A Typed Repository for OpenDLib

Leonardo Candela Donatella Castelli Paolo Manghi Pasquale Pagano
Istituto di Scienza e Tecnologie dell'Informazione "Alessandro Faedo"
Consiglio Nazionale delle Ricerche
Area della Ricerca CNR di Pisa
Via G. Moruzzi, 1 - 56124 PISA - Italy
{L.Candela|D.Castelli|P.Manghi|P.Pagano}@isti.cnr.it

May 4, 2006

Abstract

In this paper we advocate the benefits of introducing the notion of *typing* into Digital Library (DL) Repository Services. The OpenDLib Typed Repository Service allows DL designers to define typed repository sets, according to a type language T-DoMDL. Here a set becomes a container of digital objects sharing the same type-defined structure, i.e. properties, behaviors, and constraints. T-DoMDL types model a wide range of abstractions typically perceived in the Digital Library world, such as aggregation of digital objects, relations between digital objects, versioning of digital objects, and others.

The immediate advantage is that DL designers can construct repository sets tailored to their application domains. Secondly, DL developers, while implementing software components handling DL objects, can count on data consistency and static type checking of their code w.r.t. the set types. Finally, objects conforming to a given type can be physically handled in order to optimize both access and disk space, e.g. videos can be stored in a datastream access fashion.

Contents

1 Introduction and Motivations

The process of realization and usage of a Digital Library (DL) identifies three important roles: *DL users*, requesting a DL specific to their application domain, *DL designers*, designing the DL according to the DL user requirements, and *DL developers*, implementing the project of DL designers relying on the most appropriate DL System technology.

DL users perceive a DL as a set of *Functionality* ($\mathcal{F}_{DL}$) operating over the digital objects of an *Information Space* ($\mathcal{I}_{DL}$). At this level of abstraction, digital objects represent entities of the DL users application domain; for example, *conference proceedings*, i.e. collections of article objects. The structure of an $\mathcal{I}_{DL}$ is typically determined by DL designers, after analisying the DL users domain, in terms of high-level modeling primitives, e.g. a combination of *classes* of objects, each class describing domain-specific structure, namely *properties*, *constraints*, and *behavior*, of the *objects* it contains.

DL designers main task is the definition of an appropriate structure for the $\mathcal{I}_{DL}$, along with the specification of $\mathcal{F}_{DL}$. Next, we exemplify this process relying on the following example:

> **DL application domain (DL users)** Assume the scientists of the *European Space Agency* (ESA) work on an Earth observation project, aimed at observing and measuring environmental modifications of some planet sites. To this aim, scientists need to store a chronological history of site *observations*, each consisting of a satellite `avi` movie and sensor-gathered data in `txt` format. Given a site, the time of its next observation is established by elaborating the sensor-data of the last site observation. Scientists need to organize observations into site *investigations* and must be able to access the history of the observations relative to a site.
>
> $\mathcal{I}_{DL}$ **and** $\mathcal{F}_{DL}$ **(DL designers)** The structure of an $\mathcal{I}_{DL}$ for a DL supporting such investigation may include a class of *investigation* objects, each representing the history of observations relative to a planet site, and a class of *observation* objects, each representing satellite observations, i.e. pairs of an `avi` file and `txt` file. The association between an investigation object and one of its observation objects is characterized by a *version* number, which establishes the position of the former in the history of site observations. A minimal $\mathcal{F}_{DL}$ would feature an ingest interface, to handle investigation and observation objects, and a search interface, to retrieve observations by site or by version number through an investigation. Inserting an observation object requires uploading the two files and specifying both the associated investigation object and its version number.

DL developers typically implement DL's $\mathcal{I}_{DL}$ and $\mathcal{F}_{DL}$ as software *components* by extending and/or customizing the functionality supported by a DL *Repository Service* ($\mathcal{R}$). An $\mathcal{R}$ maintains a *Repository Information Space* ($\mathcal{I}_{\mathcal{R}}$), i.e. a storage unit for digital objects, and provides the primitives for *storing*, *accessing*, and *searching* digital objects into the $\mathcal{I}_{\mathcal{R}}$ [7] [4] [2]. Intuitively, DL developers would implement their $\mathcal{F}_{DL}$ in terms of $\mathcal{R}$ primitives and represent objects in $\mathcal{I}_{DL}$ with persistent objects in $\mathcal{I}_{\mathcal{R}}$. Unfortunately, differently from $\mathcal{I}_{DL}$'s, $\mathcal{I}_{\mathcal{R}}$'s cannot be partitioned into groups of objects that share the same structure, e.g. classes. Typically, an $\mathcal{I}_{\mathcal{R}}$ is a flat space of digital objects, all structured according to the *Digital Object Model* (DOM) of the $\mathcal{R}$. DOMs describe objects as general purpose storage units, whose structure and relationships with other objects can be flexibly tailored to describe "any" possible DL high-level entity. Due to this modeling "misalignment", $\mathcal{I}_{DL}$'s and $\mathcal{F}_{DL}$ can hardly be "directly interpreted", thus implemented, in terms of $\mathcal{I}_{\mathcal{R}}$'s objects and primitives.

As a consequence, DL developers "emulate" $\mathcal{I}_{DL}$ abstractions outside of $\mathcal{I}_{\mathcal{R}}$ boundaries, by "embedding" high-level modeling primitives into the engineering of components, and relying on

$\mathcal{I}_{\mathcal{R}}$ only for storage issues. Specifically, components will allocate and manage a number of local data structures in order to emulate the notion of classes, objects conforming to a class, and relationships between objects:

> **DL components (DL developers using traditional $\mathcal{R}$)** The ESA DL user interface (UI) component would offer the scientists a tool to manage an $\mathcal{I}_{DL}$-based information space. To this aim, the UI component would rely on an $\mathcal{R}$ platform to store and retrieve digital objects, and on top of this: (*i*) implement *investigations* and *observations* classes as independent containers of objects; (*ii*) support the notion of objects conforming to such classes (e.g. by implementing versioning of observation objects w.r.t. investigation objects), by performing structural controls at object insertion/retrieval time (e.g. uploading should be allowed only for `avi` and `txt` files) and implementing class methods in terms of $\mathcal{I}_{\mathcal{R}}$ primitives; and (*iii*) implement the relationship between investigation and observation objects.

The problem with such scenario is that the $\mathcal{I}_{\mathcal{R}}$ alone is "unaware" of its "real" content. $\mathcal{I}_{DL}$ structure is not stored along with the corresponding digital objects, but is instead encoded into the components together with its relationship with the underlying $\mathcal{I}_{\mathcal{R}}$. Accordingly, components are hard to develop, maintain, and integrate with others as their implementation is inspired by a logical $\mathcal{I}_{DL}$ while only $\mathcal{I}_{\mathcal{R}}$ primitives are visible to them. For example, no automatic tool prevents $\mathcal{I}_{DL}$ structural programming errors, thereby leaving data consistency up to developers skills and precision: components handling objects of the same $\mathcal{I}_{DL}$ class may mistakingly provide diverse interpretation of the $\mathcal{I}_{DL}$ structure into $\mathcal{I}_{\mathcal{R}}$ and compromise $\mathcal{I}_{DL}$ data consistency.

In this work we propose a new kind of Repository Service, which addresses all problems listed above. Based on the experience of developing DLs serving different application areas, the *OpenDLib* project [3] is experimenting the realization of a repository service $\mathcal{R}_t$ that can be configured by DL developers to support the most appropriate objects and primitives for components implementation. Such $\mathcal{R}$ contains a set of $\mathcal{I}_{\mathcal{R}}$ each containing a different *type* of objects. The type of an $\mathcal{I}_{\mathcal{R}}$, defined with the type language T-DoMDL [1][5], statically determines the structure, i.e. type-specific and user-specified properties and primitives, of the objects that will be contained into $\mathcal{I}_{\mathcal{R}}$.

$\mathcal{R}_t$ can define an information space tailored to the given DL, supporting the abstractions required by $\mathcal{I}_{DL}$. It therefore guarantees data consistency w.r.t. the application domain and facilitates component development and maintenance. Specifically, $\mathcal{R}_t$ can support all the advantages of types: (*i*) *static error detection*: types define object structure and operators, hence can be used to prevent components programming errors; (*ii*) *abstraction*: types organize the information space into interacting *modules*, i.e. $\mathcal{I}_{\mathcal{R}}$'s, thereby disciplining programming; (*iii*) *documentation*: types make component code easier to read; (*iv*) *data safety*: types guarantee data consistency, since all components can interact with an $\mathcal{I}_{\mathcal{R}}$'s only through the same type-specific behaviors; (*v*) *efficiency*: type information can be exploited to optimize object storage units, in terms of disk space and access time.

1.1 Related Work

To our knowledge, only Saidis et Al.[6] addressed typing for DLs. In their approach they introduced the notion of *prototypes*. Informally, prototypes are user-configurable components which virtualize the notion of class of objects. They reside on top of an $\mathcal{I}_{\mathcal{R}}$ and extend it in order to provide high-level class primitives to DL components developers.

2 T-DoMDL

T-DoMDL type language offers an exhaustive list of type abstractions, identified as common $\mathcal{I}_{DL}$ classes in the field of DL design: *manifestation*, *aggregation*, *relation*, *version*, and others. For example, all objects of an $\mathcal{I}_{\mathcal{R}}$ *Ris* of type *version* have the property *list of versions*, where each version in the list is identified by a version number and features a user-defined label and an object reference. Objects in *Ris* have primitives for inserting/deleting versions into/from the list, getting the last version, and others; and share the behaviors, if any, declared in *Ris* by the DL designer.

In *OpenDLib*, DL developers implement the structure of a given $\mathcal{I}_{DL}$ in terms of one or more types of T-DoMDL, then issue a request to the $\mathcal{R}_t$ for the instantiation of the corresponding $\mathcal{I}_{\mathcal{R}}$'s, i.e. a list of domain specific typed-sets. For the ESA DL, the designer may consider the following types:

DL components (DL developers using $\mathcal{R}_t$) *MovieMaps* and *Measurements*, both of type *manifestation* with format `avi` and `txt` respectively, *Observations* of type *aggregation* referring to two objects in *MovieMaps* and *Measurements*, and *Investigations* of type *version* referring to objects in *Observations.* Such $\mathcal{R}$ can thus store sets of typed objects and enable their management only through type-specific primitives. As a consequence, the $\mathcal{R}$ guarantees data consistency and facilitates component development and maintenance.

As mentioned above, typed $\mathcal{I}_{\mathcal{R}}$'s can enforce the optimization of object storage facilities. For example, from the $\mathcal{I}_{\mathcal{R}}$ *MovieMaps* of type *manifestation*, which defines precise `avi` format constraints, the $\mathcal{R}_t$ could configure a *storage unit* (object store) that optimizes disk space, by adopting *avi* compression techniques, and access performance, by providing streaming-based primitives. Furthermore, type information could be exploited to conveniently assign storage units to typed $\mathcal{I}_{\mathcal{R}}$'s. Specifically, different $\mathcal{I}_{\mathcal{R}}$'s may share the same storage unit or different storage units could be used to store the objects of the same $\mathcal{I}_{\mathcal{R}}$.

Indeed, as in DBMS systems, $\mathcal{R}_t$ supports *physical independence* between the *Content Layer*, which handles digital objects organized into typed $\mathcal{I}_{\mathcal{R}}$'s, and the *Storage Layer*, which handles *physical objects* into storage units. Physical objects are the minimal units of content, i.e. a file or, as we shall see, an action generating a file, characterized by a specific datastream format and a metadata structure. Digital objects represent instead a higher notion of modeling, allowing for viewing physical objects through *manifestation objects* and introducing structure and behavior, typical of the DL designer point of view, into the otherwise naive DOM supported by the physical layer. DL developers should therefore design the DL by both defining the types of $\mathcal{I}_{\mathcal{R}}$'s interpreting their application domain $\mathcal{I}_{DL}$ and defining the storage units required to preserve content, i.e. physical objects, into the $\mathcal{R}_t$. The two layers are independent and linked through a language which binds manifestation types, i.e. the content logical layer ground types, to the storage units they refer to.

In the following we shall introduce a basic physical layer model, where a manifestation type can refer to only one storage unit, but the same storage unit can give support to different manifestation types.

2.1 Storage Layer

The Storage Layer of an $\mathcal{R}_t$ is composed by a *metadata store*, i.e. a container of metadata schemas and mappings from schema to schema, and a set of content *stores*, i.e. containers of physical objects.

Metadata store. The metadata store (MD) is a set of *resource descriptors*. A resource descriptor is composed by a default schema, namely an XML Schema describing the structure of the metadata, and a set of mappings from the default schema to other metadata schemas. Mappings are identified by a transformation function and a target XML schema, with which typing and validating a mapping.

$$MD = \{rs_1, \ldots, rs_k\}$$

where:

$$rs_i = (ds, maps)_i$$
$$maps = \{(tf_1 : ds \ \rightarrow \ schema_1), \ldots, (tf_n : ds \ \rightarrow \ schema_n)\}$$

and tf_i maps each XML path in ds onto an XML path of $schema_i$.

Stores. A store is a container of physical objects, i.e. pairs of one *data resource* and one *metadata record* describing the resource according to one resource descriptor in the MD.[1] The simplest expression of a data resource is a document file, or a URI reference to it. The most complex expression of data resources can be actions that generate files from the elaboration of other resources, e.g. relational databases, DLs, GRID content, etc.

A store is a container of physical objects described as a tuple $name(k, BF, RD)$, where:

- *name* is the unique name of the store;
- k is the *kind*: a kind expresses the category of data resources handled by the store, i.e. the file formats allowed for storage. Available kinds are; *video*, *audio*, *textual*, *image*, *living*, or *object* (all possible formats). For example, the kind *textual* identifies data resources whose issue is human reading and/or editing, hence files with formats such as `PDF`, `PS`, `DOC`, etc. Files formatted `MP3` will not be contained into the store. Again, the kind *living* identifies data resources that can be derived from the elaboration of external resources. Typically such stores define one *action*[2] and store data resource defines the parameters for that action. Retrieving a data resource involves the firing of the action and the generation of a file which must respect *object* kind constraints.
- BF is a set of *base file formats*: a store may limit the file formats it handles to a given subset of all formats implied by its kind k;
- RD is the *resource descriptor*: the resource descriptor in MD adopted by *name*.

Typing of stores preserves data consistency by preventing storage of objects that do not conform to BF and RD. Furthermore, all stores and all physical objects do respond to the following sets of primitives:

$$name.query(Q) = \{o \mid o \in S \wedge Q(o)\}$$

where Q is a query over the objects in S. Q may for example be a predicate involving the metadata fields of *name*.[3]

As to the physical objects, given a store *name*, then $\forall o \in name$:

- $o.resource \in f$, where $f \in BF$ and $BF \subseteq Format(k)$;

[1]With data resource we mean any human or machine generated content that can be directly interpreted by a human. In other words, digital content that is not usually stored into relational database and the similar.

[2]An URI reference to on-line executable code.

[3]Note that, since both the structure of the metadata records and the structure of Q are known, techniques to check correctness of queries could be adopted.

- $o.mdrecord \in RD.ds$, i.e. $o.mdrecord$ is the metadata record of o that must conform to the default schema ds of RD.

Finally, given $mdr = o.mdrecord$, fields in mdr can be dereferenced or assigned values as follows:

$$\begin{array}{ll} mdr.l_1 \ldots l_k & (dereference) \\ mdr.l_1 \ldots l_k = v & (assignment) \end{array}$$

where l_i's are XML elements, $l_1 \ldots l_k$'s are valid paths into $RD.ds$, and l_k is a leaf element.

The breakdown of stores into different data resource kinds and base formats allows for the definition and usage of specific storage techniques and access algorithms. For example, video stores may support streaming or image stores may adopt image specific compression algorithms. Furthermore, DL designers may better organize their storage layer, thereby speeding up search algorithms and giving low level support for object categorization (independent stores may contain domain distinct resources but feature the same kind).

Within the storage layer, DL developers interact with an $\mathcal{R}_t$ composed by a set of stores. Developers use the stores to (*i*) insert/access physical objects conforming to a type and to (*ii*) query the store according to the metadata records or the data resource content. However, they could hardly directly implement their $\mathcal{I}_{DL}$ and $\mathcal{F}_{DL}$ in terms of stores and store primitives. Indeed, most of the DLs will likely require:

- querying more than one store at the same time;
- creating complex objects, obtained by the combination of physical objects;
- defining restricted views over the stores, in order to provide different users or components with different interpretations of the stores;
- handling of DL functionality such as annotations, versions, user defined relations among different objects;

These points will be addressed by the abstractions introduced into the Content Layer.

2.2 Content Layer

Within the Content Layer, the $\mathcal{R}$ is constituted by a list of *named* and *typed* $\mathcal{I}_\mathcal{R}$'s, here intended as containers of objects with the same structure and behavior. $\mathcal{I}_\mathcal{R}$'s are abstractions over the storage layer and allow for the definition of complex digital objects. Specifically, an $\mathcal{I}_\mathcal{R}$ is a container of objects described as a tuple $name(t, IF, OF, TFF, RD, B)$, where:

- $name$ is the unique name of the $\mathcal{I}_\mathcal{R}$;
- t is the type of $name$;
- IF and OF are the input and output formats of $name$;
- TFF$_{IF}$ (*Transcoding Function Formats*) is the set of formats entailed by the application of transcoding functions (e.g. ps2pdf) to the IF formats;
- RD is the resource descriptor in MD adopted by $name$;
- B is a set $\{b_1, \ldots, b_k\}$ of behaviors, i.e. functions shared by all objects in $name$ (in the style of class public methods, $b_i : (Object \times Val) \rightarrow f,\ f \in Format$.

Independently from the type t of the $\mathcal{I}_{\mathcal{R}}$, all digital objects respond to a common set of primitives. Given the $\mathcal{I}_{\mathcal{R}}$ Ris, then $\forall o \in Ris$:

$$\begin{array}{ll}
(i) & o.manifestation(f) \in f,\ f \in OF\ or\ f \in TFF_{IF} \\
(ii) & o.mdrecord \in RD.ds \\
(iii) & o.metadata(RD.maps.tf) \in RD.maps.schema \\
(iv) & o.pars \in \wp(Val) \text{ (set of actual parameters for behavior calls)} \\
(v) & o.b(p.pars) \text{ (firing of an action, which returns a data resource)}
\end{array}$$

As for physical objects, the metadata record fields of a digital object can be assigned or dereferenced. Digital objects can instead encode metadata records into one of the target schemas identified by a mapping (iii). Similarly, while physical objects can return only expose the data resource format they have been instantiated with, digital objects can also expose the data resource through one of the formats enabled by TFF_{IF} (i).

Finally, an $\mathcal{I}_{\mathcal{R}}$ Ris can respond to methods such as:

$$Ris.query(Q) = \{o \mid o \in name \wedge Q(o)\}$$

T-DoMDL introduces a minimal set of types t, which we believe are essential for the design and development of DLs or, more generally, *Content Management Systems*:

$$\begin{array}{lllr}
T & ::= & \mathcal{M}_{\mathcal{R}}(S) & (manifestationtype) \\
 & & \mathcal{V}_{\mathcal{R}}(T) & (viewtype) \\
 & & \mathcal{E}_{\mathcal{R}}(T) & (editiontype) \\
 & & \mathcal{A}_{\mathcal{R}}(T_1, \ldots, T_n) & (aggregationtype) \\
 & & \mathcal{S}_{\mathcal{R}}(l_1 : T_1, \ldots, l_n : T_n) & (structuretype) \\
 & & \mathcal{R}_{\mathcal{R}}(T_1, T_2) & (relationtype) \\
 & & \mathcal{C}_{\mathcal{R}}(Q) & (collectiontype)
\end{array}$$

where S is a store and Q is a query. In the following we introduce types, by motivating their adoption, and by listing their peculiar properties and primitives.

2.2.1 Manifestation $\mathcal{I}_{\mathcal{R}}$ ($\mathcal{M}_{\mathcal{R}}$)

Stores handle physical objects, hence pairs data resource/metadata record. In particular, they ensure storage of data resources conforming to a given BF and metadata records conforming to a given RD. Manifestation objects encapsulate physical objects of a given store by adding behaviors and ruling out, depending on the intended usage, part of the resource descriptor and of the base formats inherited by the store. In particular, there is a one-to-one correspondence between physical objects and relative manifestation objects. Accordingly, if one of two $\mathcal{M}_{\mathcal{R}}$'s referring to the same store removes or adds digital objects, the action will indirectly affect the other $\mathcal{M}_{\mathcal{R}}$. The definition of an $\mathcal{M}_{\mathcal{R}}$ Ris is of the form:

$$Ris(IF, OF, TFF_{IF}, RD', B) = \mathcal{M}_{\mathcal{R}}(S(t, k, BF, RD))$$

where

$$\begin{array}{ll}
(i) & IF \subseteq BF \\
(ii) & OF \subseteq IF \\
(iii) & RD' \subseteq_t RD.
\end{array}$$

where $RD' \subseteq_t RD$ means that RD' is a tree whose paths are also in RD, i.e. the metadata structure described by RD' is a subset of the one described by RD.

2.2.2 View $\mathcal{I}_\mathcal{R}$ ($\mathcal{V}_\mathcal{R}$)

$\mathcal{V}_\mathcal{R}$'s objects "encapsulate" objects of other $\mathcal{I}_\mathcal{R}$'s. $\mathcal{V}_\mathcal{R}$ encapsulation can hide parts of the metadata structure and rule out or add behaviors to or from the encapsulated objects. The definition of an $\mathcal{V}_\mathcal{R}$ Ris is of the form:

$$Ris(IF, OF, TFF_{IF}, RD, B) = \mathcal{V}_\mathcal{R}(Ris'(IF', OF', TFF_{IF'}, RD', B'))$$

Where

- (i) $IF \subseteq OF'$
- (ii) $OF \subseteq IF$
- (iii) $RD \subseteq_t RD'$
- (iv) there is no dependency between B and B' or between TFF_{IF} and $TFF_{IF'}$

Furthermore, the following property holds: $o \in Ris \Leftrightarrow o \in Ris'$. Since different $\mathcal{V}_\mathcal{R}$'s can be defined over the same $\mathcal{I}_\mathcal{R}$, modification over one $\mathcal{V}_\mathcal{R}$ will affect the content of the others.

2.2.3 Aggregation $\mathcal{I}_\mathcal{R}$ ($\mathcal{A}_\mathcal{R}$)

$\mathcal{A}_\mathcal{R}$'s objects aggregate objects from different $\mathcal{I}_\mathcal{R}$'s. The definition of an $\mathcal{A}_\mathcal{R}$ Ris is of the form:

$$Ris(IF, OF, TFF_{IF}, RD, B) = \mathcal{A}_\mathcal{R}(Ris_1(IF_1, OF_1, TFF_{IF}, RD_1, B_1), \dots, Ris_k(IF_k, OF_k, TFF_{IF}, RD_k, B_k))$$

Where

- (i) $IF = Object$
- (ii) $OF = Object$
- (iii) $RD \in MD$

Furthermore, $\mathcal{A}_\mathcal{R}$'s define the following operators over objects.

- $o.set \in \wp(Object)$ returns the set of objects aggregated by o;
- $o.add(o')$ additions the object o' to the set of objects aggregated by o;
- $o.del(o')$ removes o' from the set of objects aggregated by o.

2.2.4 Structure $\mathcal{I}_\mathcal{R}$ ($\mathcal{S}_\mathcal{R}$)

$\mathcal{S}_\mathcal{R}$'s objects link objects of other stores, naming them in the style of relational database records. The definition of an $\mathcal{S}_\mathcal{R}$ Ris is of the form:

$$Ris(IF, OF, TFF_{IF}, RD, B) = \mathcal{S}_\mathcal{R}(l_1 : Ris_1, \dots, l_k : Ris_k)$$

Where

- (i) $IF = Object$
- (ii) $OF = Object$
- (iii) $RD \in MD$

Furthermore, $\mathcal{S}_\mathcal{R}$'s define the following operators over objects; given $o \in Ris$, then $o.l_i \in Ris_i$.

2.2.5 Relation $\mathcal{I}_{\mathcal{R}}$ ($\mathcal{R}_{\mathcal{R}}$)

$\mathcal{R}_{\mathcal{R}}$'s contain objects representing a relationship between two objects from different $\mathcal{I}_{\mathcal{R}}$'s. Such $\mathcal{I}_{\mathcal{R}}$'s enable the definition of user defined associations between objects not related through particular object types, such as $\mathcal{A}_{\mathcal{R}}$ or $\mathcal{S}_{\mathcal{R}}$ objects. The definition of an $\mathcal{R}_{\mathcal{R}}$ Ris is of the form:

$$Ris(IF, OF, TFF_{IF}, RD, B) = \mathcal{R}_{\mathcal{R}}(Ris_1, Ris_2)$$

Where

$$\begin{array}{ll} (i) & IF = Object \\ (ii) & OF = Object \\ (iii) & RD \in MD \end{array}$$

Furthermore, $\mathcal{R}_{\mathcal{R}}$'s define the following operators over objects; given $o \in Ris$, $o.fst \in Ris_1, o.snd \in Ris_2$ return the first and the second object associated by o. Finally, an $\mathcal{R}_{\mathcal{R}}$ responds to the primitives:

- $Ris(rel, o) \in \wp(Object)$ returns the set of objects associated to o by objects in Ris, that is the set

  $$R_{(o,Ris)} = \{\bar{o} \mid \exists o' \in Ris : (o'.fst = o \wedge o'.snd = \bar{o}) \vee (o'.fst = \bar{o} \wedge o'.snd = o).$$

  Note that R is either contained in Ris_1 or in Ris_2.

- $Ris(car, o) \in \aleph$ returns the number of objects related to o by the relation Ris.

2.2.6 Collection $\mathcal{I}_{\mathcal{R}}$ ($\mathcal{C}_{\mathcal{R}}$)

$\mathcal{C}_{\mathcal{R}}$'s contain objects resulting from a query Q. A query can be factored out as a set of k predicate sub-queries Q_i, each relative to an index over a $\mathcal{I}_{\mathcal{R}}$ Ris_i. The object sets resulting from the Q_i's are then combined according to the Boolean operators used to join the predicates in Q.[4] involved in Q.

$$Ris(IF, OF, TFF_{IF}, RD, B) = \mathcal{C}_{\mathcal{R}}(Q)$$

Accessing objects in Ris requires a *projection operation* over one of the Ris_i. Similarly, querying Ris should focus on one $\mathcal{I}_{\mathcal{R}}$,

$$Ris.query(Q', Ris_i) = \{o \mid o \in Ris_i \wedge Q(o) \wedge Q'(o)\}$$

Similarly, ingest operators will be overloaded with projection operators, but also query constraints, to guarantee type checking according to both structure and value constraints of Ris. For example, if a $\mathcal{C}_{\mathcal{R}}$ Ris is obtained from a query of the form

```
Q = SELECT FROM Ris'
    WHERE dc:title LIKE "computer science"
```

where

$$[\![Q]\!]_{=}\{o \mid o \in Ris' \wedge o \in Index(RD, dc : title, LIKE, \text{“}computerscience''\text{)}\}$$

ingestion into Ris can only accept objects with the structure of Ris' and the metadata field `dc:title` set to a value that contains the string `"computer science"`.

[4]The number of $\mathcal{I}_{\mathcal{R}}$ (s) is not necessarily equal to the number of sub-queries (k); this is because an index serving a query may refer to more than one $\mathcal{I}_{\mathcal{R}}$

2.2.7 Edition $\mathcal{I}_{\mathcal{R}}$ ($\mathcal{E}_{\mathcal{R}}$)

$\mathcal{E}_{\mathcal{R}}$'s contain objects handling different versions of the same digital object. We use such objects whenever versioning is not part of the semantics of individual objects, but of their usage; that is, versioning is not an object property, but is instead a property of another object representing the association "versioning" among a set of objects. For example, each version of a technical report is a document and can be interesting by itself. However, if we were also interested in knowing how many versions of the same technical report are available, then we would need a version object, which links the versions and keeps the property of their versioning relationship.

$$Ris(IF, OF, TFF_{IF}, RD, B) = \mathcal{E}_{\mathcal{R}}(Ris_1, \ldots, Ris_k)$$

Where

$$\begin{array}{ll} (i) & IF = Object \\ (ii) & OF = Object \\ (iii) & RD \in MD \end{array}$$

Furthermore, $\mathcal{E}_{\mathcal{R}}$'s define the following operators over objects. Given $o \in Ris$:

- $o.editions \in \wp(\aleph \times Object \times \aleph)$ returns the ordered list of versions of o, i.e. a list of triples of the form $< vn, o, l >$, where vn is the version number, o is the object relative to the version, and l is a user-defined label);
- $o.length \in Nat$ returns the number of versions of o;
- $o.add(i, o', l)$: adds at position i of the version list of o a new object o' with user defined label l; all triples from position i to $o.length$ are increased their system version number by one;
- $o.del(i)$: removes the version with sequence number i; all triples from position i to $o.length$ are decreased their system version number by one.

As for array indexes, system version numbers are progressive integers starting from 0. The removal or the addition of a new edition object to an object o, entails the update of all sequence numbers. User version numbers are inserted by the user according to his/her needs and are not used by the $\mathcal{I}_{\mathcal{R}}$ to distinguish between different versions. A `UNIQUE` constraint could be enforced on the field, but it is not set by default (i.e. different versions in o may have the same user version number).

3 Conclusions and future issues

In this work we presented the definition of $\mathcal{R}_t$, a typed repository service for *OpenDLib*. $\mathcal{R}_t$ stores $\mathcal{I}_{\mathcal{R}}$'s of typed objects and enable their management only through type-specific primitives. As a consequence, $\mathcal{R}_t$ guarantees data consistency, facilitates component development and maintenance, and enables disk space and access efficiency optimization.

As a further step of investigation, we shall target the issues regarding the automatic generation of components from typed $\mathcal{I}_{\mathcal{R}}$'s. Indeed, since T-DoMDL types specify a precise semantics of the objects they define, typical of DLs world, typed $\mathcal{I}_{\mathcal{R}}$'s can enforce the automatic generation of components to handle objects of a given type. For example, from a $\mathcal{V}_{\mathcal{R}}$, the $\mathcal{R}_t$ may be able to automatically generate an user interface for the management of the relative objects, which includes versioning management. Indexes, as well as browsing components may also be generated for different $\mathcal{I}_{\mathcal{R}}$'s, depending on the metadata they support. Of course, components generation from typed $\mathcal{I}_{\mathcal{R}}$'s may be mediated by an interface generation language, allowing DL developers to refine the generation process and deliver components fitting DL designers expectations.

References

[1] L. Candela, D. Castelli, P. Pagano, and M. Simi. From Heterogeneous Information Spaces to Virtual Documents. In E. A. Fox, E. J. Neuhold, P. Premsmit, and V. Wuwongse, editors, *Digital*

Libraries: Implementing Strategies and Sharing Experiences, 8th International Conference on Asian Digital Libraries, ICADL 2005, Lecture Notes in Computer Science, pages 11–22, Bangkok, Thailand, December 2005. Springer.

[2] D. Castelli and P. Pagano. OpenDLib: A Digital Library Service System. In M. Agosti and C. Thanos, editors, *6th European Conference on Research and Advanced Technology for Digital Libraries, ECDL 2002*, Lecture Notes in Computer Science, pages 292–308, Rome, Italy, September 2002. Springer-Verlag.

[3] D. Castelli and P. Pagano. A System for Building Expandable Digital Libraries. In *ACM/IEEE 2003 Joint Conference on Digital Libraries (JCDL 2003)*, pages 335–345. Springer-Verlag, 2003.

[4] C. Lagoze, S. Payette, E. Shin, and C. Wilper. Fedora: An Architecture for Complex Objects and their Relationships. *Journal of Digital Libraries, Special Issue on Complex Objects*, 2005.

[5] OpenDLib. A Digital Library Service System. `http://www.opendlib.com/`.

[6] K. Saidis, G. Pyrounakis, and M. Nikolaidou. On the Effective Manipulation of Digital Objects: A Prototype-Based Instantiation Approach. In *Research and Advanced Technology for Digital Libraries: 9th European Conference, ECDL 2005, Vienna, Austria, September 18-23, 2005. Proceedings*, 2005.

[7] R. Tansley, M. Bass, and M. Smith. DSpace as an Open Archival Information System: Current Status and Future Directions. In T. Koch and I. Sølvberg, editors, *Research and Advanced Technology for Digital Libraries, 7th European Conference, ECDL 2003, Trondheim, Norway, August 17-22, 2003, Proceedings*, Lecture Notes in Computer Science, pages 446–460. Springer-Verlag, 2003.