Bibliographic Query Service Specification

Convenience Document - dtc/2001-12-03
December 2001

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIEDWARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org*, under Documents & Specifications, Report a Bug/Issue.

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

OMG Documents

The OMG documentation is organized as follows.

OMG Modeling

- ***Unified Modeling Language (UML) Specification*** defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems.
- ***Meta-Object Facility (MOF) Specification*** defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models.
- ***OMG XML Metadata Interchange (XMI) Specification*** supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information.

Object Management Architecture Guide

This document defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

CORBA: Common Object Request Broker Architecture and Specification

Contains the architecture and specifications for the Object Request Broker.

OMG Interface Definition Language (IDL) Mapping Specifications

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

CORBAservices

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include specifications such as *Collection*, *Concurrency*, *Event*, *Externalization*, *Naming*, *Licensing*, *Life Cycle*, *Notification*, *Persistent Object*, *Property*, *Query*, *Relationship*, *Security*, *Time*, *Trader*, and *Transaction*.

CORBAfacilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include specifications such as *Internationalization and Time*, and *Mobile Agent Facility.*

Object Frameworks and Domain Interfaces

Unlike the interfaces to individual parts of the OMA "plumbing" infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Currently, specifications are available in the following domains:

- *CORBA Business*: Comprised of specifications that relate to the OMG-compliant interfaces for business systems.
- *CORBA Finance*: Targets a vitally important vertical market: financial services and accounting. These important application areas are present in virtually all organizations: including all forms of monetary transactions, payroll, billing, and so forth.
- *CORBA Healthcare*: Comprised of specifications that relate to the healthcare industry and represents vendors, healthcare providers, payers, and end users.
- *CORBA Life Science*: Comprised of specifications that relate to the OMG-compliant interfaces for the life science industry.
- *CORBA Manufacturing*: Contains specifications that relate to the manufacturing industry. This group of specifications defines standardized object-oriented interfaces between related services and functions.
- *CORBA Telecoms*: Comprised of specifications that relate to the OMG-compliant interfaces for telecommunication systems.
- *CORBA Transportation*: Comprised of specifications that relate to the OMG-compliant interfaces for transportation systems.

Obtaining OMG Documents

The OMG collects information for each book in the documentation set by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters

250 First Avenue

Needham, MA 02494

USA

Tel: +1-781-444-0404

Fax: +1-781-444-0320

pubs@omg.org

http://www.omg.org

Acknowledgments

The following companies submitted and/or supported parts of this specification:

- EMBL-EBI (European Bioinformatics Institute)

Introduction 1

Contents

This chapter contains the following sections.

1.1 Bibliographic Query Introduction

Bibliographic search and citation are central to all scholarly and research activities. Within the domain of life sciences research, bibliographic citation is of particular importance for annotation of large bodies of experimentally developed and computationally derived data and the rapidly increasing corpus of research literature makes efficient and effective bibliographic searches increasingly critical.

The relevant “literature” may include traditional (hardcopy) research journal publications, books, theses, reviews and the like. Recent developments require researchers and scholars to use and cite a wider variety of sources, including database records, electronically published journals, World Wide Web sites and multimedia works. While several standards exist for the representation of bibliographic citations, some of them are not readily adapted to newer forms of publication and there is no common set of interfaces for distributed object implementations of bibliographic servers.

Examples of uses of a common set of CORBA-based bibliographic service interfaces include, but are not limited to:

- Enabling access to heterogeneous bibliographic databases and the development of interoperable clients that make use of this access.
- Enabling the development of clients that can be easily extended to novel bibliographic data sources.

This specification seeks solutions for the given situation.

1.2 Naming Conventions

The authors of this specification considered several possible names for the main object described and used by this specification. Finally, the short name “Citation” was not used in IDL definitions because of possible confusion with the meaning “quotation” (especially in the scientific community). The main data type is called “BibliographicReference,” which is in some constructs shortened to “BibRef.” However, the text of the specification uses both names, “Citations” and “Bibliographic references,” interchangeably.

Note that through this specification the name *class* is often used when talking about valuetypes. We consider this name more appropriate when talking about overall design strategies. It also allows grouping together valuetypes, structs and interfaces.

1.3 Scope and Extensibility

This specification can be used for both

- providing the ability to query a bibliographic repository for citations of a variety of document types, and
- retrieving the citation matching the query criteria.

Having these abilities, the specification can be used as a part of other LSR specifications for dealing with citations, which includes but is not limited to:

- biomolecular sequence objects,
- gene expression data objects,
- macromolecular structure objects, either directly, or by inheriting.

The specification defines a set of attributes that can be extended using “dynamic properties” described in Section 2.1.5, “Dynamic Properties,” on page 2-3. This allows extendibility without losing interoperability.

1.4 Module Dependencies

The specification is composed of two new modules, and uses several modules of existing or emerging CORBA standards as shown in Figure 1-1 on page 1-3.

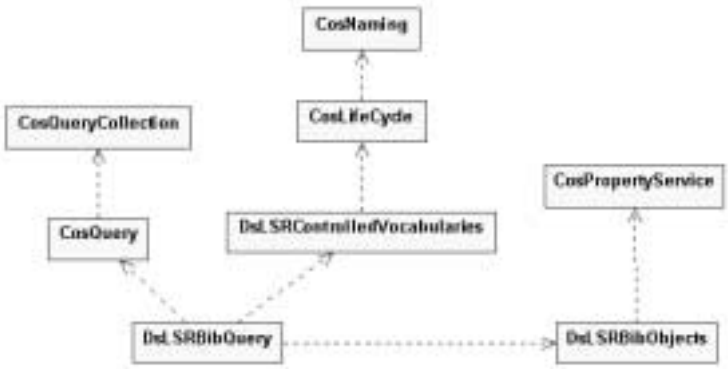

Figure 1-1 Module Dependencies

Modules and Interfaces 2

Contents

This chapter contains the following sections.

2.1 The DsLSRBibObjects Module

2.1.1 Overview

This module specifies the bibliographic domain data. It is independent of other modules found in this specification, therefore it can be used everywhere where only the data model is needed, without query capabilities.

The data model represents more than the minimum required by the RFP but is still quite extendible by using dynamic properties. This allows the adding of additional attributes without losing interoperability.

2.1.2 Dublin Core Metadata

The information model of bibliographic citations is based on the **Dublin Core Metadata for Resource Discovery** [DC] that represents a set of descriptors that has emerged from a workshop series started in 1995 in interdisciplinary and international consensus building. The focus of the initiative was electronic resources. It was clear at the outset,

however, that the semantics of resource discovery should be independent of the medium of the resource, and that there are obvious advantages for using the same semantics model across media. Thus, considerable attention has been invested in making the Dublin Core sufficiently flexible to represent resources (and relationships among resources) that are both digital and exist in traditional formats as well.

One of the issues presented in the Bibliographic Query Service RFP was to consider both traditional and new media types. The Dublin Core seems to be a perfect match for this requirement. This specification, however, adds a few new attributes to the core set, especially for purposes of easier queries and for better extendibility.

2.1.3 Objects-by-value

The CORBA 2.3a specification [CORBA2.3] provides the concept of a valuetype, an IDL data type intermediate between struct and interface. It is a part of the so-called Objects by Value (OBV) specification. In the context of this standard, the benefit of valuetypes over interfaces is scalability (a single round trip transfers the whole state of the object). The benefit of valuetypes over structs is their extendibility through inheritance. This standard uses valuetypes essentially as extendible structs, by applying the following constraints:

- all members (‘attributes’) are *public*,
- there are no methods,
- inheritance is only of other valuetypes (i.e., no “supports *SomeInterface*”),
- all inheritance uses *truncatable* (i.e., “casting” a sub-type to its super-type by simply omitting the extra members is a semantically valid operation).

2.1.4 Illustrative UML Diagram

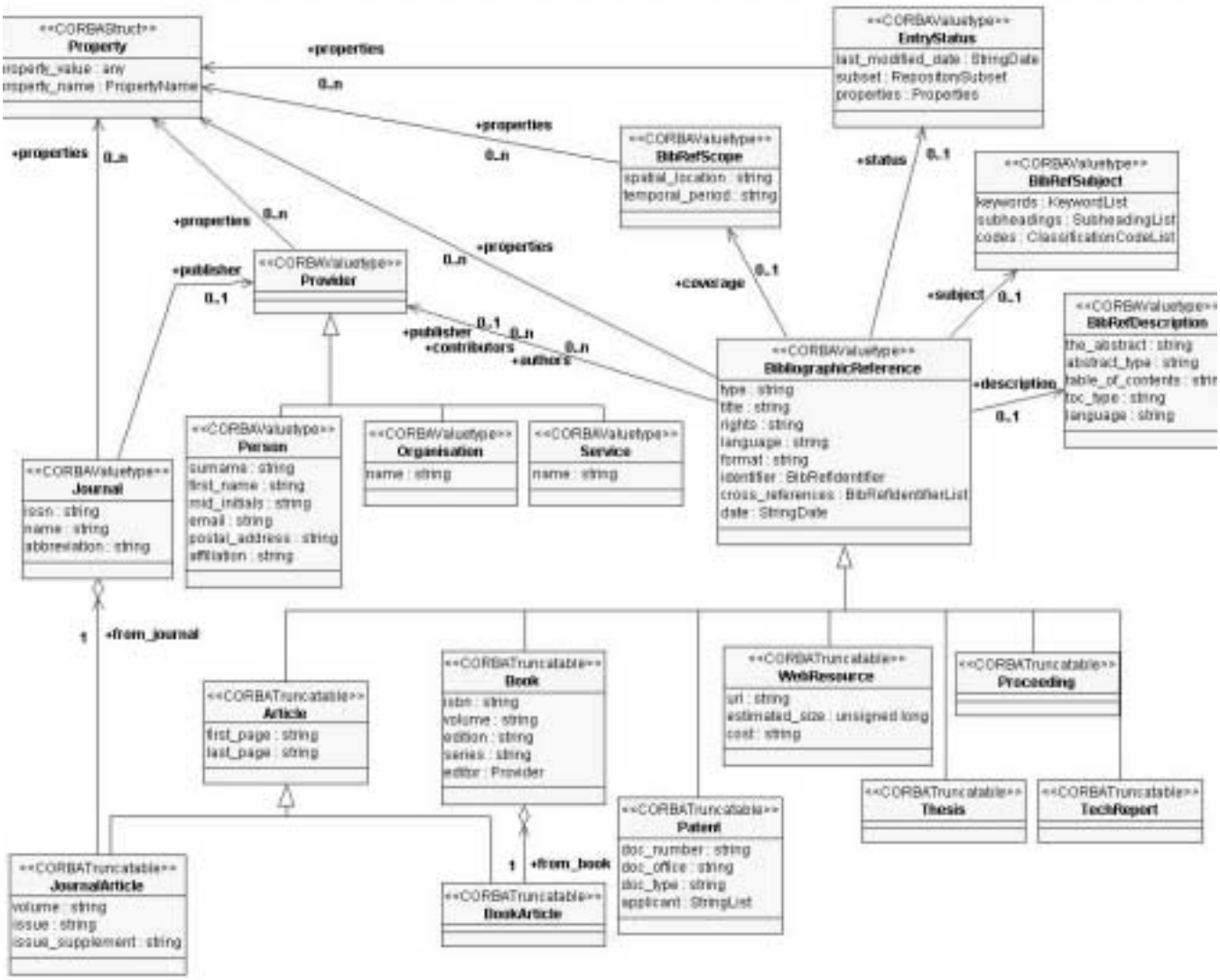

Figure 2-1 Data Model

The UML in this specification is illustrative, not normative.

2.1.5 Dynamic Properties

For achieving extendibility without losing interoperability, several data classes use attribute of type **CosPropertyService::Properties**.

Together with the repository introspection mechanism (see Section 2.2.6, “Repository Introspection,” on page 2-18) it allows the inclusion of any property while still being able to control the property names (which is necessary for queries) and, if needed, property values.

2.1.6 Data Structures

2.1.6.1 BibliographicReference

The BibliographicReference class is the core of the data model. It is a super-class for all specialized citation types, but it can also be instantiated and represent an additional type not specifically defined in this specification.

public BibRefIdentifier identifier;

It is an unambiguous reference to the citation “within the world.” It is a string conforming to an identification system described and rationalized below:

There is a requirement for a simple data type to store a bibliographic reference identity across various bibliographic repositories. In most cases, this need is, or can be, addressed by using a string type. The advantages are that it is simple, lightweight, and used universally throughout the realm of computing (and indeed outside). However the risk of using strings is that they can be too flexible, both in terms of syntax and semantics. This easily results in the lack of interoperability. To allow strings, yet mitigate their potential for abuse, this specification (and indeed some other LSR specifications [BSA], [MAPS]) uses the syntax convention of **CosNaming::StringName** as described in the Interoperable Naming service [INS]. This convention is mainly a syntactical one; in no way is the use of a naming service implementation required or implied (but it is not precluded either).

A brief description of **CosNaming::StringName** is as follows. **CosNaming::Name** is a list of **struct NameComponent**s. For the purpose of illustration, a **NameComponent** can be likened to a directory or filename, whereas **CosNaming::Name** constitutes a full path-name. The **struct NameComponent** has string members **id** and **kind**. To transform a **CosNaming::Name** into a string, all its **NameComponent**s are represented as strings “ **id.kind**”. If the **kind**-field is empty, this becomes simply “ **id**”; if the **id**-field is empty, this becomes “**.kind**”; finally, the Naming service also allows both the **id**- and **kind**-fields to be empty, which is represented as “`.`”. The full *stringified* **CosNaming::Name** is then obtained by concatenating all the **NameComponent**s using “`/`” as a separator character. The character “`\`” is designated as an escape character; if it precedes any of the special characters “`.`”, “`/`” and “`\`”, these special characters are taken as literal characters. The **typedef string CosNaming::StringName** is provided for strings used as object names using this convention.

This specification adopts the same syntax convention for a bibliographic reference identifier, but requests that the components of **BibRefIdentifier** data type adhere to some additional semantic constraints. These rules do not follow from, nor are implied by,

any semantics of the Naming Service. The additional constraints make this data type sufficiently different from **CosNaming::StringName** to warrant the dedicated **typedef string BibRefIdentifier**.

In the remainder of this description, ‘component’ means: the sub-string of a **BibRefIdentifier** that corresponds to one **CosNaming::NameComponent**; likewise, **id**-field and **kind**-field correspond to the equivalent fields of **NameComponent**.

The rules are as follows:

- The first component represents the bibliographic data source. Data sources can be anything: transient collections, local databases, public repositories, etc. It is up to the implementation to document the accepted names for the data source.
- The empty name (“.”) is valid for the first component, and represents the ‘local’ or ‘default’ collection. It is up to the implementation to document what the semantics of ‘local’ or ‘default’ are.
- Names that refer to citations within collections (which is the usual case) consist of two or more components. The second component of such names represents an identifier that is unique in the context of the data source. No empty **id**-fields are allowed in this or any further components.
- If two components are not enough to uniquely identify an entity, a **BibRefIdentifier** can contain more than two components, but no more than necessary to make the identification unique. That is, a **BibRefIdentifier** may not be used to freely attach textual information.
- The only characters valid in a component are “a” through “z,” “0” through “9,” and “-“ (hyphen), “_” (under_score), “$” and “.” (period). Use of the latter is discouraged since it has a special meaning in the *stringifying* convention, and has therefore to be escaped.

To comply with existing practice in the field of public data repositories, it is strongly advised that implementations do string comparisons in a case-insensitive manner. The CosNaming Service specification fails to mention whether type-case is, for string comparison purposes, significant or not. Implementations that use a third-party implementation of the Naming service may therefore wish to restrict **BibRefIdentifier**s to only use one type-case. It is up to an implementation to state whether mixed type-case is allowed, and whether type-case is significant in comparisons.

The **id** and **kind** parts of the string components of **BibRefIdentifier** are used as follows:

- The **id**-field of a component contains the principal value that makes it unique in the scope provided by the proceeding component. It may only be empty in the case of the first component of a **BibRefIdentifier** (see above).
- The **kind**-field of a component is used to represent information indicating the release (for a data source) or version (for a citation), and can be empty.

The examples of **BibRefIdentifiers**:

Medline/10881088

Embl-pub.56/123456

Note that **BibRefIdentifier** identifies a citation, not necessarily the cited resource (such as a document) itself. Depending on the type of the cited resource, there can be other identification systems used (such as ISBN number for books or ISSN number for journals).

public string type;

It defines the nature or genre of the cited resource. Although a working draft of Dublin Core Types [DCT1] recommends a type classification, the proposed types are mostly out of scope of this specification. The majority of cited resources would fall in the same category "text." For the future, however, Dublin Core is considering the addition of subtypes to the high level types, or other ways of making sub-categories.

Therefore, the recommended best practice is to select a value from a controlled vocabulary **RESOURCE_TYPES** using methods of repository introspection (see Section 2.2.6, "Repository Introspection," on page 2-18).

Syntactically, and because of making query navigation easier, the value of this attribute should be the same as the name of a sub-class that inherits from BibliographicReference class and that describes the type. However, there may be bibliographic resources, which are not described by specialized sub-classes (for example, "letters," "practical guideline," or "archive"). For such cases the usage of a controlled vocabulary is needed.

In contrast to the Dublin Core recommendations, the resource type in this specification cannot be repeated to include different categories of cited resources in one citation.

Note that for the description of the physical or digital manifestation of the cited resource there is an attribute **format** described later.

public BibRefIdentifierList cross_references;

It is a list of identifiers, all of them pointing *to the same cited source* but usually stored in different bibliographic repositories. Note that this attribute *is not* for referencing citations to documents *related* to a document or citation.

public wstring title;

A title given to the cited resource (a name by which the resource is formally known).

public BibRefSubject subject;

It defines the topic of the content of the cited resource. It is expressed in one or more ways using a construct **BibRefSubject**:

typedef wstring Keyword;
typedef sequence <Keyword> KeywordList;

typedef ~~VocabularyS~~string SubjectH~~h~~eading;
typedef sequence <SubjectH~~h~~eading> SubjectH~~h~~eadingList;

```
typedef string ClassificationCode;
typedef sequence <ClassificationCode> ClassificationCodeList;

valuetype BibRefSubject {
    public KeywordList keywords;
    public SubjectHeadingList subject_headings;
    public string subject_heading_collection;
    public ClassificationCodeList codes;
}
```

The keywords are usually (but not limited to) one word long. They are not controlled by any vocabulary.

The subject headings usually come from standard lists, such as *Sears List of Subject Headings* [SEARS], or *Library of Congress Subject Headings (LCSH)* [LCSH]. This specification does not specify what list to use but implementors are advised to provide a controlled vocabulary for the list that is used, and to specify the source of subject headings in **subject_heading_collection** field (using, for example, values "SEARS," "LCSH," or "MeSH."

Classification code (call number) is usually either Dewey decimal or Congress classification. But this specification does not prescribe it. Note that the classification codes are unique (unlike ~~in contradiction to~~ some subject headings). Therefore, they can be even expressed as identifiers using the same notation as used for the citation identifiers (*repository/id*).

public BibRefDescription description;

An account of the content of the cited resource. It is either an abstract, or table of contents, or both. It can be written in a language different from the language of the cited resource.

Both abstract and table of contents can contain more than just a plain text – typically they may be expressed in a "markup" language. Their formats are defined according to the specification *Multipurpose Internet Mail Extensions (MIME)* [MIME]. Precisely, the values of attributes **abstract_type** and **toc_type** are equivalent to the "Content-Type Header Field" of the MIME specification, with exclusion of the keyword "Content-Type." For example, an **abstract_type** can have value "text/html," or, using additional parameters "text/plain; charset=us-ascii." If any of these attributes contains an empty string, a default value "text/plain; charset=us-ascii" is assumed.

Often abstracts are also available from the same or separate repository as URLs. There are several ways to provide this information in the here described data model. The implementations may choose their own way and still remain compliant with this specification. However, the first approach, described below, is recommended to achieve interoperability between implementations.

- Use **abstract_type** "text/url," and put the URL into **the_abstract** field.
- Use **abstract_type** "text/plain; url=*xxxxx*" where *xxxxx* is a URL of the abstract (in this case **the_abstract** may still have a full or partial text of the abstract).

- Use “multi-part” (see [MIME]) in **abstract_type**. In such case **the_abstract** will have both the full or partial abstract text, and a URL.
- Use a dynamic property of **BibliographicReference** class for the URL.

```
valuetype BibRefDescription {
    public wstring the_abstract;
    public string abstract_type;
    public wstring table_of_contents;
    public string toc_type;
    public string language;
}
```

public BibRefScope coverage;

It defines an extent or scope of the content of the cited resource. It can include spatial location (a place name or geographic co-ordinates), temporal period (a period label, date, or date range), or both. Finally, it can have additional dynamic properties such as jurisdiction).

```
valuetype BibRefScope {
    public string spatial_location;
    public string temporal_period;
    public CosPropertyService::Properties properties;
}
```

This specification does not suggest any rules for representing geographical names but implementations may consider “Getty Thesaurus of Geographic Names” [TGN], or MARC lists of countries [MARC-COUNTRIES] and list of geographical areas [MARC-AREAS].

public ProviderList authors;
public ProviderList contributors;
public Provider publisher;

These attributes define the active participants. They may be persons, organizations, or even services. A publisher is responsible for making the resource available. The authors and contributors are in *ordered* lists. The authors and contributors are responsible for creating the contents of the cited resource. There is no formal definition of how this responsibility is divided between them. However, the authors are usually primary creators while contributors may be illustrators, translators, or other creative providers. Their role may be specified in a separate attribute in dynamic properties.

public string rights;

Specifies information about rights over the cited resource. Typically, it contains a rights management statement for the resource, or it refers to a service providing such information. Rights information often encompasses Intellectual Property Rights [IPR], Copyright, and various Property Rights.

If the attribute is empty, no assumptions can be made about the status of these and other rights with respect to the cited resource.

public StringDate date;

Defines a date associated with an event in the life cycle of the cited resource when this resource became available. Usually, it is a date of publishing, however, for not yet published resources, it can be a date of creation.

typedef string StringDate;

Using a **typedef** instead of a simple string indicates that there are some rules implied for the attribute value. The suggested encoding is as defined in a W3C NOTE “Date and Time Formats” [W3CNOTE]. This NOTE defines a profile of ISO8601 standard [ISO8601]. ISO8601 describes a large number of date/time formats and the NOTE reduces the scope and restricts the supported formats to a small number. The profile offers a number of options from which this specification permits the following ones:

- Year

  YYYY (e.g., 2000)

- Year and month

  YYYY-MM (e.g., 2000-12)

- Complete date

  YYYY-MM-DD (e.g., 2000-12-31)

- Complete date plus hours, minutes, and seconds

  YYYY-MM-DDThh:mm:ssZ (e.g., 2000-12-31T23:59:59Z)

YYYY	four-digit year
MM	two-digit month (01=January, etc.)
DD	two-digit day of month (01 through 31)
hh	two digits of hour (00 through 23)
mm	two digits of minute (00 through 59)
ss	two digits of second (00 through 59)

Exactly the components shown here must be present, with exactly this punctuation. Note that the “T” appears literally in the string, to indicate the beginning of the time element, as specified in ISO 8601.

Times are expressed in UTC (Coordinated Universal Time), with a special UTC designator (“Z”), again as specified in ISO 8601.

For query purposes, the format with fewer details is considered as having all possible values in place of missing details. Thus, YYYY-MM would mean all dates and times in the given month.

This specification expects that all attributes dealing with dates will use the same rules.

public string language;

Defines a language of the intellectual contents of the cited resource. The recommendation is to use values as defined by RFC1766 [RFC1766] which includes a two-letter Language Code (taken from the ISO639 standard, followed optionally by a two-letter Country Code (taken from the ISO3166 standard). For example, “en” for English, “fr” for French, or “en-uk” for English used in the United Kingdom. Another possibility is to use MARC List of Languages [MARC-LANG].

public string format;

Describes the physical or digital manifestation of the cited resource. It can have very different content depending on the citation type. Therefore, it is highly recommended to use the mechanism of repository introspection to find possible values.

public EntryStatus status;

Defines information related to the citation itself rather than to the cited resource.

typedef VocabularyString RepositorySubset;

```
valuetype EntryStatus {
    public StringDate last_modified_date;
    public RepositorySubset subset;
    public CosPropertyService::Properties properties;
};
```

Some bibliographic repositories consist of several, or even many, databases. The **subset** helps to locate the citation.

The **last_modified_date** can be used to retrieve new or revised (since a specified date) citations.

The dynamic properties may be used to add features related to the citation itself (for example, a name of the citation annotator, or citation version).

2.1.6.2 Derived types from BibliographicReference

The **BibliographicReference** class is a parent class for derived classes representing bibliographic references to specialized bibliographic resources. The names of the derived classes can be used in constructing queries. The following classes are defined by this standard:

- Book
- Article
- Book Article
- Journal Article
- Patent

- Thesis
- Conference proceeding
- Technical Report
- Web Resource

The other resource types, those not mentioned here, can still be accessed and queried by this standard using dynamic properties and attribute **type** of the class **BibliographicReference**.

```
valuetype Book : truncatable BibliographicReference {
    public string isbn;
    public string volume;
    public string edition;
    public string series;
    public Provider editor;
};
valuetype Article : truncatable BibliographicReference {
    public string first_page;
    public string last_page;
};
valuetype BookArticle : truncatable Article;
    public Book from_book;
};
valuetype JournalArticle : truncatable Article {
    public string volume;
    public string issue;
    public string issue_supplement;
    public Journal from_journal;
};
valuetype Patent : truncatable BibliographicReference {
    public string doc_number;
    public string doc_office;
    public string doc_type;
    public StringList applicant;
};
valuetype WebResource : truncatable BibliographicReference {
    public string url;
    public unsigned long estimated_size;
    public string cost;
};
valuetype Thesis : truncatable BibliographicReference {
};
valuetype Proceeding : truncatable BibliographicReference {
};
valuetype TechReport : truncatable BibliographicReference {
};
```

2.1.6.3 Provider

The class **Provider** and its sub-classes define active participants of the process of creation and dissemination of the bibliographic resources. The most obvious examples are authors, but it includes also publishers and other contributors.

```
valuetype Provider {
    public CosPropertyService::Properties properties;
};
typedef sequence <Provider> ProviderList;

valuetype Person : Provider {
    public wstring surname;
    public wstring first_name;
    public wstring mid_initials;
    public string email;
    public wstring postal_address;
    public wstring affiliation;
};

valuetype Organization : Provider;
    public wstring name;
};

valuetype Service : Provider {
    public wstring name;

};
```

The participants can be people, organizations, or even software services (mainly used for new digital resources). This specification does not provide any unique identifiers for these providers (but does not preclude having such identifiers as dynamic properties).

Note that a person's affiliation is not expressed as an instance of **Organization** because it would be out of scope of this specification to define the personal domain. However, for example, nothing precludes the implementation populating affiliations with stringified identifiers of institutions.

2.1.6.4 Journal

```
valuetype Journal {
    public wstring name;
    public string issn;
    public string abbreviation;
    public CosPropertyService::Properties properties;
};
```

A class describing journals. The citations referring to the journal articles have a reference to this class. There are only few explicit attributes defined, the rest are accessible using dynamic properties.

public string issn;

A standard number for journals. Be aware, however, that in the real world even this attribute may change over time. Therefore, it is not suitable as a primary unique identifier for journals.

public wstring name;

A journal title. The list of available titles can be provided using a controlled vocabulary (taken, for example, from MEDLINE [MEDLINE-J1]).

public string abbreviation;

An abbreviation of the journal title. Note that some repositories use more abbreviation systems. For such cases, use dynamic properties for additional abbreviations. (An example for biological journals is in [BIO-ABBR].)

2.2 The DsLSRBibQuery Module

2.2.1 Overview

The module **DsLSRBibQuery** allows searching for and retrieving citations from a bibliographic repository. It uses class and attribute names defined in the data model in module **DsLSRBibObjects**.

The queries return collection of citations that are again query-able. When a client is satisfied with the query results, the collection contents can be retrieved either as a list of citations, or as an XML document.

Querying itself can be done using different approaches:

- Using direct methods, which are convenient for typical queries. This includes also very vague queries of the type “give me everything about pathology.”
- Using aggregate methods for counting resulting citations.
- Using a query language (OMG Constraint language).
- Using SQL or OQL.

The module also allows introspection of the underlying repository. This capability, for example, helps to build more sophisticated and user-friendly clients.

2.2.2 Simple and Qualified Attribute Names

There are several places where clients need to know exact attribute names:

- The citations are retrievable by specifying combinations of attribute names and values.
- The query results are citation records but not necessarily fully populated – they may be restricted only to a subset of attributes.

- The results may be ordered by one or more attributes.

Therefore, this standard defines some rules how to specify attribute names whenever they have to be expressed as strings. The existence of these rules will make the implementations interoperable even for attributes that are not directly named in the specification (those hidden in dynamic properties).

The following rules define how to create stringified names for individual attributes.

1. The stringified names of attributes of class **BibliographicReference** are equal to the member names of this class. For example, “identifier,” “type,” “title,” “authors.”

2. The stringified names of attributes of sub-classes derived from **BibliographicReference**, and of attributes of other classes, are also equal to the member names but additionally they must be *qualified* by the class name using two underscores (“__”). For example, “book__isbn,” “journalarticle__from_journal,” “journal__name.”

Note – The somewhat unusual “double underscore” is used here because underscore is the only non-alphabetic character allowed for variables in the OMG Constraint language. We prefer to use a slightly unusual syntax that remains fully compliant with the language.

---

3. The qualification part of the stringified name (together with underscores) can be omitted if there is no ambiguity. For example, if an implementation does not use property name “isbn” anywhere else, the “book__isbn” can be replaced by “isbn.” It can be omitted also when the usage specifically allows it. For example, a query (as described in Section 2.2.8, “Querying,” on page 2-24) allows a client to ask for all citations related to a given “location” regardless of the citation type – in this case a stringified attribute would be “location” and not, for example, “book__location.”

   Be aware, however, that dropping the qualifier may compromise extendibility because a client that expects a unique attribute name may break if another citation type is added with the same attribute name.

4. The stringified names of the attributes from dynamic properties are equal to their property names, applying the rule about qualification as defined above. Thus, for example, an attribute “registry_number” hidden in member “properties” of class **BibliographicReference** will be stringified as “registry_number,” and an attribute “location” hidden in member “properties” of sub-class **Book** will be stringified as “book__location.”

5. The stringified names of the attributes from dynamic properties of the class **BibliographicReference** for instances without their own sub-class must be qualified (as described above) by their “type.” For example, a citation can be of type “letter,” but there is no sub-class “Letter” defined in this specification. Therefore an attribute “type” has value “letter.” This value is then used to create a qualified stringified name “letter__subject.”

6. The stringified names are considered case-insensitive. Thus, “book__location” is the same as “Book__location,” and “journalarticle__issue” equals to “JournalArticle__issue.”

2.2.3 Query Constraint Language

If the search cannot be accomplished by any direct method (see Section 2.2.1, “Overview,” on page 2-13), then a query language has to be used. This specification proposes to use the “OMG Constraint Language” as the main tool for searching by combination of attributes.

OMG Constraint Language [OMG_CL], sometimes called “CORBA standard constraint language” was designed for the Trading service but can also be used, without any changes, for specifying queries. As a language it is similar to the WHERE clause of the SELECT statement in SQL (however, here it is used independently of SQL).

This specification does not include the description of the language (see [OMG_CL] for details) but defines some rules for property names used by the language, and shows a few examples.

A “property name” is a basic element of the constraint language. In the queries used by this specification, the property names are represented by “stringified attributes names,” as defined in Section 2.2.2, “Simple and Qualified Attribute Names,” on page 2-13, applying the following additional searching rules.

1. If a stringified attribute name represents an attribute of a basic type (string, short, etc.), then the corresponding value in the query expression shall be of the same type. For example:

   JournalArticle__volume == “XX”*(correct)*

   JournalArticle__volume == 32*(incorrect, because volume is string)*

   date == “1999-12”*(correct)*

   date == “1999” *(correct)*

   date == 1999 *(incorrect, because* date *is* string*)*

2. If a stringified attribute name represents an attribute of a constructed type (BibRefSubject, Person…), then the corresponding value in the query expression shall be of type *string*. Additionally, the implementations are advised to search all reasonable members of the constructed type for the given value. For example:

   BibRefSubject == “pathology”

   will be looked for in “keywords,” and “subject headings,” and depending upon what the implementation considers reasonable, also in “codes” of **BibRefSubject**.

3. If a stringified attribute name represents an attribute of a list type (ProviderList, KeywordList, etc.), then the stringified name can be shortened by the plural “s” at the end, and the search is done only for the first element of the list. This is primarily used for finding the first author:

authors == “Linus”	will search “Linus” in all authors,
author == “Linus”	will search only in the first author.

4. If a stringified attribute name is ambiguous then the search should be done for all reasonable representations of this attribute. For example:

book__location == “shelves”	will look for all books on shelves,
location == “shelves”	will look for citations of all types on shelves
type == “book” AND location == “shed”	will look for books in a shed.

2.2.4 Query Matching and Ordering Criteria

Several methods dealing with queries and sorting use a list of criteria. The criteria define how the matching or ordering should be done.

The introspection capability gives access to all criteria provided by the underlying repository.

```
enum CriterionType {
    QUERY_CRITERION,
    SORT_CRITERION
};
```

The criteria can be used for defining rules for matching (type **QUERY_CRITERION**), or for ordering (type **SORT_CRITERION**).

```
valuetype Criterion {
    public VocabularyString name;
    public CriterionType type;
    public VocabularyStringList mutually_exclusive_with;
};

typedef sequence<Criterion> CriterionList;
```

Each **Criterion** is identified by its name. A list of criteria names is used in methods for querying and sorting. The implementations are advised to use descriptive names. For example, the names for matching can be:

“match all words”

“match any word”

“case insensitive”

“case sensitive”

“partial word match”

“full word match”

and the names for ordering can be:

"ascending"

"descending"

Another example of how to use Criteria is to allow regular expressions in queries. Not every implementation is supposed to have the capability of matching by regular expressions but those who have can specify (and document), for example, criterion with name "regular expression."

Each Criterion can also have a non-empty list of other criteria that it is *mutually exclusive* with. For example, a sort criterion "ascending" will probably have "descending" in its **mutually_exclusive_with** list.

The valuetype **Criterion** is used only in introspective methods in order to find a list of supported criteria, their types, and their mutual relationships. In querying and sorting methods, only the lightweight ~~**CriterionList**~~ **VocabularyStringList** is used.

The ~~**CriterionList**~~ **VocabularyStringList** is used always as an *inout* parameter. It allows an implementation to return a list of criteria that were actually used. The returned values can differ from the client's wishes if an implementation does not support a particular function (such as proximity search), or if some criteria were mutually exclusive.

2.2.5 Lists of Stringified Attribute Names

On several occasions, the query methods require a list of attribute names. The lists indicate that some action should be done only for or only with a given set of attributes. The attributes are stringified as described in Section2.2.2, " Simple and Qualified Attribute Names," on page 2-13.

2.2.5.1 excluded attributes

The query results are instances of the **BibliographicReference** class with all attributes. But sometimes it may be better to avoid transferring long data if the client is not interested in it. Typical examples are abstracts. To achieve this, a client can specify a list of attributes that it is not interested in. Such attributes are then returned empty.

The "emptiness" means null wherever possible, or an empty string, or a number zero.

If a **BibRefCollection** was created in several steps (e.g., by navigational query) only the last used "searched attributes" are remembered.

Another use for the "excluded attributes" is in retrieval methods. A query collection, once created, can be asked to return more lightweight citations by specifying such list. Note that the same query collection can be asked to retrieve again the same citations but with a different set of the excluded attributes.

2.2.5.2 searched attributes

The method **find()** (described later) also uses a list of attributes to specify which attributes should be searched. This list is also created from the stringified attribute names.

2.2.6 Repository Introspection

The capabilities for introspection of the underlying repository are present in order to learn in advance what citation attributes (and other data) are provided. The methods of introspection deal both with metadata and data. It means that in some cases they return information on supported attribute names (and similar), and in some cases they return a list of all possible values for the given attribute.

The introspection (with the exception of searching and sorting criteria) is based on using controlled vocabularies. It allows getting back not only simple names of available attributes but also their descriptions.

There are several categories of introspection, as follows.

2.2.6.1 Global introspection

Provides overall information about the repository. The returned data are not dependent on any particular citation type or repository subset. The global introspection uses directly the **VocabularyFinder** interface with the following pre-defined vocabulary names:

```
const string RESOURCE_TYPES          = "resource_types";
const string REPOSITORY_SUBSETS      = "repository_subsets";
const string SUBJECT_HEADINGS        = "subject_headings";
const string LANGUAGES               = "languages";
const string JOURNAL_TITLES          = "journal_titles";
const string JOURNAL_ABBREV          = "journal_abbreviations";
const string ENTRY_PROPERTIES        = "entry_properties";
```

Vocabulary **RESOURCE_TYPES** contains stringified names of all citation types stored in the repository. The names of types that are explicitly defined in this specification should be equal to the constant strings for types (see below).

Some bibliographic repositories consist of several databases. Their list can be provided by the vocabulary **REPOSITORY_SUBSET**.

Vocabulary **ENTRY_PROPERTIES** contains names of properties that characterize a bibliographic reference as a repository entry. Such properties are not related to the cited resource but to the reference itself (for example, a name of the annotator).

The rest of the global vocabularies contain all possible values for the corresponding attributes. The implementations may provide additional global vocabularies.

2.2.6.2 Class introspection

Provides metadata about a specified part of a repository, or about specified type of citations. Here belong the introspective methods of the **BibRefUtilities** interface (see Section 2.2.7, “Interfaces,” on page 2-21). They can answer questions such as “What attributes are available for books in this repository?” or “What searching criteria are supported by a particular sub-part of this repository?”

To specify a type of citation, use values returned back by the global vocabulary **RESOURCE_TYPES** (see above). The predefined constants are provided for types explicitly defined in this specification:

```
const string TYPE_BOOK              = "Book";
const string TYPE_ARTICLE           = "Article";
const string TYPE_BOOK_ARTICLE      = "BookArticle";
const string TYPE_JOURNAL_ARTICLE   = "JournalArticle";
const string TYPE_PATENT            = "Patent";
const string TYPE_THESIS            = "Thesis";
const string TYPE_PROCEEDING        = "Proceeding";
const string TYPE_TECHREPORT        = "TechReport";
const string TYPE_WEB_RESOURCE      = "WebResource";
```

In order to specify an attribute name, all of whose possible values are asked for, use predefined values of attribute names:

```
const string ATTR_PROPERTIES        = "properties";
const string ATTR_FORMAT            = "the_format";
const string ATTR_SCOPE             = "coverage";
```

A special case is the attribute **ATTR_PROPERTIES**. It is described in Section 2.2.6.3, “Dynamic introspection,” on page 2-20.

The properties of providers are obtained by a separate method using CORBA **TypeCode** to define a provider type. The interface design is different from the strategy for citation properties because here the number of provider types (classes) is considered final (in contrast to citation types that can be dynamically added without extending this specification).

The introspection mechanism allows to find what attributes are available in the repository. The attributes, however, play two roles: they can be used in query methods (query-able attributes) and/or they can be returned back in the retrieved citations (very often the first role is a subset of the latter one). In order to achieve an interoperable way how to find the attribute roles there are two predefined constants:

```
const string ROLE_ATTR_QUERYABLE= "queryable";
const string ROLE_ATTR_RETRIEVABLE= "retrievable";
```

The constants above are advised to be used anywhere in the description field of a controlled vocabulary entry describing an attribute.

2.2.6.3 Dynamic introspection

Methods of the global introspection return lists of possible values; methods of the class introspection return possible values for a given subset. However, some values returned by those methods can again be vocabulary names and can be used to find another list of values. Such chaining is called *dynamic introspection.* This strategy is particularly useful for property names returned for attribute **ATTR_PROPERTIES**.

Theoretically, such chaining can be repeated more than once. But for practical reasons this specification assumes a maximum of two levels (see an example below).

As with all recursive strategies, there must be a rule defining the end of recursion. In this case, there must be a way to recognize whether a returned value is or is not a vocabulary name. Two approaches can be used:

- Try to use it as a vocabulary name and if it is not a valid vocabulary, the **VocabularyFinder** raises a NotFound exception.
- Try to find the value in a list provided by the **VocabularyFinder** method **get_all_vocabularies**.

Another issue is a namespace for vocabulary names. It is usually easy for an implementation to assure uniqueness of global and class vocabulary names. But with dynamic introspection the namespace can be quite "polluted." Therefore, in order to achieve interoperability, this specification expects that vocabulary names for dynamic introspection will be prefixed by related citation type, grouped together in the same manner as citation identifier (see Section 2.1.6, "Data Structures," on page 2-4). For example, a vocabulary name for property "location" of citation type "Book" will be "Book/location."

Note that this convention is used only for vocabulary names, not for attribute names in queries (for those see rules in Section 2.2.2, "Simple and Qualified Attribute Names," on page 2-13).

Here is an example of how to use dynamic introspection. A query builder needs to introspect a citation type "Book" to create a graphical user interface with book properties and possible values:

1. Use global introspection to be sure that the given repository has citations about books.

   list = VocabularyFinder::get_vocabulary_by_name (RESOURCE_TYPES);

   Check if the returned list contains string **TYPE_BOOK**.

2. Find all dynamic properties for books (additionally to the explicit properties).

vocabulary_name = BibRefUtilities::supported_~~citation~~bibref_properties (TYPE_BOOK, ATTR_PROPERTIES);

vocabulary = VocabularyFinder::get_vocabulary_by_name (vocabulary_name);

3. Use the vocabulary to retrieve all properties. Now the query builder has the names of all book attributes that can be used in constructing queries.

4. Some properties obtained in the previous step have a controlled set of possible values. Investigate, for example, a property “location.” Create a vocabulary name.

vocabulary_name = TYPE_BOOK + “/” + “location”;

5. Find whether the vocabulary name does represent an existing vocabulary (do it either by checking list of “**get_all_vocabularies**” or by calling “**get_vocabulary_by_name**”).

6. Use the vocabulary to retrieve all possible values for book locations.

2.2.7 Interfaces

2.2.7.1 BibRefCollection

The main entry point to the bibliographic query service is an interface **BibRefCollection**.

```
interface BibRefCollection : CosQuery::QueryEvaluator,
                             CosLifeCycle::LifeCycleObject {
                    ...
};
```

At the beginning, this interface represents citations of the whole bibliographic repository. Later, various query methods return objects of the same type, but now representing only a subset of the repository. This way, a client can make results finer and finer using the navigational query.

The most important methods of this interface that provide searching and retrieval are described below in separate sections. Here are the remaining methods:

```
BibRefCollection sort (in AttributeList ordered_by,
                       inout VocabularyStringList~~Criteria~~ criterions)
    raises (LimitExceeded);
```

It allows ordering citations in a collection. It returns an ordered collection. The order direction and other sort criteria (such as case-insensitive or lexical sort) can be given in “criterions” (see Section 2.2.4, “Query Matching and Ordering Criteria,” on page 2-16).

However, the implementation may refuse to sort excessively large collections by raising a LimitExceeded exception. Imagine, for example, a request to sort the collection representing the whole repository.

XMLstring export (
in DsLSRBibObjects::BibliographicReference the_citation);

This method converts a bibliographic reference into an XML representation using the same rules as exporter methods in *BibRefCollection* representing a query collection where **the_citation** comes from.

2.2.7.2 *BibRefIterator*

The data from any **BibRefCollection** can be returned to the caller directly as a list, or through an iterator, or using a combination of both.

BibRefIterators (designed in the same way as in [MAPS]) are objects that 'point to' elements in a set, and which can be used to 'step through' the set. During this stepping process, each element is visited once. If the underlying set is ordered, this ordering is also preserved in the output of the iterator methods. If, during the iteration, the underlying result set changes (e.g., by another process), an exception InvalidIterator is thrown.

```
interface BibRefIterator {
    boolean next (…) …
    boolean next_n (…) …
    void reset();
    void destroy();
};
```

The most important methods for retrieving data are described in detail later. The remaining methods are described here:

Calls to **reset()** re-position the iterator such that subsequent calls to **next()** or **next_n()** start at the beginning of the result set. It raises CORBA::NO_IMPLEMENT exception if the iterator cannot be reset (for example when the iterator provides access to streaming data).

The **destroy()** method is used to indicate that the iterator is no longer needed, and deletes the iterator object.

2.2.7.3 *BibRefExporter*

This interface has the same behavior as **BibRefIterator** – except for the fact that it returns citations as an XML stream. The exporting methods are described later.

```
interface BibRefExporter {
    boolean export_next (…) …
    boolean export_next_n (…) …
    void reset();
```

```
    void destroy();
};
```

2.2.7.4 *BibRefUtilities*

```
interface BibRefUtilities {
    ...
};
```

This interface provides methods for repository introspection, for managing several citation collections, and other utility methods.

```
readonly attribute
    DsLSRControlledVocabularies::VocabularyFinder voc_finder;
```

A **VocabularyFinder** is a provider of all controlled vocabularies used for finding dynamic attribute names and their allowed values. Section 2.3, “The DsLSRControlledVocabularies Module,” on page 2-29 describes how to use the finder, and Section 2.2.6, “Repository Introspection,” on page 2-18 advises which vocabulary names to use.

```
string supported_bibref_properties (in string bibref_type,
                                    in string attribute_name)
    raises (NotFound);
```

This method returns the name of a vocabulary containing all possible values of attribute **attribute_name** for citation **bibref_type**. For attribute name **ATTR_PROPERTIES**, it returns a list of property names. They represent dynamic properties for the given citation type.

It raises a NotFound exception if no such vocabulary exists.

```
string supported_provider_properties (
    in CORBA::TypeCode provider_kind)
    raises (NotFound);
```

This method returns the name of a vocabulary containing dynamic property names for the given provider type. **provider_kind** is supposed to be a **TypeCode** of class **Person**, **Organization**, or **Service**.

It raises a NotFound exception if no such vocabulary exists.

```
CriterionList supported_criterions (
    in DsLSRBibObjects::RepositorySubset repository_subset)
    raises (NotFound);
```

This method returns a list of all supported searching and sorting criteria for the given repository subset. The values of **repository_subset** may be obtained from a global vocabulary **REPOSITORY_SUBSETS**.

```
BibRefCollection union_it (in BibRefCollectionList collections)
```

raises (LimitExceeded);

A method **union_it()** creates one collection from a list of collections. The resulting collection should contain only unique citations. It can be used to remove repeated citations from different repositories. However, this would be quite a difficult task, and implementations are not required to do so. The implementation can also throw an exception LimitExceeded when the collections are too big.

~~**XMLstring export (**~~
~~**in DsLSRBibObjects::BibliographicReference the_citation);**~~

~~This method converts a bibliographic reference into an XML representation using the same rules as exporter methods in Section 2.2.7.1, "BibRefCollection," on page 2-21.~~

2.2.8 *Querying*

2.2.8.1 *Query by the direct methods*

There are several convenient methods that can be used for querying without using any query language.

DsLSRBibObjects::BibliographicReference find_by_id (in string id)
in ~~string~~ DsLSRBibObjects::BibRefIdentifier id,
in AttributeList excluded)
raises (CosQuery::QueryInvalid, NotFound);

It matches attribute **BibRefIdentifier** in the **BibliographicReference** class and returns a corresponding citation (perhaps without attributes specified in the **excluded** list). Remember that this identifier is meant to be unique even across various repositories (see description of "BibRefIdentifier" in Section 2.1.6.1, "BibliographicReference," on page 2-4). The implementation may raise a QueryInvalid exception if the identifier is not in the scope of the searched repository or it throws a NotFound exception if the searched value is in the scope but cannot be found.

BibRefCollection find_by_author (
in DsLSRBibObjects::Provider author,
in AttributeList excluded,
inout VocabularyStringList~~Criteria~~ criterions);

This is a convenient method for a common query. The search is done only for attributes having non *empty* values in parameter "author." For example, a search for citations written by authors with surname "Doe" can be specified by sending an instance of **Person** with surname "Doe" and with other attributes filled with empty strings. Alternatively, a search for institution "EBI" can be specified by sending an instance of **Organization** with name containing "EBI."

The returned citations can have some attributes empty if parameter "excluded" is used (see "excluded attributes" above). The search can be influenced also by parameter "criterions" (see Section 2.2.4, "Query Matching and Ordering Criteria," on page 2-16).

When no matching authors are found, the implementation should return an empty collection. However, it can still return changed **criterions** indicating why the query failed.

```
BibRefCollection find (
    in PhraseList phrases,
    AttributeList searched,
    AttributeList excluded,
    inout VocabularyStringListCriteria criterions)
        raises (CosQuery::QueryInvalid);
```

This is the most powerful direct method for querying a repository. It is modeled on examples of web-based searches. A client can specify virtually anything in the list of "phrases" and the implementation tries to search for these in as many attributes as are possible and reasonable, applying logical "AND" between them. However, a client can also specifically limit the search only to attributes specified in the "searched" list.

Again, parameters "excluded" and "criterions" can influence the returned results (as described above).

An implementation can raise a QueryInvalid exception if the search demand cannot be accepted (for example, because of repository size, or because of limited indexing capabilities of the repository). Note that this is not the same as raising the NOT_IMPLEMENTED exception, because the client can still use this method but possibly with more restricted "searched" parameters.

2.2.8.2 *Query by the aggregate methods*

These are methods providing functionality similar to some SQL constructs, but without using SQL.

```
unsigned long num_bibrefs();
```

It returns the number of citations in the current collection.

2.2.8.3 *Query by the Constraint Language*

When direct methods (as described above) are not sufficient, a constraint query language is used. The usage of the language is described in Section 2.2.3, "Query Constraint Language," on page 2-15. It is used by a method "evaluate" inherited from **QueryEvaluator** interface from **CosQuery** module.

```
any evaluate (
    in string query,
    in QLType ql_type,
    in ParameterList params)
        raises (QueryTypeInvalid, QueryInvalid, QueryProcessingError);
```

The exceptions are defined in **CosQuery** module:

- The query language type specified must be supported, otherwise QueryTypeInvalid exception is raised.
- If the query syntax or semantics is incorrect or if the input parameter list is incorrect, the QueryInvalid exception is raised.
- If any error is encountered during query processing, the QueryProcessingError exception is raised.

The following rules describe how to use this method. The rules are applied only if the query language used is of type **OMG_CLQuery** defined as:

```
interface OMG_CLQuery : CosQuery::QueryLanguageType {};
```

1. It returns a **BibRefCollection** type.
2. The parameter **query** contains a query in OMG Constraint Language as defined in [OMG_CL]. A QueryInvalid exception is raised if the query syntax or semantics is incorrect.
3. The **params** can contain a property named "criteria" with an instance of ~~**StringList**~~ **DsLSRControlledVocabularies::VocabularyStringList** containing a list of criteria names, and a property named "excluded" with an instance of **AttributeList**. The implementation uses these parameters in the same way as described by the method **find()** above, except that the changed criteria are not returned back.

2.2.8.4 *Query by SQL and OQL*

Some complex queries can be achieved neither by direct methods nor by the OMG Constraint Language. For example, the results are always lists of **BibliographicReference**s. One cannot ask questions like "What journals are cited in this repository?"[1].

Therefore, this specification allows the use of the **evaluate()** method from **CosQuery** without the restrictions described above, but applying a different set of rules:

1. An implementation is still compliant if it supports only the **OMG_CLQuery** type. It may raise an exception QueryTypeInvalid when a different type is used.
2. SQL and OQL queries will be interoperable only if query strings contain stringified attributes names (as described in Section 2.2.2, "Simple and Qualified Attribute Names," on page 2-13) and table/class names equal to class names used in this specification.

---

1. Theoretically, you can do it: Ask for *all* citations of type "JournalArticle," then take only values of an attribute "from_journal," make it unique, and you have a list of available journals. But, in practice, this sort of query probably raises an exception LimitExceeded.

Note that this is quite a serious limitation. For example, it does not allow usage of foreign keys, or joining tables associated with many-to-many-relationships.

Note that some implementations can still use this specification with the full SQL query capabilities if they expose their data model (in documentation, for example). However, such implementations will be rarely interoperable.

2.2.9 Retrieving Citations

A **BibRefCollection** represents the number of **BibliographicReference**s. There are methods to retrieve them and get data to the client.

DsLSRBibObjects::BiblRefIdentifierList retrieve_all_ids()
raises (LimitExceeded);

It returns a list of identifiers of all bibliographic references from a given collection. An implementation can raise a **LimitExceeded** exception if the number of returned identifiers causes problems.

DsLSRBibObjects::BibliographicReferenceList retrieve_all_elements(
in AttributeList excluded)
raises (LimitExceeded);

It returns a list of valuetypes with bibliographic references. Some attributes may be empty if the “excluded” parameter was used. An implementation can raise a LimitExceeded exception if the number of returned citations causes problems.

typedef string XMLString;

XMLstring export_all_elements()
raises (LimitExceeded);

This method also returns all bibliographic references represented by the given collection (also making excluded attributes empty), but as an XML string. It is suitable for exporting data in an exchangeable format for further processing. Again a LimitExceeded exception is raised when the size of data causes problems.

The returned XML string must correspond to a known XML DTD. The following rules apply:

1. If the underlying repository provides its own, well-known DTD for its citations, the method is advised to use it.
2. Otherwise, the DTD used should be compliant with XMI [XMI] rules applied to class **BibliographicReference.**

Both retrieval methods can raise a LimitExceeded exception. This can be avoided by getting data back in parts, using **BibRefIterator** or **BibRefExporter**. The first can be obtained using:

BibRefIterator create_iterator (in AttributeList excluded);

The iterator has the following methods for getting data as **BibliographicReference**s:

```
boolean next (
    out DsLSRBibObjects::BibliographicReference the_citation)
    raises (IteratorInvalid);
boolean next_n (
    in unsigned long how_many,
    out DsLSRBibObjects::BibliographicReferenceList citations)
        raises (IteratorInvalid, LimitExceeded);
```

Iteration using these methods can be done in steps using the **next()** method, which returns a citation as the **out** parameter. Alternatively, when using the **next_n()** method, a batch of at most **how_many** citations are returned in the **out** parameter. If the retrieval was successful, the **out** parameter contains the next citations. TRUE is returned if the call did not yet exhaust the iteration (i.e., if more elements are available for subsequent calls to **next()** or **next_n()**). Conversely, a FALSE return value signifies that no more elements are available from the iterator. If, in a call to **next_n()**, less than the requested **how_many** elements can be returned, the **out** parameter contains as many elements as were available, and the return-value is FALSE.

The **next()** and **next_n()** methods can fail (e.g., if the underlying set changed). In this case, the IteratorInvalid exception is raised. Its **reason** member can be used to provide human-readable information on details of the failure.

Empty result sets (such as from queries yielding no matches) are not represented by NULL objects, but by real iterators that are 'empty' (i.e., invoking their **next()** or **next_n()** methods only ever return FALSE).

The returned citations may have some attributes empty if the iterator was created with a non-empty **excluded** attribute list.

In the same way as just described, a client can ask for data in XML format. It is done using an instance of **BibRefExporter**:

```
BibRefExporter create_exporter();
```

The implementation must guarantee that each returned part is a valid (and, of course, well-formed) independent XML document. The method **export_next()** is a twin to the method **next()** and method **export_next()** to the method **next_n()**:

```
boolean export_next (out XMLString the_citation)
    raises (IteratorInvalid);
boolean export_next_n (
    in unsigned long how_many,
    out XMLString citations)
        raises (IteratorInvalid, LimitExceeded);
```

2.3 The DsLSRControlledVocabularies Module

When describing and representing domain-specific systems, there is frequently a need for a string type that can only assume a limited set of allowed values; a set, however, that is allowed to change over time (as values are added or removed) or space (different servers accepting different sets of strings). Such strings are called *controlled vocabulary* strings ("vocabulary strings" for brevity). A particular set of such strings, valid in some context, is called a controlled vocabulary. Vocabulary strings typically denote domain-specific concepts, usually as a short descriptive string or common abbreviation, rather than as a code.

The IDL specification of such vocabularies is taken from the LSR Genomic Maps Specification [MAPS] where there is also detailed explanation of how to use them. This specification describes only those issues related to their usage in the bibliographic context.

All clients can receive all vocabularies from one central vocabulary finder. The finder can be a shared object. The clients get it from interface **BibRefUtilities**.

References A

A.1 List of References

1. [DC] Dublin Core Metadata Element Set, Version 1.1: Reference Description
http://purl.oclc.org/docs/core/documents/rec-dces-19990702.htm

3. [CORBA2.3] CORBA/IIOP 2.3.1 Specification, OMG document
formal/99-10-07
http://www.omg.org/corba/corbaiiop.html

4. [BSA] Draft adopted specification for Biomolecular Sequence Analysis
http://www.omg.org/cgi-bin/doc?lifesci/99-12-01

5. [MAPS] Genomic Map Draft Adopted Specification
http://www.omg.org/cgi-bin/doc?dtc/99-12-01

6. [INS] Interoperable Naming Joint Revised Submission
http://www.omg.org/cgi-bin/doc?orbos/98-10-11

7. [DCT1] List of Resource Types. Dublin Core Draft Working Group Report.
http://purl.org/dc/documents/wd-typelist.htm

8. [SEARS] Sears List of Subject Headings (16th Ed, July 1997),
Minnie Earl Sears, Joseph Miller, ISBN: 0824209206

9. [LCSH] Library of Congress Subject Headings, 22nd edition (1999)

10. [MIME] Multipurpose Internet Mail Extensions (MIME), Part One, Internet RFC 2045
http://www.ietf.org/rfc/rfc2045.txt

11. [TGN] Getty Thesaurus of Geographic Names
http://shiva.pub.getty.edu/tgn_browser/

[MARC-COUNTRIES] USMARC Code List for Countries
http://lcweb.loc.gov/marc/countries/

[MARC-AREAS] USMARC Code List for Geographic Areas
http://lcweb.loc.gov/marc/geoareas/

12. [IPR] Basic information concerning intellectual property
http://www.itds.treas.gov/ITDS/ITTA/ipr.html

13. [W3CNOTE] Date and Time Formats
http://www.w3.org/TR/NOTE-datetime

14. [ISO8601] International Standard for representation of dates and times
http://www.iso.ch/markete/8601.pdf

15. [RFC1766] Tags for the Identification of Languages, Internet RFC 1766
http://www.ietf.org/rfc/rfc1766.txt

[MARC-LANG] USMARC Code List for Languages
http://lcweb.loc.gov/marc/languages/

16. [ISO639] Codes for representation of names of languages

17. [ISO3166] Codes for representation of names of countries

18. [MEDLINE-J1] MEDLINE Journals With Links to Publisher Web Sites
http://www.ncbi.nlm.nih.gov/PubMed/fulltext.html

19. [BIO_ABBR] Biological Journals and Abbreviations
http://arachne.prl.msu.edu/journams/

20. [OMG_CL] OMG Constraint Language, CORBA Services, Trading Object Services Spec.
http://www.omg.org/cgi-bin/doc?formal/97-12-23

21. [XMI] XML Metadata Interchange (XMI)
http://www.omg.org/cgi-bin/doc?ad/99-10-02

OMG IDL B

Note – This appendix is a complete replacement of the original IDL. It has not been marked as inserted text for the sake of readability.

B.1 DsLSRBibObjects.idl

```
//File: DsLSRBibObjects.idl
//

#ifndef _DS_LSR_BIB_OBJECTS_IDL_
#define _DS_LSR_BIB_OBJECTS_IDL_

#pragma prefix "omg.org"

#include <CosPropertyService.idl>

module DsLSRBibObjects {

   typedef sequence<string> StringList;

   typedef string StringDate;

   typedef string BibRefIdentifier;
   typedef sequence<BibRefIdentifier> BibRefIdentifierList;

   typedef wstring Keyword;
   typedef sequence<Keyword> KeywordList;

   typedef string SubjectHeading;
   typedef sequence<SubjectHeading> SubjectHeadingList;

   typedef string ClassificationCode;
   typedef sequence<ClassificationCode> ClassificationCodeList;
```

```
typedef string RepositorySubset;

valuetype Provider {
   public CosPropertyService::Properties properties;
};
typedef sequence<Provider> ProviderList;

valuetype Person : Provider {
  public wstring surname;
  public wstring first_name;
  public wstring mid_initials;
  public string  email;
  public wstring postal_address;
       public wstring affiliation;
};

valuetype Organization : Provider {
   public wstring name;
};

valuetype Service : Provider {
   public wstring name;
};

valuetype Journal {
   public wstring name;
   public string  issn;
   public string  abbreviation;
   public CosPropertyService::Properties properties;
};

valuetype BibRefSubject {
  public KeywordList keywords;
  public SubjectHeadingList subject_headings;
       public string subject_heading_collection;
  public ClassificationCodeList codes;
};

valuetype BibRefDescription {
  public wstring the_abstract;
  public string  abstract_type;
  public wstring table_of_contents;
  public string  toc_type;
  public string  language;
};

valuetype BibRefScope {
  public string spatial_location;
  public string temporal_period;
  public CosPropertyService::Properties properties;
};

valuetype EntryStatus {
  public StringDate last_modified_date;
  public RepositorySubset subset;
```

```
    public CosPropertyService::Properties properties;
};

valuetype BibliographicReference {
    public BibRefIdentifier identifier;
    public string type;
        public BibRefIdentifierList cross_references;

    public wstring title;
    public BibRefSubject subject;
    public BibRefDescription description;
    public BibRefScope coverage;

    public ProviderList authors;
    public ProviderList contributors;
    public Provider publisher;
    public string rights;

    public StringDate date;
    public string language;
    public string format;

    public EntryStatus status;
    public CosPropertyService::Properties properties;
};
typedef sequence<BibliographicReference> BibliographicReferenceList;

valuetype Book : truncatable BibliographicReference {
     public string isbn;
    public string volume;
    public string edition;
    public string series;
    public Provider editor;
};

valuetype Article : truncatable BibliographicReference {
    public string first_page;
    public string last_page;
};

valuetype BookArticle : truncatable Article {
    public Book from_book;
};

valuetype JournalArticle : truncatable Article {
    public string volume;
    public string issue;
    public string issue_supplement;
    public Journal from_journal;
};

valuetype Patent : truncatable BibliographicReference {
     public string doc_number;
    public string doc_office;
    public string doc_type;
```

```
    public StringList applicant;
  };

  valuetype WebResource : truncatable BibliographicReference {
    public string url;
    public unsigned long estimated_size;
    public string cost;
  };

  valuetype Thesis : truncatable BibliographicReference {
  };

  valuetype Proceeding : truncatable BibliographicReference {
  };

  valuetype TechReport : truncatable BibliographicReference {
  };

#pragma version Journal 1.1
#pragma version BibRefSubject 1.1

};

#endif // _DS_BIB_OBJECTS_IDL_
```

B.2 DsLSRBibQuery.idl

```
//File: DsLSRBibQuery.idl
//

#ifndef _DS_LSR_BIB_QUERY_IDL_
#define _DS_LSR_BIB_QUERY_IDL_

#pragma prefix "omg.org"

#include <CosLifeCycle.idl>
#include <CosQueryCollection.idl>
#include <CosQuery.idl>
#include <DsLSRBibObjects.idl>
#include <DsLSRControlledVocabularies.idl>

module DsLSRBibQuery {

  // shorthands for imported types for controlled vocabularies
  typedef DsLSRControlledVocabularies::VocabularyString VocabularyString;
  typedef DsLSRControlledVocabularies::VocabularyStringList VocabularyString-
List;

  typedef sequence<string> AttributeList;
  typedef sequence<wstring> PhraseList;
  typedef string XMLString;

  enum CriterionType {
    QUERY_CRITERION,
```

```
    SORT_CRITERION
  };

  valuetype Criterion {
    publicVocabularyString name;
    public CriterionType type;
    publicVocabularyStringList mutually_exclusive_with;
  };
  typedef sequence<Criterion> CriterionList;

  exception NotFound { string reason; };
  exception IteratorInvalid { string reason; };
  exception LimitExceeded { string reason; };

  interface OMG_CLQuery : CosQuery::QueryLanguageType {};

  interface BibRefIterator {
    boolean next (out DsLSRBibObjects::BibliographicReference the_citation)
  raises (IteratorInvalid);
    boolean next_n (in unsigned long how_many, out DsLSRBibObjects::Biblio-
graphicReferenceList citations)
  raises (IteratorInvalid, LimitExceeded);
    void reset();
    void destroy();
  };

  interface BibRefExporter {
    boolean export_next (out XMLString the_citation)
        raises (IteratorInvalid);
    boolean export_next_n (in unsigned long how_many, out XMLString citations)
        raises (IteratorInvalid, LimitExceeded);
    void reset();
    void destroy();
  };

  interface BibRefCollection : CosQuery::QueryEvaluator, CosLifeCycle::LifeCycle-
Object {

    // direct methods (convenient methods)

    DsLSRBibObjects::BibliographicReference find_by_id (in DsLSRBibOb-
jects::BibRefIdentifier id,
                                in AttributeList excluded)
        raises (CosQuery::QueryInvalid, NotFound);

    BibRefCollection find_by_author (in DsLSRBibObjects::Provider author,
                                     in AttributeList excluded,
                                     inout VocabularyStringList criterions);

    BibRefCollection find (in PhraseList phrases,
                           in AttributeList searched,
                           in AttributeList excluded,
                           inout VocabularyStringList criterions)
        raises (CosQuery::QueryInvalid);
```

```
        // aggregate methods

        unsigned long num_bibrefs();

    // sort methods
        BibRefCollection sort (in AttributeList ordered_by,
                inout VocabularyStringList criterions)
         raises (LimitExceeded);

        // retrieval methods

        DsLSRBibObjects::BibliographicReferenceList retrieve_all_elements (in
    AttributeList excluded)
         raises (LimitExceeded);

        DsLSRBibObjects::BibRefIdentifierList retrieve_all_ids()
         raises (LimitExceeded);

        XMLString export_all_elements()
         raises (LimitExceeded);

        BibRefIterator create_iterator (in AttributeList excluded);
        BibRefExporter create_exporter();

        XMLString export (in DsLSRBibObjects::BibliographicReference the_citation);

    };

    typedef sequence<BibRefCollection> BibRefCollectionList;

    interface BibRefUtilities {

        // constants for global vocabulary names

        const string RESOURCE_TYPES          = "resource_types";
        const string REPOSITORY_SUBSETS      = "repository_subsets";
        const string SUBJECT_HEADINGS        = "subject_headings";
        const string LANGUAGES               = "languages";
        const string JOURNAL_TITLES          = "journal_titles";
        const string JOURNAL_ABBREV          = "journal_abbreviations";
        const string ENTRY_PROPERTIES        = "entry_properties";

        // constants for citation types

        const string TYPE_BOOK               = "Book";
        const string TYPE_ARTICLE            = "Article";
        const string TYPE_BOOK_ARTICLE       = "BookArticle";
        const string TYPE_JOURNAL_ARTICLE    = "JournalArticle";
        const string TYPE_PATENT             = "Patent";
        const string TYPE_THESIS             = "Thesis";
        const string TYPE_PROCEEDING         = "Proceeding";
        const string TYPE_TECH_REPORT        = "TechReport";
        const string TYPE_WEB_RESOURCE       = "WebResource";

        // constants for attribute names
```

```
        const string ATTR_PROPERTIES  = "properties";
        const string ATTR_SCOPE       = "scope";
        const string ATTR_FORMAT      = "format";

        // constants for attributes roles

        const string ROLE_ATTR_QUERYABLE   = "queryable";
        const string ROLE_ATTR_RETRIEVABLE = "retrievable";

    // methods allowing repository introspection

        readonly attribute DsLSRControlledVocabularies::VocabularyFinder voc_finder;

        string supported_bibref_properties (in string bibref_type,
                                            in string attribute_name)
          raises (NotFound);

        string supported_provider_properties (in CORBA::TypeCode provider_kind)
          raises (NotFound);

        CriterionList supported_criterions (in DsLSRBibObjects::RepositorySubset
    repository_subset)
          raises (NotFound);

        // other methods

        BibRefCollection union_it (in BibRefCollectionList collections)
          raises (LimitExceeded);

      };

    #pragma version BibRefCollection 1.1
    #pragma version BibRefUtilities 1.1

    };

    #endif // _DS_BIB_QUERY_IDL_
```