termed information resource dictionary) operating as a central point of control for a whole enterprise.

MOF (Meta Object Facility) [23] is a standardized, technology independent meta-meta model from OMG. It is gaining industry acceptance with initiatives such as the Model-Driven-Architecture, and in the fields of UML (UML Metamodel, UML Profiles) or data warehousing (OMG Common Warehouse Metamodel). The OMG MOF standard defines an abstract meta-meta model and mappings to a generic OO lan-

guage – CORBA IDL, OMG XMI. JMI (Java Metadata Interface) is a standardized MOF to Java mapping. MOF relies on XMI [24] for exchange (export/import) of meta-models. In addition, MOF defines two kinds of Meta Object Protocols (MOP): a general and completely reflective one in terms of the MOF Reflective API (Package); and a set of generated interfaces, custom-tailored to a concrete meta-model.

There are a number of MOF-based repository implementations: DSTC dMOF [10], MDR [22], CIM (JMI reference implementation) [29], Adaptive Repository Enterprise Edition [1], Unisys Universal Repository Manager [30].

Microsoft Repository [3] is another repository product. It supports only limited consistency (between M_0 and M_1 , M_2 is static). It is currently (Version 3) available as Metadata Services for the Microsoft SQL Server 2000. Microsoft repository implements a standard meta-model – MDC Open Information Model.

The idea of supporting consistency across different meta-levels is not repository system specific - it emanates from the field of computational reflection [21] and Meta Object Protocols [19]. Metadata integrity is a key characteristic of reflective systems [5] supporting different Meta Object Protocols: OpenC++ [8], MPC++ [14], SOM [26]. A number of languages contain built-in reflective facilities supporting MOP with high levels of intercession: CLOS [18], SmallTalk [12], Schema, Lisp etc. All of the above MOPs support intercession, which is the kind of reflection most relevant in the context of metadata integrity. Additional introspective MOPs are also available, e.g. Java Reflection API or RTTI in C++.

To recapitulate – the concept of repository consistency is not new. But it is insufficiently specified in MOF and the existing MOF repositories do not support it fully. The related concept of intercession has a long history and various implementations in reflective systems.

3 Architecture of iRM/RMS

The architecture of a repository system and the tasks its modules perform are defined in [3, 4, 7]. The logical architecture of the iRM/RMS module (Fig. 1) will be briefly described in this section. It provides useful insights as to how the different modules implementing algorithms discussed here interact. The architecture comprises: a repository client, which is a generic library used to build repository applications on top of it; Repository Management System (repository manager [4]), handling the metadata and providing repository clients with various services; a well defined RMS interface (repository API); persistent data store; and import/export utilities. The iRM/RMS API is the API of the repository management system. It is based on the JMI Reflective API [27] and extends it to handle the M_0 data. The Metadata Manager handles the repository metadata organizing it into layers. It implements the JMI Reflective API. The Lock Manager provides isolation by preventing multiple repository clients from modifying the same pieces of repository data concurrently by employing a locking mechanism. The Consistency Manager enforces metadata integrity upon a series of metadata modification operations. The Data Store Manager handles the persistence of the metadata and the M_0 data, both of which are stored in separate data stores due to the significant difference of the data properties. Currently iRM/RMS uses Oracle 9i databases as data and metadata stores.

Fig. 1. Conceptual Architecture of the iRM/RMS

4 Introductory Example

In this section we present an example motivating the need for repository consistency. Fig. 2 shows two meta-models (M_2 : RDB and M_2 :RDBEx) and two models (M_1 :Schema1 and M_1 :Schema2). For simplicity reasons we focus only on the interaction between M_1 and M_2 , but the example may be easily extended to include M_0 . Without structural integrity, for example, the object M_1 :Emp1obj (Fig. 2) may be moved from Schema1 to Schema2, which is clearly an invalid operation since Schema2 is an instance of another package. If the application then deletes the M_2 :RDBEx package and consequently all contained classes and their instances, then M_1 :Emp1obj must be deleted too, which would make links (M_2 :hasAttr instances) to M_1 :Col1obj and M_1 :Col2obj invalid. As a result Schema1 will not conform to the M_2 :RDB model because M_1 :Col1obj and M_1 :Col2obj are not associated with instance of table anymore.

Consider the meta-classes M_2 :Table and M_2 :Column, the containment relationship “has” (Fig. 2) and the multiplicity of its role (its association end) “columns”, which model a rudimentary definition of a database table.

Case 1: The containment relationship “hasAttr” implies that all instances of a M_2 :Table (i.e. type definitions or M_1 instances) must have an element instance of “ M_2 :Column” as part of their composite structure. Therefore the RMS must disallow the creation of an instance M_2 :Column, not associated with an instance of M_2 :Table. This entails two specific problems: (a) the RMS must automatically check whether any existing instance of Column is associated with instance of table when enforcing structural integrity; in case of failure the modifications must be undone. (b) the RMS API object model provides different constructs for creating the different model elements one-at-a-time. For example, first Emp1obj instance will be created, then the Col1obj and the link between them. To solve the above problems we need the concept of repository transactions. Demarcating which RMS operations belong together and must be executed in an atomic manner as a logical unit (problem (b)) is a classical transaction processing problem. Structural integrity (problem (a)) can be enforced in deferred manner, after the end of the transactions. This is the reason why repository transactions are needed.

Case 2: Modification of models must be handled properly; the respective changes must be propagated on all underlying levels. Deleting the generalization relationship between Table and TableEx, for example, would mean that TableEx will not inherit

the attributes Name and colCnt. Enforcing structural integrity must result in the removal of the instances of the attributes Name and colCnt from all instances of TableEx (e.g. Dept1obj).

Case 3: Deleting just the RDBEx package, would force the RMS to automatically delete all of its contained elements (TableEx, Trigger and the association triggers) and their instances when enforcing structural integrity.

Fig. 2. Example of conformance with the metamodel

Existing OMG-MOF repository implementations either completely ignore structural integrity or implement it only partially. The case of adapting existing instances after the meta-model has been changed (or revoking the operation) is handled in none of these products. In the following sections we will describe the structural integrity mechanism implemented in iRM/RMS.

5 Implementing Operational Consistency

As motivated in Section 0, from the standpoint of a repository client consistency in repository systems has multiple aspects, most of which are related to providing trans-

actional support. The concept of repository transactions was first described in [4]. The primary aspects are concurrent multi-client access, and cooperative atomicity. [32] defines cooperative atomicity as “a group of concurrent activities externally viewed as a single action and synchronized with respect to the group as a whole”.

iRM/RMS repository management system is designed to handle concurrent multi-client operations. To ensure isolation the iRM repository API provides a locking mechanism based on long exclusive (X-locks) and short shared (S-) locks. The employed locking mechanism is 2PL compatible. It extends the traditional locking [14, 28] and especially the multi-granularity locking in OODB [32]. We introduce what we call instance lattice to accommodate locks on objects on the multiple meta-levels ($M_0 \dots M_2$).

Table 2. Extended locking table / instance lattice

		Requested		
		X	S	
→	X	–	–	M_2
	S	–	+	
→	X	–	–	M_1
	S	–	+	
→	X	–	–	M_0
	S	–	+	

In principle, changes to a meta-level M_n ($0 \leq n < 3$) affect the structural integrity of all instance models on all the underlying levels. In other words, once a repository client makes changes on M_2 level model, all M_1 instance models and the respective M_0 data, must be altered accordingly by the RMS. Hence the following rule: X-lock on M_n , ($0 \leq n < 3$) will be set if and only if an X-lock can be set on all instance models (and data) on all underlying meta-levels M_{n-p} , ($0 \leq p \leq n$) (see Table 2). X- and S-locks would be set unless a lock is set on all instance models on all of the underlying levels (Table 2) – hence the name instance lattice. Within a single level the traditional locking rules [13] hold. More details may be found in [34].

6 Managing Metadata Integrity

Metadata integrity is a vital property of the repository metadata architecture. As pointed out in Section 1 metadata integrity has multiple aspects, enforced by different modules of the RMS at different times. Metadata integrity is subdivided into well-formedness and structural integrity. While well-formedness guarantees syntactical correctness of the artifacts and models, structural integrity must ensure among other things that definitions on level M_n , $0 < n < 3$, conform to the model on the next higher meta-layer. In the next sections we describe these aspects in detail.

6.1 Well-Formedness of the Metamodels

Well-formedness guarantees the syntactical correctness of the artifacts defined within a single meta-layer at the time they are created. For example, well-formedness ensures whether the created packages, classes, associations etc. have the proper syntax; whether an association is not created with just one association end; or whether an attribute is not defined without a data type. Well-formedness can be enforced ad hoc without reasoning about the next higher meta-layer.

“Syntactical correctness”, enforced by well-formedness, implies a set of fixed rules predetermined by the “hard-wired” MOF abstract syntax. Therefore they are formulated as “constraints” (MOF Model Constraints) in the MOF specification. Some of the syntactical rules have immediate evaluation policy, i.e. after the end of a single modification operation. Satisfying these rules determines the minimum level of repository consistency. In iRM/RMS well-formedness of this type is enforced by the Metadata Manager (Fig. 1) alone. Some examples are given below:

- Enforce the properties of generalization hierarchies and containment hierarchies. For example, no class or package can be defined as super types of themselves; no name conflicts with super-type elements are allowed.
- Check whether containment rules are satisfied. For instance, operations may contain only parameters, constraints and tags; or operations are allowed to have at most one parameter marked as return.
- Check whether miscellaneous properties are properly set. Such properties are, for instance: frozen, root or leaf, singleton for abstract classes. For example, creating sub-classes of a newly created class whose attribute “isLeaf” is set to true should be disallowed.

6.2 Structural Integrity

Structural integrity is the major aspect of metadata integrity and a crucial characteristic of the layered repository metadata architecture. Structural integrity ensures cross-level integrity, i.e. the structure of the objects on a layer M_n conforms to the type definitions on the upper layer M_{n+1} , $0 \leq n < 3$. It concerns changes of M2 or M1 level artifacts, whose instance objects exist on underlying levels. In iRM/RMS structural integrity is enforced automatically by the Consistency Manager (Fig. 1) in deferred manner. Therefore operational consistency (Section 0) is a prerequisite.

Structural integrity is expressed in terms of conformity to the respective meta-model (M_{n+1}) and the so-called structural constraints. Structural constraints (not to be mixed with MOF Model Constraints) are conditions expressed in any constraint language such as OCL.

In section 0 we already introduced some examples as to what should happen when M_2 and M_1 models are modified. Such considerations emanate from topics such as architecture of a meta-language [11, 25] and meta-modeling [9]. Structural constraints, which result from the structural constraints of different MOF model elements, are expressed as follows:

Binary associations express a generic kind of relationship between instances of the connected model elements. Every association comprises two association ends, which are of the type of the connected model elements. Creating new association entails the

creation of new objects and their respective proxy. If an association is deleted then the association ends and all respective instances (called MOF links) are to be deleted, too. Changing the type of an association end requires the new type to be a super-type of the old one. Type incompatibility leads to repository transaction rollback.

Containment hierarchy and Generalization hierarchy are instrumental to the organization of the meta-models and the architecture of a meta-language [13 (p. 23)]. While a generalization hierarchy defines model elements, a containment hierarchy defines nesting and containment rules. Changing the hierarchy means changing the structure of the lower level artifacts. Restructuring existing instance artifacts is complex and cannot be performed in every case. Therefore an attempt to change a hierarchy when instance artifacts exist causes a repository transaction rollback. Such an action is allowed only if no instances exist.

Multiplicities of association ends or attributes define the number of instances of type the association end's type (or attribute type) included in the instance artifact and whether they are optional or mandatory. Changing a multiplicity, therefore, results in checking optionality or the lower-upper bound conditions. The repository transaction is aborted if any of these conditions is violated.

Attributes and attribute types. Attributes of a meta-class serve to define static properties or properties of the instance artifacts, e.g. name, count etc. Composites, i.e. attributes of type MOF classes, or aggregation relationships result in nested structures in the instance artifact. Attributes can be added or deleted. Adding a new attribute results in creating new instance objects and initializing them to the default value (NULL, if none is defined). Adding new static attribute requires that a class definition is changed. The deletion of an instance removes the instance of the attribute associated with all instances of a class. Deletion of a static attribute causes the class definition to change. If an attribute type changes the compatibility between the new and the old data type must be tested. Compatibility of primitive data types is defined in MOF and JMI. If the attribute is of type class then the new attribute type can only be a super-class of the old one. Type incompatibility leads to transaction rollback.

Classes can be created, deleted or altered. Creating a new class yields the creation of a new class object and the respective proxy. Deleting a class means deleting its proxy and all its instances. If the class is a type of an association end, the association end is deleted as a result of the class deletion, which would subsequently lead to the deletion of the whole association. The deletion of the class must fail if it violates a containment or generalization hierarchy.

Packages are generic containers for module elements. Creating a package means creating its object and its type definition (package proxy). Deleting a package means deleting its proxy and all contained elements, their proxies and instances.

An additional consideration results from the fact that a repository is a reflective system [23, 5]. Every repository object must have a respective meta-object. In case of M_0 data – every M_0 data element must have a respective M_1 metadata element. If the meta-objects are missing or changed then the respective objects and all instance objects must be deleted or altered, too. The complementary rule states that upon insertion of a new element to a meta-definition (e.g. an attribute to a class) all instance objects must be extended with values initialized to the default value of the type, and the respective M_0 data elements must be added. Type definitions in repository systems must be dynamic. Some implementation languages, e.g. Java or C++, do not support

dynamically changeable class definitions in contrast to other languages such as Smalltalk or Eiffel. JMI utilizes the concept of proxies to handle repository managed types and instances in repository applications. Upon enforcing structural integrity all proxies must be rebuilt to make the repository application coherent with the repository data. If meta-objects of package, class and association exist without a proxy then their proxies are automatically created.

6.3 Algorithm for Enforcing Structural Integrity

In this section we define (in pseudo-code) the algorithm for enforcing structural integrity implemented by Consistency Manager. For reasons of simplicity we have skipped parts of the algorithm handling elements such as operations or complex data types (e.g. structures or enumerations).

Algorithm 1 triggers structural integrity check for M_2 models, eventually triggering Algorithm 2. Input data is a reference to the package proxy of the respective model. In Algorithm 1 all elements contained in the respective package are enumerated by traversing the containment hierarchy (line 1) and then multiplicities of all attributes or references are checked (3). All deferred well-formedness constraints are enforced (4). Eventually all M_1 instance models are enumerated and for each one a structural integrity check (6) is carried out. Step (6) triggers Algorithm 2.

Algorithm 1: Multiplicity Check

```

performTACCommitChecks( MofPackage rootPkg ){
1:   for all element in contents( rootPkg ) do
2:     for all feature in allStructuralFeatures( metaObject( element ) ) do
3:       checkMutiplicity( feature )
     end for
   end for
4: checkDeferredEvaluationConstraints(rootPkg )
5: checkStructuralIntegrity( getMIPackageExtent(rootPkg) )
6: createMissingProxyObjects(rootPkg )
}

```

Algorithm 2: Structural Integrity

<pre> checkStructuralIntegrity(RefPackage pkg){ 1: if not hasValidMetaObject(pkg) then 2: delete(pkg) 3: return end if 4: if containmentOrGeneralizationHierarchy- Changed(pkg) then 5: throw InconsistencyException("Package has been moved to another Package or is not inherited any more."); end if 6: for all package in refPackagesInPackage(pkg) do 7: checkStructuralIntegrity(package) end for 8: for all class in refClassesInPackage(pkg) do 9: if not hasValidMetaObject(class) then 10: delete(class) </pre>	<pre> 11: continue with next iteration end if 12: if containmentOrGeneralizationHierarchy- Changed (class) then 13: throw InconsistencyException("Class has been moved to another Package or is not inherited any more."); end if 14: for all instance in allOfClass(class) do 15: for all instanceAttr in allAttributes(instance) do 16: if hasBeenDeleted(instanceAttr) then 17: removeFromInstance(instanceAttr) end if 18: if valueCompatibleWithType(instanceAttr) then 19: convertValue(instanceAttr) else </pre>
--	--

```

20:  throw InconsistencyException("Attribute
    value's type has changed and is not
    compatible.");
    end if
  end for
end for
21:  for all classAttr in allAttributes( class ) do
22:    if hasBeenDeleted( classAttr ) then
23:      removeFromClass ( classAttr )
    end if
24:    if valueCompatibleWithType( classAttr ) then
25:      convertValue( classAttr )
    else
26:      throw InconsistencyException("Attribute
        value's type has changed and is not
        compatible.");
    end if
  end for
27:  for all mmAttr in attributesDefinedFor-
    Metaobject( class ) do
28:    if isInstanceScoped( mmAttr ) then
29:      for all instance in allOfClass( class ) do
30:        if not hasRepresentation( mmAttr,
          instance
31:          addToInstance( mmAttr, instance )
        end if
      end for
    else
32:      if not hasRepresentation( mmAttr, class )
        then
33:        addToClass( mmAttr, class )
      end if
    end if
  end for
end for
34:  for all assoc in refAssociationsInPackage(pkg)
    do
35:    if not hasValidMetaobject( assoc ) then
36:      delete( assoc )
37:      continue with next iteration
    end if
38:    if containmentOrGeneralizationHierarchy-
      Changed( assoc ) then
39:      throw InconsistencyException("Association
        has been moved to another Package or is
        not inherited any more.");
    end if
40:    for all link in allLinks( assoc ) do
41:      if not linkEndsMatchAssociationEnd-
        Type(assoc, link) then
42:        throw InconsistencyException("
          LinkEnd does not match
          AssociationEndType");
      end if
    end for
  }
  Type(assoc, link) then
42:    throw InconsistencyException("
      LinkEnd does not match
      AssociationEndType");
    end if
  end for
end for
}

```

6.4 Evaluation

In this section we briefly discuss the performance of iRM/RMS with respect to the consistency implementation. All presented results reflect the combined effect of enforced operational consistency and metadata integrity.

We performed extensive tests to prove experimentally the validity of the proposed algorithm. The tests were constructed to handle various cases (some of which were described in Section 0). The algorithm performed successfully in any of the following cases: changes to the abstraction hierarchies (aggregation and generalization hierarchy); creation, deletion and modification of classes; creation and deletion of attributes; attribute multiplicity checking; changing the data type of an attribute; validity of references; association end multiplicity and type checking; deletion of packages (nested and sub-packages). The experimental results cover fully the required computational semantics described in JMI. In addition the experiments showed that the proposed consistency algorithm behaves correctly.

The performance tests were performed on a Pentium 3, 1.13GHz computer with 512 MB RAM. All measured times are in milliseconds. All tests were performed without data store support. The reason for this is twofold: to discard the influence of issues such as distribution; to avoid the influence of the underlying database system and type of storage schema.

Fig. 3. Creation of M2 and M1 models

The first group of measurements (Fig. 3) shows the performance of iRM/RMS when creating M₂ and M₁ models of different size. Fig. 3.a depicts the performance in the case of a small M₂ model with 5 Classes (total of 8 elements), while varying the number of structurally conform M₁ model instances and checking for consistency. Fig. 3.b depicts the system performance in the case of larger M₂ and M₁ models. An approximately linear dependency between the consistency enforcement performance and the number of checked elements can be seen, as it may be expected.

Fig. 4 shows the system performance when carrying out modification operations on the M₂ model with existing M₁ model instances. Fig. 4.a shows the performance of the structural integrity algorithm, when modifying M₂ model elements, with existing 40 M₁ models with a total of 750 elements. The graph in Fig. 4.a shows that the structural integrity algorithm exhibits acceptable performance on this relatively small set of data.

The most expensive operations are changing an attribute or operation in a M₂ class, which is root of a generalization hierarchy. Fig. 4.b shows the performance of the structural integrity algorithm on the same data set, when the M₂ model is modified, and is not inconsistent with the existing data. Fig. 4.b shows a slight exponential curve. Detecting conflicts with changed attribute types or operation parameter types in containment or generalization hierarchies is the most time-consuming case.

Fig. 4. Modification of M2 models

The test results show that the iRM/RMS consistency implementation exhibits acceptable performance, on the average, based on the small set of measurements and without persisting the repository data in a data store. The results show that the realization is suitable for the typical repository use, which involves mainly read and create operations on repository data on different meta-layers. The results illustrate that validation or updates on repository data involving structural integrity exhibit acceptable performance, which could be improved. iRM/RMS fails to show good performance when enforcing structural consistency on incorrect M_1 or M_2 models. In a realistic environment however significant performance penalty is incurred, predominantly due to the performance of the underlying data store.

Authors of the MOF specification [23] hint at the existence of different degrees of consistency although the specification does not specify any in particular. We distinguish five categories – the lowest denoted by “0”, and the highest being “4” (see Table 3). Each degree requires that all lower number degrees are satisfied.

Table 3. Degrees of metadata integrity

Degree	Description
0	Well-formedness - immediate
1	Transactional support (operational consistency)
2	Well-formedness – deferred
3	Structural integrity
4	Support for MOF Constraints (rules) expressed in a constraint language, e.g. OCL

Only iRM/RMS covers degree 3, whereas the majority of existing implementations cover degree 2. iRM/RMS does not have an OCL support therefore it cannot evaluate dynamically MOF Constraints. This disadvantage prevents metamodel designers from specifying a whole class of structural constraints reflecting the specifics of an application domain on M_2 .

7 Conclusions

In this paper we presented an approach to providing consistency in OMG MOF-based repository systems, which is implemented in iRM/RMS. Repository consistency, has many facets and structural integrity is a major one. It is insufficiently specified and incompletely implemented in existing MOF-based repository systems.

We showed that the concept of repository transactions is needed for enforcing structural integrity in a deferred manner. To reflect the specifics of the repository systems the locking mechanisms known from OODBMS need to be extended. We proposed such an extension.

In this paper we also describe the way structural consistency was implemented in iRM/RMS. We define a general algorithm for enforcing structural integrity, which is the major contribution of the paper.

References

- [1] Adaptive Ltd. Adaptive Enterprise Repository (White Paper). May 2002
- [2] Bernstein, P. Repositories and Object-Oriented Databases. Proceedings of BTW '97, Springer, March 1997
- [3] Bernstein, P., T. Bergstraesser, J. Carlson, S. Pal, P. Sanders, D. Shutt. Microsoft Repository Version 2 and the Open Information Model. Information Systems 24(2), 1999, pp. 71-98.
- [4] Bernstein, P., U. Dayal: An overview of repository technology. Proceedings of the 24th VLDB Conference Santiago Chile, 1998
- [5] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture - A System of Patterns. John Wiley & Sons. August 1996.
- [6] CDIF - Integrated Meta-Model EIA CDIF documents EIA/IS-111 EIA/IS-112, EIA/IS-114, EIA/IS-115
- [7] CDIF CASE Data Interchange Format - Overview EIA CDIF document EIA/IS106
- [8] Chiba, S. A metaobject protocol for C++. In 10th Annual Conference on Object-oriented Programming Systems, Languages and Applications, volume 30 of ACM SIGPLAN Notices, pages 285-299, October 1995
- [9] Clark T., A. Evans, S. Kent: Engineering Modelling Languages: A Precise Meta-Modelling Approach. FASE. pp 159-173. 2002
- [10] Cooperative Research Centre for Distributed Systems Technology (DSTC). dMOF Version 1.1. User guide. 2000.
- [11] Erich, O. Repository Systems Teil 1: Mehrstufigkeit und Entwicklungsumgebung" and "Repository Systems Teil 2: Aufbau und Betrieb eines Entwicklungsrepositoriums" . Informatik-Spektrum, Abstract Volume 22 Issue 4 (1999) pp 235-251 and Abstract Volume 22 Issue 5 (1999) pp 351-363
- [12] Foote, B. R. E. Johnson. Reflective Facilities in Smalltalk-80. SIGPLAN Notices. 1989
- [13] Gray, J., A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, 1994
- [14] Ishikawa, Y., A.Hori, M.Sato, M.Matsuda, J.Nolte, H.Tezuka, H.Konaka, M.Maeda, K.Kubota. Design and Implementation of Metalevel Architecture in C++, MPC++ Approach. Reflection '96 Conference, April, 1996
- [15] ISO/IEC 10027:1990 Information Technology - Information Resource Dictionary System Framework 1990- 06-15
- [16] ISO/IEC 10728, Information Technology - Information Resource Dictionary System (IRDS) Services Interface, April 1993
- [17] ISO/IEC 13238-3, Information Technology - Data Management - Part 3: IRDS export/import facility, Dec. 1998
- [18] Kiczales, G., J. Rivieres, D. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991
- [19] Kiczales, G., J.M. Ashley, L. Rodriguez, A. Vahdat, D. G. Bobrow, Metaobject protocols: Why we want them and what else they can do. In Object-Oriented Programming: The CLOS Perspective, pages 101 - 118. MIT Press, Cambridge, MA, 1993
- [20] Lefkovits, H. IBM's Repository Manager/MVS, Wellesley MA:QED Information Sciences. 1991
- [21] Maes, P. Concepts and experiments in computational reflection. Conference proceedings on Object-oriented programming systems, languages and applications. pp. 147 - 155. 1987
- [22] Matula, M. NetBeans Metadata Repository (White Paper). March 2003
- [23] Object Management Group: Meta Object Facility Specification Version 1.4.
- [24] OMG. XMI - XML Metadata Interchange Specification. Version 2.0. OMG Document formal/03-05-02, May 2003

- [25] Ortner, E.. Wissensmanagement Teil 1 und 2 Informatik-Spektrum, Volume 23 Issue 2 (2000) pp 100-108
- [26] S. Danforth, I. R. Forman. Reflections on metaclass programming in SOM. Proceedings of the 9-th Conference on Object-oriented programming systems, language, and applications. pp. 440-452. 1994
- [27] SUN. JMI - Java Metadata Interface Specification Version 1.0, June 2002
- [28] Traiger, I.L., J. Gray, C. A. Galtieri, B. G. Lindsay Transactions and consistency in distributed database systems. ACM Transactions on Database Systems (TODS), Volume 7 Issue 3, September 1982
- [29] Unisys Corporation. JMI-RI Documentation. CIM Guide. Version 1.3. October 2002
- [30] Unisys Universal Repository Manager, <http://www.unisys.com/marketplace/urep/>
- [31] Wakeman, L., J. Jowett. PCTE - The Standard for Open Repositories. Prentice-Hall. May 1993
- [32] Ozsu, M., Tamer. Transaction Models and transaction management in Object-oriented database management systems, in Advances in Object-oriented Database Systems, Editors: A.Dogac, M.Tamer Ozsu, A.Bilris and T. Sellis, Series F: Computer and System Sciences, Vol. 130, 1994, Springer Verlag, New York.
- [33] Petrov, I., Stefan Jablonski, An OMG MOF based Repository System with Querying Capability - the iRM Project. Proceedings of the iiWAS Conference, September 2004
- [34] Petrov I., Stefan Jablonski, Marc Holze, Towards efficient locking of repository objects. Proceedings of IADIS Conference, October 2004.
- [35] Petrov I., Stefan Jablonski, Marc Holze, Gabor Nemes, Marcus Schneider, iRM: An OMG MOF Based Repository System with Querying Capabilities. Demo Paper. ER Conference, November 2004.