Typed Compound Object Models for Digital Library Repository Systems

(*ISTI Technical Report 2008-TR-023*)

Leonardo Candela, Donatella Castelli, Paolo Manghi,

Marko Mikulicic, Pasquale Pagano

{candela, castelli, manghi, mikulicic, pagano}@isti.cnr.it

Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo"

Centro Nazionale delle Ricerche

Via G. Moruzzi, 1 - 56124 Pisa - Italy

Abstract

This report presents a Compound Object data model for Digital Libraries whose novelty is to re-introduce the notion of types as a mean of providing more optimized and efficient implementations of Repository Systems.

Table of Contents

Table of Figures

1 Introduction

Compound Objects are generically intended as sets of digital objects, associated with each other according to relationships that express the form of their relations. Compound Object models differ by the modeling primitives they offer, i.e. the way they characterize the nature of the digital objects and the relationships into play, and thus by the kind of *Document Models*, i.e. digital document structure, they can represent. A number of Compound Object models have been defined, whose features range from very flexible, where the model can potentially describe any document model (e.g. Fedora), to very strict, where the model defines one specific document model (DSpace).

The Compound Object model proposed in this report aims at:

(i) Capturing the modeling abstractions required to describe document models of any Digital Library application domain;

(ii) Inspiring the design of Repository Systems supporting efficient and optimized storage and search of such objects.

The novelty of the model is that of being *statically-typed*, in the traditional database sense.[1] The Repository Systems implementing the model will support construction of DL databases based on the static definition of the structure of the compound objects involved. By exploiting static information about the object structure (so called "object types"), the Services will offer their safe, optimized and efficient management.

[1] The type of an object is the description in a formal language of the object structure. When a data model imposes that objects must be preceded by the definition of their type, the model is called statically-typed.

2 The Model

The data model we propose is presented in two stages. First we introduce a *low-level typed data model*, capable of representing typed graphs of complex objects. The model is flexible enough to provide DL content designers with basic primitives for the definition of customized document models for Compound Objects. In the second stage, we introduce a *high-level typed data model*, whose primitives are expressed in terms of the low-level model primitives and offer common DL-abstractions for Compound Objects, such as versioning, aggregation, collections, etc. The high-level model should be interpreted as modeling syntactic sugar for DL-content designers, who could anyway express the same document model structures through the low-level primitives of the language, but at a finer and therefore more complex granularity.

The main difference between the low-level data model and the high-level data model is that the former is meant to be “minimal”, while the latter can be in principle be extended and modified according to the DL designers needs. The low-level data model provides the minimal set of primitives required to design efficient Compound Object DLs, i.e. removing one of such primitives would compromise the expressivity of the language and leave a subset of our application domain out of our solution domain. The high-level model offers modeling primitives that can be expressed in terms of the low-level primitives and can be therefore seen as add-ons, i.e. extensions to the core model.

3 Low-level Model

3.1 Data Model Primitives

The low-level model supports the notions of *Type*, *Set* and *Object*. Types define the *abstract* structure and the operators of the Object entities in our data model. Sets are instead the instantiation of Types, i.e. the *concrete* type, of the Objects they contain. More specifically, the relation between the three entities is:

- Types can be *instantiated* to create new Sets conformant to the type properties; A Set is therefore the result of the instantiation of one Type and also acts as "generator" of Objects of that Set.
- Sets have unique names and are referred to add, delete or update the Objects therein; in that sense, each Set defines a new unique *Named Type* for all Objects it will contain, whose structure and operators will be that of the Set's Type; Sets are therefore (possibly empty) containers of "structurally homogenous" Objects.
- Objects have unique identifiers and they conform to the Name Type of the Set through which they were created; Objects can be casted to belong to other Named Types, i.e. to other Sets with "compatible" Types.

Figure 1 depicts how a Type *T* can be instantiated in a number of Sets, here with unique names *A* and *B*. Such Sets will have type *T* (*A::T*) and will be able to generate and contain Objects *o* with named types T_A and T_B respectively ($o:\{T_A\}$ and $o:\{T_B\}$). Objects in *A* and *B* share the same structure and behavior, but belong to different Types. We shall see how Objects can be casted to belong to different Sets of "compatible" Types.

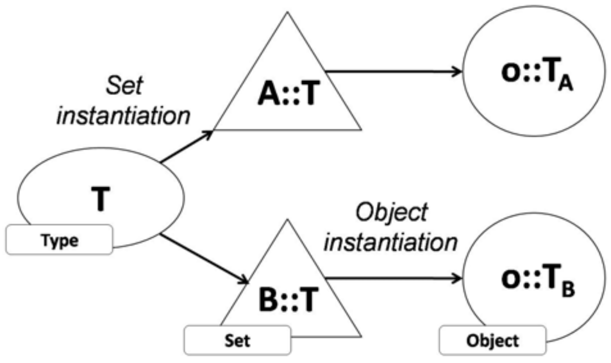

Figure 1 – Relationships between Types, Sets and Objects

Sets can be of three main Types:

- *Atom Type*: Atoms are intended as "simple digital objects", i.e. files or URN references to files. A Set of atoms can be specialized to contain Objects of a specific format/media;
- *Description Type*: Descriptions are human/machine readable descriptions of digital or physical entities in the real world. As such, Descriptions can be given in terms of "properties", i.e. attributes-value pairs, or more complex structures, such as trees of records;
- *Relations Type*: Relations represent binary relationships between Objects of two given Sets, i.e. they consist of two pointers at two existing objects in the given target sets. As all other objects, relation objects have an identity; unlike other objects, relation objects cannot exist without the relative target Objects to exist.

The Object Model attempts to capture the essence of modern Digital Libraries, whose content may be the result of the combination of new content with existing content, in turn possibly heterogeneous and not locally available. In this scenario, Atom objects might be available from different sources and might come or not come with a human/machine readable "description"; e.g. metadata description. Equally, Description objects might exist in the system without for the objects they describe to be present; e.g. metadata catalogs. Finally, given a DL populated with objects, relations between such objects are and should be defined independently by DL communities, based on their needs of connecting the objects. This is why relation objects are independent from the objects they connect, i.e. they are always added in a "second stage" and can be removed with no implicit impact on the objects they were binding together.

Figure 2 depicts the relation between Set Types, where the triangles represent Sets and the other shapes the different kinds of Objects therein. The picture illustrates how Relation, Atom and Description Sets are all specializations (can be casted to) of Object Sets while Relation Sets depend on two other Sets, and relation Objects relate two Objects in two such Sets. In particular, for any object *o* (be it *a* or *r*) that is "linked" to a description object *d* through one relation object *r*, we say that *o is described by d*, or that *d is a description of o.*

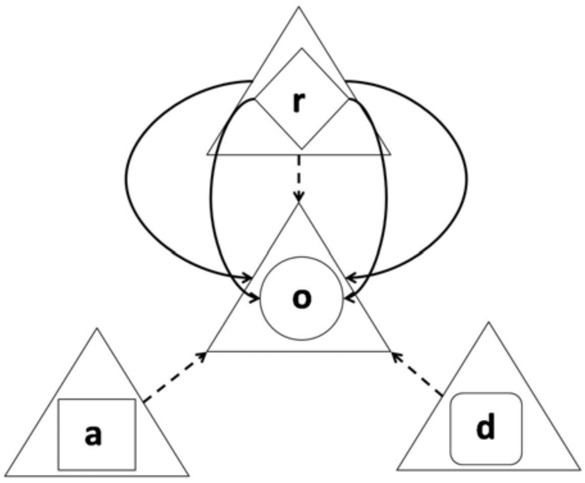

Figure 2 – Object type's interrelation.

3.2 Type Definition Language (TDL)

A Type definition declares the structure and nature of a Set of Objects obtained as instantiation of the Type (see Data Definition Language). The TDL enables the definition of types by appropriately combining type declarations, i.e. type variables assigned to type definitions, according to the abstract syntax in Figure 3.

```
T  ::=  (X = T),T'              // type variable declaration
        | atom(F)               // atom type
        | des(D)                // description type
        | obj                   // generic object type
        | rel(A,B,M,TP)         // relation type
        | union(A1, … ,Ak)      // union type
        | ε
F  ::=  pdf | xml | avi | other file formats…
M  ::= 1:1 | 1:N | N:1 | M:M    // relation multiplicity
```

```
TP ::= t:p | p:t | p:p | t:t      // relation partiality/totality
D  ::=  [l1:K1, … lk:Kk]          // description type
K  ::=  D                         // nested description type
        | coll(D)                 // collection type
        | int | string | date | bool | others…
```

Figure 3 – Type Definition Language

Type semantics (informal)

`atom(F)`

When instantiated, the Type defines a Sets of Atom Objects of type (format) `F`. Note that the same Set may contain Atoms of different formats (`F,F`). Moreover, although not specified by the grammar, each format can be accompanied by parameters specifying further static physical customization of the format; e.g. compression, max size, etc.

`des(D)`

When instantiated, the Type defines a Set of Objects that are descriptions of others according to the *description type* `D`. The Type $[l_1:K_1, \ldots l_k:K_k]$ declares a Set of description objects with unique identity and a set of properties with name l_i and value v_i of type K_i. The type `coll(D)` defines a set of values of type `D`, which has to be at least one level below a record type (we simplify collection treatment, by enabling collection evaluation through a record label). Description objects are strongly typed and are used for internal search purposes and the like.

`rel(A,B,M,TP)`

The Type depends on two Sets `A` and `B`. Such a type, when instantiated, defines a set of Relation Objects linking object pairs in the two sets `A` and `B`. Special constraints can be defined on the relation objects, regarding *multiplicity* and *partiality* of the relation set. Given the Sets `A` and `B`, then `1:1`, `1:N` and `N:M` state that the relation between their objects must be one-to-one, one-to-many or many-to-many, respectively. Besides, the constraints `p:t`, `p:p` or `t:t` define the *partiality* or *totality* of the relation in one direction and the other.

$\texttt{union}(A_1, \ldots, A_k)$

The Type depends on *k* Sets A_i. Such a Type, when instantiated, defines the Set of Objects that contains the union of all Objects in the Sets $A_1, \ldots, A_k$.

`obj`

It is a special type, whose objects represent the existence of a fact or concept in the real world; objects of all types can be casted-generalized to `Object` type.

Type Equivalence

Two types are equal if they share the same structure.

Type Compatibility

If `T` is *compatible* with `T′` (`T <: T′`), then the objects in sets of type `T` can be potentially used in context where objects of sets of type `T′` are expected. Compatibility is determined by the structure similarity relation <: (see Figure 4) and is the property required to enable the *cast* operation in the DML (see Figure 7).

```
     T = T′
--------------------
     T <: T′
```

$$\mathtt{T} <: \mathtt{obj}$$

$$\frac{\mathtt{D} <: \mathtt{D'} \text{ or } \mathtt{D} = \mathtt{D'}}{\mathtt{des(D)} <: \mathtt{des(D')}}$$

$$\frac{\text{for all } t:\{1\ldots k\} \text{ exists } q:\{1\ldots s\} \text{ such that } l'_t = l_q \text{ and } K_t <: K_q}{[l_1:K_1,\ \ldots\ l_k:K_k] <: [l'_1:K'_1,\ \ldots\ l'_s:K_s]}$$

$$\frac{\mathtt{D} <: \mathtt{D'}}{\mathtt{coll(D)} <: \mathtt{coll(D')}}$$

$$\frac{\mathtt{F} <: \mathtt{F'}}{\mathtt{atom(F)} <: \mathtt{atom(F')}}$$

Figure 4 – Compatibility relation `<:`

3.3 Data Definition Language (DDL)

The core of DDL enables the creation of sets of objects with a given type (see Figure 5).

```
C ::=   C;C
        | A = create T           \\ instantiation of a set A of type T
        | delete A               \\ deletion of a set A and its objects
        | ε
```

Figure 5 – Data Definition Language

The informal semantics of such commands is the following.

`A = create T`

The command generates a new Set with name `A` of type `T (A::T)`. Note that a subsequent command `B = create T` would generate another Set from the same type. Although the type, i.e. the underlying structure, is the same, objects in `A` could not be used in contexts where objects in `B` are expected. The objects o_A in `A` and o_B in `B` would have named types $o_A:\{T_A\}$ and $o_B:\{T_B\}$.

3.3.1 Example Proceedings of Articles

In the following we show the steps required to define a Digital Library storing Conference Proceedings. In our example, Proceedings are sets of scientific articles in PDF format, where both proceedings and articles are described by Dublin Core records. Proceedings are therefore compound objects, whose existence represents a set of articles accepted to a given conference.

The type definitions are the following:

```
ProceedingType = obj();
```

```
ArticleType = atom(PDF);
DCType = des([DCField1:DCFieldType1,…, DCField15:DCFieldType15]);
```

where the `DCFieldk`'s are the fields of the Dublin Core metadata format. Through such types we can define the following Sets

```
Proceedings = create ProceedingType;
ProceedingsDC = create DCType;
ProceedingsMetadata = create rel(Proceedings, ProceedingsDC,1:1,t:t);

Article = create ArticleType;
ArticleDC = create DCType;
ArticleMetadata = create rel(Article, ArticleDC,1:1,p:t);

ProcArticle = create rel(Proceedings,Article,1:n,p:t);
```

Figure 6 exemplifies a possible instance of the DLDB resulting from this Set instantiation. Note how Proceedings can exist without related articles and how articles can exist without a corresponding DC description.

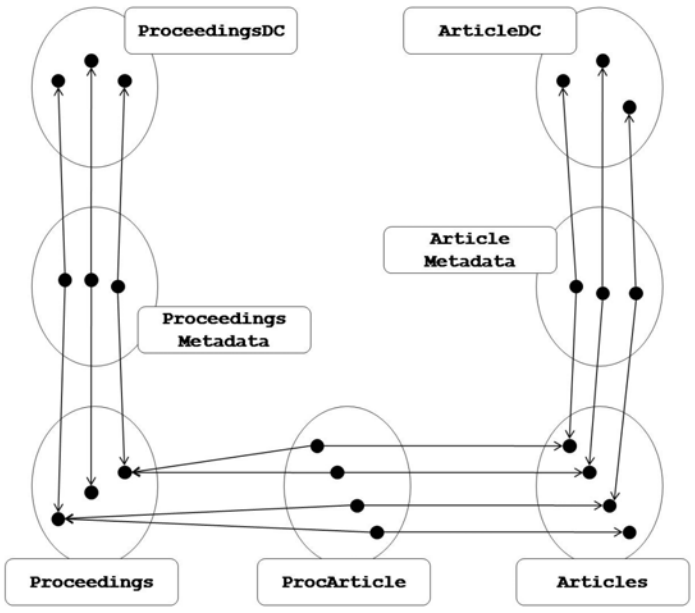

Figure 6 – Graphical representation of the Proceedings DL

The first three commands create the set of objects `Proceedings`, a special set `ProceedingsDC` for the relative description objects and then a relation object set `ProceedingsMetadata`, which will include the relation objects linking proceedings to their descriptions. Note that the constraints on such relation objects are: (*i*) from `1:1,` there cannot be more than one object referring to one object in `Proceedings` or in `ProceedingsDC`; *(ii)* from `t:t`, there cannot be a `ProceedingsDC` object without one related `Proceedings` object and viceversa.

Similarly, the subsequent three commands define a set scenario where PDF files can be created and associated to their metadata objects. In this case, however, `Article` objects can also exist without the relative `ArticleMetadata` object. Finally, a relation set `ProcArticle` between `Proceedings` objects and `Article` objects is created, according to which all `Article` objects must be associated to one `Proceedings` object.

The DDL can be extended with special clauses for the customization of the physical storage. For example description object sets can also include the indexing features, if any, to be applied; atom sets can specify the compression algorithm to be applied or the kind of access granted to the file format involved. The physical storage management will be dealt with in a later stage of design.

3.4 Data Manipulation Language (DML)

The core of the DML (see Figure 7) enables the creation/deletion/update/casting of objects into/from the sets created in the DDL.

```
C ::=   C;C
        | x = new A(pars)         \\ instantiation of an object in A
        | A.cast(o)               \\ object o cast
        | A.drop(o)               \\ deletion of an object o from A
        | A.update(o,pars)        \\ update of an object o w.r.t. A
        | {C}                     \\ ACID transaction
        | ε
```

Figure 7 – Data Manipulation Language

`x = new A(pars)`

The command generates a new object of type T_A $(x:\{T_A\})$, where A::T, assigns it to the variable x of type T_A, and inserts it into A. Note that `pars` is a set of parameters whose number and structure depends on the nature of the type T:

- `T=atom(F)`: here `pars` is a pair `(object, mode)` where `object` is an URN to a file and `mode` can be either `reference` or `payload`. In the first case the object stores the file, i.e. the system uploads it from the URN; in the second case only the URN is stored with the object.
- `T=des(D)`: here `pars` is a value expression of type D.
- `T=rel(A,B,M,TP)`: here `pars` is a pair `(fst, snd)` where `fst` and `snd` are respectively objects in the two sets A and B.
- $T=union(A_1, \ldots A_k)$: here `pars` is a pair $(pars, A_i)$ where `pars` is the set of parameters required to create an element of the set A_i.

`A.cast(o)`

The command declares that the object $o:\{T_1,\ldots,T_k\}$ also belongs to the set A, thus $o:\{T_A, T_1,\ldots,T_k\}$. This is possible only if for all k: T_k <:: T_A. The *castability* relation <:: is defined in Figure 8, based on the compatibility relation in Figure 4.

```
T <: T', A = new T, B = new T'
```

$$\frac{}{T_A <:: T_B}$$

$$\frac{k>s,\ \text{for all } t:\{1\ldots k\}\ \text{exists } q:\{1\ldots s\}\ \text{such that } T_{At} = T_{Aq}}{\text{union}(A_1,\ \ldots\ A_s) <:: \text{union}(A_1,\ \ldots\ A_k)}$$

$$\frac{\text{exists } q:\{1\ldots s\}\ \text{such that } T_B <: T_{Aq}}{T <:: \text{Union}(A_1,\ \ldots\ A_q)}$$

Figure 8 – Castability of Objects

`A.drop(o)`

The command removes the object $o:\{T_A, T_1, \ldots, T_k\}$ from `A`, thus making it of type $o:\{T_1, \ldots, T_k\}$. If T_A is the only type of `o`, then `o` is removed from the system.

`{C}`

The command executes a sequence of commands in one transactional step. The clause ensures that all commands it contains are executed or none. It is typically used to ensure that relations set constraints are respected. Transaction side effects and constraints on Sets are verified when the transaction is committed. If a transaction violates Set constraints, the state is rolled back to the one before the transaction.

For example if we want to insert a new article to a proceeding object `o`, we should execute the following transaction:

```
{a = new Article(URN,payload);   \\ URN: ref to the article
DC = new ArticleDC(v);           \\ v: DC record for the article
r1 = new ArticleMetadata(a,DC);  \\ creation of relation r1
r2 = new ProcArticle(o,a);       \\ creation of relation r2
};
```

which creates the object `a` in `Article`, creates the relative description object `DC` in `ArticleDC`, creates the relation `r1` between the article and its metadata in `ArticleMetadata`, and finally creates the relation `r2` between the proceedings `o` and the article `a` in `ProcArticle`.

Constraint integrity checking
Constraint checking is performed at every execution step, i.e. execution of individual commands `C`. If the check returns an error, the last operation is unrolled and an error reported for exception handling.

3.5 Data Query Language (DQL)

DQL queries, given a set of objects, return a set of objects satisfying certain conditions. Queries are type-checked against the Set types, i.e. their "correctness" can be determined before execution. The Data Model's DQL offers two query typologies, which can be combined to perform complex navigation patterns:

- *Target-oriented queries*: the query returns the set of objects that are reachable from a given initial set, based on some navigational/conditional criteria; e.g. "return all Articles that are reachable from Proceedings published in year 2007".

- *Source-oriented queries*: the query returns the subset of objects in a given source set that satisfy some navigational/conditional criteria within the graph of objects; e.g. “return all Proceedings that contain an Article published in year 2006”.

The former kind selects objects from a source set, while the second kind returns objects that are reachable from the initial set. In order to express the semantics of the query language, we first need to introduce some definitions.

Definition *(Relationship) o* is in relation *R* with *o’* and vice versa (o $\leftrightarrow^{R}$ o’) if and only if there exists r$\in$R such that *R* $\in$ *rel(A,B,M,TP)*, *r.A = o* and *r.B = o’* (o $\leftrightarrow$ r $\leftrightarrow$ o’).

Definition (*Reachability*) *o* is reachable from *o’* and vice versa (o $\leftrightarrow^{*}$ o’) if and only if there exist $R_1,\ldots,R_k$ $\in$ *rel(…)* such that o $\leftrightarrow^{R1}$ o_1 $\leftrightarrow^{R2}$ o_2 … o_{k-1} $\leftrightarrow^{Rk}$ o’.

Figure 9 shows DQL’s grammar.

```
Q   ::=  Q!L                     // target-oriented navigation
         | Q?L                   // source-oriented by navigation
         | Q|label               // relation-oriented navigation
         | label                 // Sets of objects
         | ε
L   ::=  L/label | L/*
         | L//label | L//*
         | L[P]
         | ε
P   ::=  (P) | not P | P Bop P | V Con V | inSet(A) | ofType(T) | count()
Con ::=  > | < | =
Bop ::=  And | Or
V   ::=  Des | At
Des ::=  [l1:Val,…,lk:Val]
Val ::=  De | {De,…,De} | v        // v is Int, String or Bool
At  ::=  .label At | ε            // label: Set name, description label
                                  // or atom attribute
```

Figure 9 – Data Query Language

The semantics of the language is expressed in terms of relationship and reachability relations, through the functions *Sem(Q)* and *sem(L)(Θ)*, where *Θ* is a set of Objects.

Sem: Q $\rightarrow$ $\wp(\Theta)$, where Θ is the Set of Objects.

Sem(Q!L) = sem(L)(Sem(Q))

Sem(Q?L) = {o $\in$ Sem(Q) | $\exists$ o’ $\in$ Sem(Q!L): o $\leftrightarrow^{*}$ o’}

Sem(Q|label) = {r $\in$ label | $\exists$ o $\in$ Sem(Q): o $\leftrightarrow$ r $\leftrightarrow$ o’}

Sem(A) = {o $\in$ A}

$\mathrm{Sem}(\varepsilon) = \varnothing$

Sem: $\mathtt{L} \times \Theta \rightarrow \wp(\Theta)$

$\mathrm{sem}(\mathtt{L/label})(\Theta) = \{o \mid \exists\, o' \in \mathrm{sem}(\mathtt{L})(\Theta): o' \leftrightarrow^{\mathtt{label}} o\}$

$\mathrm{sem}(\mathtt{L/*})(\Theta) = \{o \mid \exists\, \mathtt{label} \wedge \exists\, o' \in \mathrm{sem}(\mathtt{L})(\Theta): o' \leftrightarrow^{\mathtt{label}} o\}$

$\mathrm{sem}(\mathtt{L//*})(\Theta) = \{o \mid \exists\, o' \in \mathrm{sem}(\mathtt{L})(\Theta)\ o' \leftrightarrow^{*} o\})$

$\mathrm{sem}(\mathtt{L//label})(\Theta) = \{o \mid \exists\, o' \in \mathrm{sem}(\mathtt{L})(\Theta) \wedge o'': o' \leftrightarrow^{*} o'' \leftrightarrow^{\mathtt{label}} o\})$

$\mathrm{sem}(\mathtt{L[P]})(\Theta) = \mathrm{sem}(\mathtt{P})(\mathrm{sem}(\mathtt{L})(\Theta))$

$\mathrm{sem}(\mathtt{P})(\Theta) = \{o \in \Theta \mid o \text{ satisfies } \mathtt{P}\}$

The definition of sem(P)(Θ) is standard and therefore not fully expressed here.

The query Q!L returns all objects reachable through the navigation path L from the set of Objects Q. The query semantics matches standard X-Path navigation behavior.

The query Q?L returns all objects in Q for which there exists an object satisfying the navigation path L; note that such semantics is expressed in terms of Q!L, which is the set of L-reachable objects from Q.

The query Q|label returns all relation objects in the Set named label that are reachable from objects in Q. In the model, relationships are represented as objects, which are themselves navigable from others through such special query clauses.

Navigation paths can introduce predicates, in order to skim the objects they reached in the last navigation step according to some conditions they respect. The predicate language introduces simple logical algebra and a value language for objects of type atom, description, relation and union; special functions capable of verifying object structure properties, such as Set membership (inSet(A)) and type conformance (ofType(T)), are also provided.

Sample queries:

- “Find all proceedings having one paper whose author is Yannis Ioannidis”

```
Proceedings?ProcArticles[author=”Yannis Ioannidis”]
```

- “Find all articles about computers in proceedings of year 2007”

```
(Proceedings?ProceedingsMetadata[year=2007])!ProcArticles[subject=”Com
puter Science”]
```

4 High-level Model

The high-level data model offers primitives, i.e. Types, that aim at representing Compound Objects best practices at a coarser granularity. The structure and operators of such Types are expressed in terms of the low-level model type and set primitives. As such they could be seen as add-ons to the core data model, which can be then further extended, varied and customized in the future by DL designers with further primitives according to the same principles shown in this section.

4.1 TDL and DDL extensions

The TDL is extended with the following Types, matching common abstractions in the DL world:

```
T  ::=  objDes(T,D,Pt)
        | aggregation(A,Tp)
        | version(T)
        | annotation(A)
Pt ::=  p:t | t:t
Tp ::=  p:p | t:p
```

Figure 10 – Extended Type Definition Language

`objDes(T,D,M,Pt)`

This Type models the common Digital Library pattern according to which “digital objects are described by human and/or machine readable metadata information”. In that sense, this notion matches the more generic pattern of objects with properties, which are instead represented as separated concepts in the low-level model. The type “hides” the definition of the relation set required to associate objects of type `T` to the respective descriptions of type `D` through an intermediate relation set.

`A = objDes(T,D,Pt)`, where `A` is a set, `D` is a description type, executes the following DDL-core transaction:

```
{A = create T;
Desc_of_A = create des(D);
BlendingRel = rel(A,Desc_of_A,1:1,Pt);
};
```

Sets of this type contain objects of type `T` that also inherit the properties defined by `D`. For this “blending” operation to be consistent, the type `T` should not inherit from another `objDes` type (or own itself) properties whose labels are equal to those in `D`.

`aggregation(A,TP)`

The Type models the common DL pattern of objects that are used to aggregate others according to some application domain set-logic; e.g. a proceeding object aggregates a number of article objects. Aggregation objects are expressed as `des` type objects in relation with objects in the set `A` according to the partiality criteria `Tp` (this criteria leaves the set `A` independent from the set `B`, by enforcing partiality from `A` to `B`); such objects feature the `cardinality` property, stating the number of objects present in the aggregation. The type “hides” the definition of the object, its properties and the relation Set required to satisfy such pattern and introduces special extra methods in the DDL and DQL languages for aggregation object manipulation.

`B = aggregation(A,Tp)` executes the following DDL-core transaction:

```
{B = create des([cardinality:int]);
AggregationRel = rel(B,A,1:n,Tp);
};
```

`version(T)`

The Type models the common DL pattern of “versioning objects”, which are capable, at any update, to keep the history of old versions. Special operations are possible on such objects, in order to add, remove, update and search their different versions.

`A = version(T)` executes the following DDL-core transaction:

```
{A = create obj;
 VersionSet = create T;
 VersionInfoType=des([vers_name:string;
                      vers_number:int;
                      vers_date: date]
                      );
 VersionRelation =
    create objDes(rel(A,VersionSet,1:n),VersionInfoType,t:t)
};
```

Figure 11 exemplifies the generic instantiation of a version type Set. Applications interact with the Set `A`, unaware of the manipulation of the object Sets created to support it.

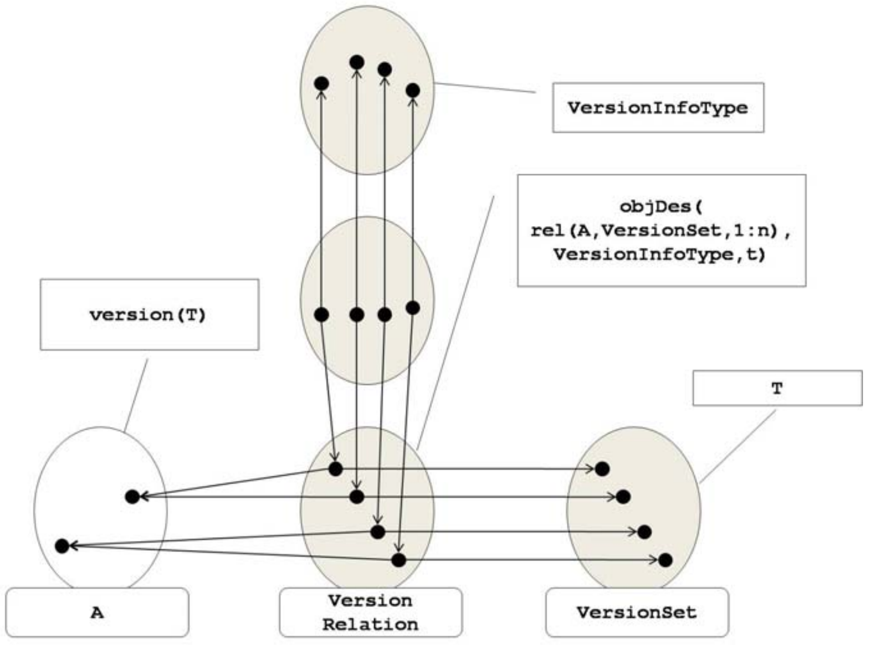

Figure 11 – Example of Version Type instantiation

`annotation(A,M,Tp)`

The Type models the common DL pattern of “object annotations, meant as special text or notes to be attached to objects of given set `A`. Annotations may be mandatory or not (`Tp`) and can be one or more per object in `A` (`M`).

`B = annotation(A,M,Tp)` executes the following DDL-core transaction:

```
{B = create des([ann_owner:string;
                ann_text:string;
                ann_creation_date: date;
                ])
AnnotationRelation = create rel(B,A,M,Tp)
};
```

4.1.1 Example Proceedings of Articles

The example proceedings-article in Section 3.3.1 can be rewritten in the high-level data model as follows:

```
DCType = des([DCField1:DCFieldType1,…,DCField15:DCFieldType15])
Articles = objDes(PDF,DCType,p);
Proceedings = objDes(aggregation(Article),DCType,t);
```

It is clear how high-level abstractions eased the definition of the same document model. The graphical representation of the example is shown in Figure 12, where the grey circle represent the hidden sets, whose management and consistency is delegated to the operators of the `objDes` Type' Sets. For example if we want to insert a new article to a proceeding object o, we should execute the following transaction:

```
{a = new Article(URN,payload,d);
            \\ URN: ref to the article
            \\ d is a value of type DCType
Proceedings.addObj(o,a);
            \\ operator for adding objects to aggregation objects
};
```

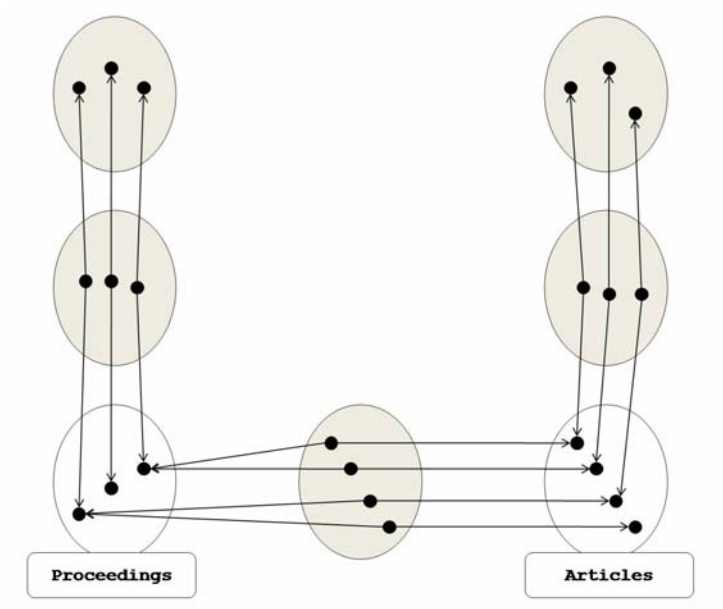

Figure 12 – Example Proceedings-Articles in the high-level data model

4.2 DML and DQL extensions

DML and DQL language have to be extended to support operators and query semantics over the new type abstractions.

`objDes(T,D,Pt)`

In the DML, for `A = objDes(T,D,Pt),` objects are created and updated with parameters required by `T` and `D`, where parameters for `D` can be optional if `TP = p:t`. Creation of new objects has the following transactional semantics:

`x = new A(pars,d)` executes the transaction

```
{x = new A(pars);
dOfx = new(d);
r = hiddenRel(x,d)
};
```

Similarly, in the DQL, the semantics of an object `o:objDes(T,D,Pt)`$_A$ is as if the object `o` could respond to dereference expressions for both `T` and `D`.

Removal of such objects has the following transactional semantics:

`A.drop(o)` executes the transaction

```
{Desc_of_A.drop(o.BlendingRel);
A.drop(o)
};
```

`aggregation(A,Tp)`

In the DML, for `B = aggregation(A,Tp),` objects are created and updated with parameters required by `T` if `Tp = p:p`. Creation of new objects has the following transactional semantics:

`x = new B(o)`, where `o` is of type `A`, executes the transaction,

```
{x = new B([cardinality=0]);
};
```

The removal of an aggregation object in `B` causes the removal of all relation objects connecting the object to objects in `A`.

The addition of an object `o'` of type `A` to an aggregation object `o` of type `B` has the following transactional semantics:

`B.addObj(o,o')` executes the transaction,

```
{rel = new AggregationRel(o,o')
B.update(o,[cardinality=o.cardinality+1]);
};
```

Similarly, the operation `B.addObj(o,o')` removes `o'` from the aggregation object `o` in `B`. The operation causes the elimination of the relation object connecting them and decreases by one the `cardinality` property of the object `o`.

The command `B.getObj(o)` returns the set of objects in `A` aggregated by `o`.

`version(T)`

In the DML, for `A = version(T),` objects are created and updated with parameters required by `T` plus a name for the current version. Creation of new objects has the following transactional semantics:

`x = new A(pars,version_name)` executes the transaction

```
{x = new A;
y = new VersionSet(pars)
rel = new VersionRelation((x,y),
                        [vers_name = version_name;
                         vers_number = 0;
                         vers_date = Date()]
                        );
};
```

The update of version objects causes the creation of a new version of the same object. In particular:

`A.update(o,pars,version_name)` executes the transaction

```
{y = new VersionSet(pars)
 rel = new VersionRelation((x,y),
                        [vers_name = version_name;
                         vers_number = o.vers_number+1;
                         vers_date = Date()]
                        );
};
```

Version objects respond to particular operators:

- `A.getVersionByDate(o,dateRange)`: the command returns the set of objects whose version was release in the given date range;
- `A.getVersionByNumber(o,versNumberRange)`: the command returns the set of object versions in the given number range;
- `A.removeVersion(o,number)`: the command removes version at position `number`; the position numbers of the remaining version objects are adapted to restore the sequence.

`annotation(A,Tp)`

In the DML, for `B = annotation(A,Tp),` objects are created and updated with parameters required to set the annotation properties for a given object in `A`. Creation of new objects has the following transactional semantics:

`x = new B(annOwner, annText, o)` executes the transaction

```
{x = new B([ann_owner = annOwner;
           ann_text = annText;
           ann_creation_date = Date()
           ];)
rel = new AnnotationRelation(x,o);
};
```

The removal of an annotation object in `B` consists in the deletion of the corresponding description object and in the relation objects that connect it to objects in `A`. Annotation objects respond to special operators:

- `B.getAnnotationsByObject(o)`: the command returns the annotation objects relative to the object `o` in `A`.
- `B.getAnnotations(owner,dateRange)`: the command returns the annotation objects created by the given entity `owner` in the given date range.