BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

OBJECT DATABASES AND THE SEMANTIC WEB

A THESIS SUBMITTED IN FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY.

ING. JAKUB GÜTTNER

Field of specialization:	Information Systems
Supervised by:	Prof. Ing. Tomáš Hruška, CSc.
State doctoral exam:	June 18, 2003
Submitted on:	June 31, 2004
Availability:	Library of the Faculty of Information Technology, Brno University of Technology, Czech Republic

I would like to give thanks to those who taught me so much:

Dagmar Güttnerová for teaching me to remain steadfast in all I do;
Stanislav and Pavla Güttnerovi for teaching me to learn;
Tomáš Hruška for teaching me to separate the wheat from the chaff;
Alexander Meduna for teaching me the joys of pure research;
and Jesus Christ for teaching me the most important thing, and how to put it all together.

Ing. Jakub Güttner
Brno, June 24, 2004

ABSTRACT

This thesis presents the idea of using object-oriented database mechanisms within the Semantic Web, and provides the theoretical foundations that are necessary for this to happen.

The thesis starts with an overview of object-oriented database standards, data models and formal paradigms, followed by an overview of the Semantic Web RDF/S framework, its model-theoretic semantics, current state of art and emerging applications. The idea of the Semantic Web as a worldwide loosely bound database is then explored by comparing the features of object-oriented data models and RDF/S graphs. After outlining their similarities and the advantages of applying several database concepts to the Semantic Web environment, a formal model of an object database on top of RDF/S is presented. This description is followed by several extensions to the core model (object graph navigation, mining objects from RDF/S and implementation notes) and suggestions of future research

KEYWORDS

Object-Oriented Database, Semantic Web, Data Modeling, RDF, RDFS, CDL, ODMG, SODA

ABSTRAKT

Tato disertační práce předkládá myšlenku využití mechanismů objektově orientovaných databází v prostředí sémantického Webu a pokládá teoretické základy nezbytné k dosažení tohoto cíle.

Disertační práce začíná přehledem současného stavu standardů, datových modelů a formálních paradigmat objektově orientovaných databází, a přehledem současného stavu, teoretických východisek a vznikajících aplikací sémantického Webu a modelu RDF/S. Dále je zkoumán sémantický Web z pohledu celosvětové volně vázané databáze a jeho rysy jsou srovnány s principy objektově orientovaných datových modelů. Po analýze jejich podobností a výhod, které by přinesla aplikace některých databázových konceptů do prostředí sémantického Webu, je představen formální model objektové databáze nad sémantickým základem RDF/S. Tento popis je následován několika rozšířeními základního modelu (navigace v objektovém grafu, získávání objektů z RDF/S a implementační poznámky) a návrhem možných směrů dalšího výzkumu.

KLÍČOVÁ SLOVA

Objektově orientované databáze, sémantický Web, datové modelování, RDF, RDFS, CDL, ODMG, SODA

TABLE OF CONTENTS

2 INTRODUCTION

> There's so much more that could be done. The reason is, in a word, databases: dusty, musty databases filled with useful data that would be far more useful if linked with other, equally dusty databases; enormous databases that are locked up inside ancient mainframes and quaintly archaic minicomputers; lonely databases residing on specialized file servers throughout an enterprise; even modern databases on Web servers, all dressed up and ready to go, but stuck in long-obsolete proprietary formats or accessible only through hypermodern scripting languages.
> **Second-generation e-commerce will depend on unlocking those databases.**[1]

Opening up databases to the world and letting them cooperate on solving problems is not a simple task, because seamless integration of massively heterogeneous technologies requires a carefully designed framework of standards, languages and tools. All of these need to be based on rigorous mathematical foundations that define underlying semantics. This thesis aims at working with such foundations for the integration of two technologies that may be important in fulfilling the above goal — the Semantic Web (chapter 4) and object-oriented databases (chapter 3).

- **Semantic Web** is a W3C initiative that provides standards for creating an interoperable, semantically rich, and machine-readable network of information similar to the World Wide Web — it aims to do for computers what the Web has done for humans. Its theoretical foundations lie in the areas of ontologies, knowledge-based and logical programming, and artificial intelligence. Proposed data structure is very loose, does not differentiate between data and metadata, and allows incomplete or conflicting information.

- **Object-oriented databases** are gaining new popularity with the advent of XML, Web services and business-to-business integration. The leading relational database vendors are now providing object-relational bindings, and both .NET and Java provide object-oriented data management frameworks (ADO.NET, JDO). There is a pressing need to design a unified formalism for the object-oriented data model.

Object-oriented database schemas are much richer than relational ones, and their features come close to ontological languages. Formalizing them with the Semantic Web RDF model theory is the goal of this thesis.

The growing Semantic Web needs to address a number of database issues, such as dynamic behavior, transactions and updates, security and access control, and efficient data storage. Another goal of this thesis is to naturally adapt these concepts from the area object databases.

Object databases can thus gain a widely accepted formal foundation, increased flexibility thanks to the loose format of Semantic Web RDF data, and natural integration with the upcoming Semantic Web.

---

[1] IEEE Spectrum, [Castro-Leon04]

2.1 GOALS OF THIS THESIS

- Explore existing object-oriented database standards and models. Give an overview of the Semantic Web and its database-oriented research and applications.
- Describe the similarities and differences between elements and constructs of object-oriented data models and the Semantic Web RDF/S model. Outline the advantages of merging these two areas.
- Define a formal model of an object-oriented database on top of the RDF/S framework.
- Show how some database concepts can be used in context of the RDF/S based Semantic Web.

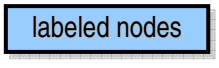
2.2 OVERVIEW OF THE THESIS

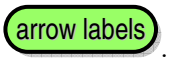
- **Chapter 3**, Overview of Object–Oriented Databases, clarifies the modeling paradigm and design choices of object-oriented databases and overviews different standards, languages and models.
- **Chapter 4,** Overview of the Semantic Web, explains why and how the Semantic Web initiative plans to expand the existing World Wide Web. This chapter gives model-theoretic semantics of the RDF/S model and shows examples of existing RDF research and applications.
- **Chapter 5**, Semantic Web as an Object Database, compares the models of object databases, RDF/S graphs, and the models used in human thinking. The chapter lists similarities between the RDF-based Semantic Web and an object database and advocates a RDF/S object model that would constitute a loosely bound worldwide object database. Different database concepts could then be adopted for the Semantic Web.
- **Chapter 6**, Database Model on Top of RDF/S, gives a formal description of an object model on top of RDF model-theoretic semantics. Design choices and limitations of the model are also discussed. This chapter contains a large example of how the model can be used.
- **Chapter 7**, Extending the Model, discusses three aspects of extending the formal model — why and how to extract object structures from raw RDF/S data, how to access and navigate RDF-based object graphs, and how the proposed model could possibly be implemented.
- **Chapter 8**, Conclusion, sums up the contribution of this dissertation and suggests directions for further research.
- **Appendices** include the list of references, an overview of several interesting Semantic Web projects, a brief survey of RDF query languages, a list of figures and a glossary of the most frequent abbreviations in this thesis.

Chapters 3 – 4 have been edited and summarized from cited sources with the addition of evaluative notes, while chapters 5 – 7 represent author's original contribution.

2.3 TYPOGRAPHIC CONVENTIONS

In normal text, *italics* are used for emphasis, while **bold type** outlines itemized paragraphs and sans-serif font represents symbols and mathematic expressions.

RDF graphs contain 

connected by arrows that have 

.

> Examples are set apart from the rest of the text.

> Equations and formal statements also use a specific paragraph style and font.

3 OVERVIEW OF OBJECT–ORIENTED DATABASES

> Relational database management systems (RDBMSs) are designed to store data according to the most efficient method of data cataloging… In many cases, however, [this] is not the most efficient method for storing and retrieving such data… When dealing with data in complex interdependent structures, or [rapidly retrieving data] by following paths of associations, the relational database begins to show impediments… In some cases, RDBMS must be regarded as impractical for certain data management tasks.[1]
>
> The limitations of the relational data model in terms of its ability to handle complex data types, complex data relationships, and multiple access methods are starting to be recognized. Relational databases… do not scale well to accommodate complex transactions. The unique needs of the Internet… have been a major catalyst in this need for change.[2]

The first widespread database standard was the CODASYL norm of 1980 [CODASYL80] — it formalized the field of *network databases* with records interconnected by physical address references. However, the model suffered from many problems with distribution, consistency checking, and migration. All of these shortcomings were addressed by the *relational approach*, first presented by E. F. Codd in 1970 [Codd70], which remains the prevalent paradigm until today. It stands on a firm mathematical foundation of relational algebra that has been fully used in designing the SQL query language.

The third generation, *object databases*, tries to address problems that arose from the relational approach. These include impedance mismatch between relational datatypes and object-oriented language data structures; lack of support for complex data types and relationships (such as object and type hierarchies, large binary objects and semistructured data); no systematic approach to storing and encapsulating algorithms in the database; and problems with efficient lookup of objects due to indexing by a large number of key types.

DB paradigm	Unique ID	Relationships	Lookup	Embedded data
Network model	Direct physical address	Yes	By address	No
Relational model	Many table-unique key types	No	By values	No
Object model	Single OID type	Yes	By OID	Yes

This chapter presents an overview of data models in object-oriented databases. It introduces multiple standards, languages, implementations, and formalisms, ranging from influential papers like the OODBS Manifesto of 1990, to the recent object-oriented data management standard for application

[1] IDC Consulting white paper on user data management [IDC03]

[2] Dataquest (Gartner Group) Consulting market overview of post-relational databases [Dataquest99]

servers — Java Data Objects 1.0.1 of 2003 (see the chronological outline in Figure 3.1). All sections of this chapter are arranged in a similar structure so the reader can better compare them.

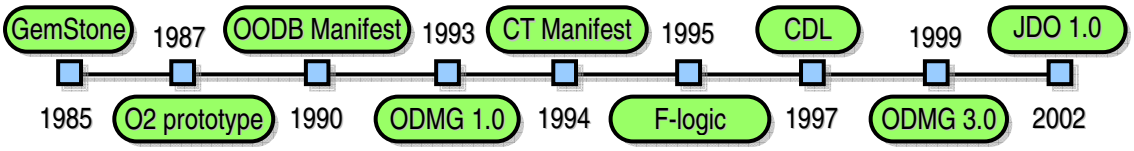


Figure 3.1 A timeline of milestones in OODB data modeling

3.1 Object–Oriented Database System Manifesto

Why Was it Chosen

This Manifesto is one of the first and most influential attempts to characterize the area of object databases. It is not a formal model, and not even a standard, but it points out the most important features that an object-oriented database system should have. Compared to the exact definition of a "relational database", the notion of "object-oriented database" has always been quite informal, and this brief, 15-page document tries to underpin it with several useful axioms.

Introduction

The Object–Oriented Database System Manifesto [Atkinson90] of 1990 is an attempt to summarize:

- **Mandatory features** required of a program to be both a database and an object-oriented system.
- **Optional features** that clearly improve the system, yet the system is still an object-oriented database without them.
- **Open choices** are up to the individual database implementors to decide. In these, the scientific community has not yet reached a consensus and it is not clear which option is the most suitable.

3.1.2 Mandatory Features

Features of an Object-Oriented System

- **Complex Objects.** Complex objects are constructed from simpler ones. Examples of the simplest objects are integers, floats, integers, strings and characters. Examples of constructors are tuples, sets (bags), lists or arrays. Any constructor can be applied to any object. Retrieval, deletion or copying of complex objects is available.
- **Object Identity.** Every object has a unique identifier independent of its value. Objects can be shared through relationships and independently updated.
- **Encapsulation.** Both object data and algorithms (methods) are stored in the database. The only visible part of an object is its interface, while its implementation and physical storage are hidden. However, in certain cases encapsulation may not be appropriate (e.g. for ad hoc queries).

- **Types and Classes.** The database schema is given by a set of classes or types that describe the format of objects. While types (C++, Simula, O2) are mainly a static notion used to ensure program correctness at compile time, classes (Smalltalk, Lisp) are rather first-class citizens used for creating and warehousing objects at runtime. The system should be able to maintain extents of selected classes or types.

- **Class or Type Hierarchies.** The system has support for inheritance between types or classes — it is able to derive more specialized classes or types from existing ones. No specific type of inheritance is prescribed.

- **Overriding, Overloading, and Late Binding.** Different algorithms can have the same name and which one will run is only chosen at runtime. For example, invoking the display operation on a general type gives different results for different types of graphical primitives.

- **Computational Completeness.** Any computable function is expressible using the data manipulation language of the database system.

- **Extensibility.** The set of predefined types must be extensible. There must be no distinction in using system-defined and user-defined types.

FEATURES OF A DATABASE SYSTEM

- **Persistence.** Each object should be allowed to survive the execution of a process and to be reused later without explicit load/store operations.

- **Secondary Storage Management.** The system must supply features such as index management, data clustering, buffering and query optimization. They are so performance-critical that they need to be present if the database is to complete certain tasks in realistic time frame.

- **Concurrency.** The system should ensure harmonious coexistence among multiple users working simultaneously on the database by providing atomicity of operations and controlled sharing.

- **Recovery.** In case of failures, the system should restart to some coherent state of its data.

- **Ad Hoc Query Facility.** The *functionality* of an ad hoc query language should be provided to express simple queries. This does not necessarily require a full query language; for example, a graphical browser can achieve the same thing.

3.1.3 OPTIONAL FEATURES

- **Multiple Inheritance.** An object should be able to inherit from multiple predecessors. Not everyone in the object-oriented community agrees that this should be a required feature.

- **Type Checking and Type Inference.** The degree of compile-time type checking and type inferencing is left open, but the more, the better.

- **Distribution.** Distribution is orthogonal to the object-oriented nature of the system; that is, the database system may or may not be distributed.

- **Design Transactions.** In many applications with long running transactions, the usual serializability requirement is not adequate and other types of transactions are needed (nested, distributed, long etc.).

- **Versions.** Different versions of database contents and their management may be supported.

3.1.4 Open Choices

- **Programming Paradigm.** The choice of logic, functional, imperative, or any other programming style is left to the designers along with language syntax.
- **Representation System.** The choice of specific atomic types and type constructors is left to the designers.
- **Type System.** Any kind of type formers beyond type constructors can be implemented — generic types, restrictions, boolean operations, functions etc. The type system for variables can be richer than the one for objects.
- **Uniformity.** It is up to the designers to decide whether schema information should be stored as normal objects, whether types are first-class citizens in the programming language and whether the user sees any difference between types, objects and methods.

3.1.5 Sources

The OODB Manifesto can be found in [Atkinson90].

3.2 The ODMG Standard

Why Was it Chosen

ODMG 3.0 is a de facto standard for object databases and object-to-database mappings. Members of the Object Data Management Group included major object database vendors like Ardent, POET, Object Design, Objectivity, GemStone, Micro Data Base Systems, Computer Associates and Versant along with other companies (Sun Microsystems, NEC, CERN, Baan, Hitachi, Barry & Associates, Microsoft etc.). In 2001, ODMG activity was suspended.

Introduction

The ODMG standard gives a set of specifications for writing applications that are portable at the source code level among different object data management systems — ones that integrate database capability with object-oriented language features. Thus, object-oriented languages are extended with transparently persistent data, concurrency control, data recovery, associative queries etc.

The ODMG standard was also to correspond to standards efforts such as Java Community Process (in 2003, Java bindings were superseded by JDO — Java Data Objects Specification, section 3.3), OMG (which adopted ODMG-93 in 1994 and OQL in 1995), SQL (the goal of INCITS X3H2 was to converge SQL3 and OQL), C++ and Smalltalk (X3J16 and X3J20).

The components of ODMG 3.0 are:

- **Object Model** to be supported by ODMG implementations — this takes the OMG Object Model which is intended for object request brokers, OODBs, object programming languages and other applications, and adds an OODB *profile* with specific extensions like relationships. The

object model is more thoroughly presented in the following subsection. Its semantics is not formally defined, although its structure is described by ODL metamodel interfaces.

- **Object Specification Languages** describe ODL (Object Definition Language) for defining ODMG data types; and OIF (Object Interchange Format) for migrating the contents of a database in a standard way.

- **Object Query Language** (OQL) is a declarative language based on SQL, but more powerful than SQL, for querying and updating objects.

- **Programming Language Bindings** for C++, Smalltalk and Java explain how to write portable code for manipulating persistent objects. They define a map to and from ODL along with bindings for invoking OQL, managing the database, and executing transactions.

3.2.2 ODMG 3.0 Object Model

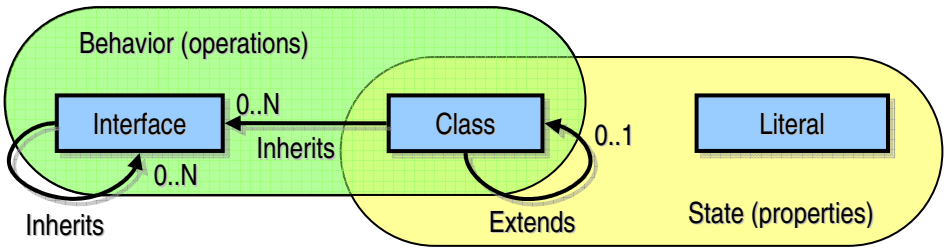


Figure 3.2 Metatypes in the ODMG model

Types

- **Specifications and Implementations.** The ODMG object model supports encapsulation because every type has a *specification* consisting of implementation-independent signatures of operations, properties and exceptions, and one or more *implementations* through prescribed bindings to programming language data structures and methods. There are three kinds of types — interfaces specify abstract behavior (signatures of operations and properties), literals specify abstract state and classes specify both (see Figure 3.2).

> **Classes** can instantiate objects and they contain enough state and behavior information to be incorporated in the OODB schema. Their operations are implemented through methods and properties mapped onto data structures.
> **Interfaces**, on the other hand, cannot be instantiated (they function as "abstract classes"). Implementations are supplied by classes that inherit from them.
> **Literals** are instantiated as data structures with no operations and no OID.

- **Object Types.** Every object or literal has a type and every operation requires typed operands. Type equivalence is only determined by the type's name, no implicit type conversions are provided.

  Atomic Objects. No atomic object types are predefined by ODMG type system, but users may define their own atomic objects with custom properties and behavior.

Collection Objects are composed of a number of instances of the same type (an atomic, collection or literal). Supported collection types are set, bag, list, array, and dictionary. They function as type generators parameterized by the type of their members. Standard operations include tests for membership, emptiness, cardinality, collection-specific boolean operations, concatenation, indexing etc. Collections also have support for OQL queries and iterators.

Structured Object types predefined by the ODMG specification correspond to the INCITS SQL standard. They are Date, Time, Interval and Timestamp, including operations on them.

- **Subtyping and Inheritance.** In subtyping hierarchies, all instances of a type also have the type of all its supertypes. Subtypes share the supertypes' state and behavior signatures, and can be used in their place. Behavior and state of a subtype can be added or specialized (overloading and late binding are supported). The subtyping relationship is transitive.

  There are two kinds of subtyping relationships in ODMG — inheritance and extension. *Multiple inheritance* is supported for sharing abstract behavior of interfaces, while only single *extension relationship* for a given object type allows to share both abstract behavior and state (see Figure 3.2).

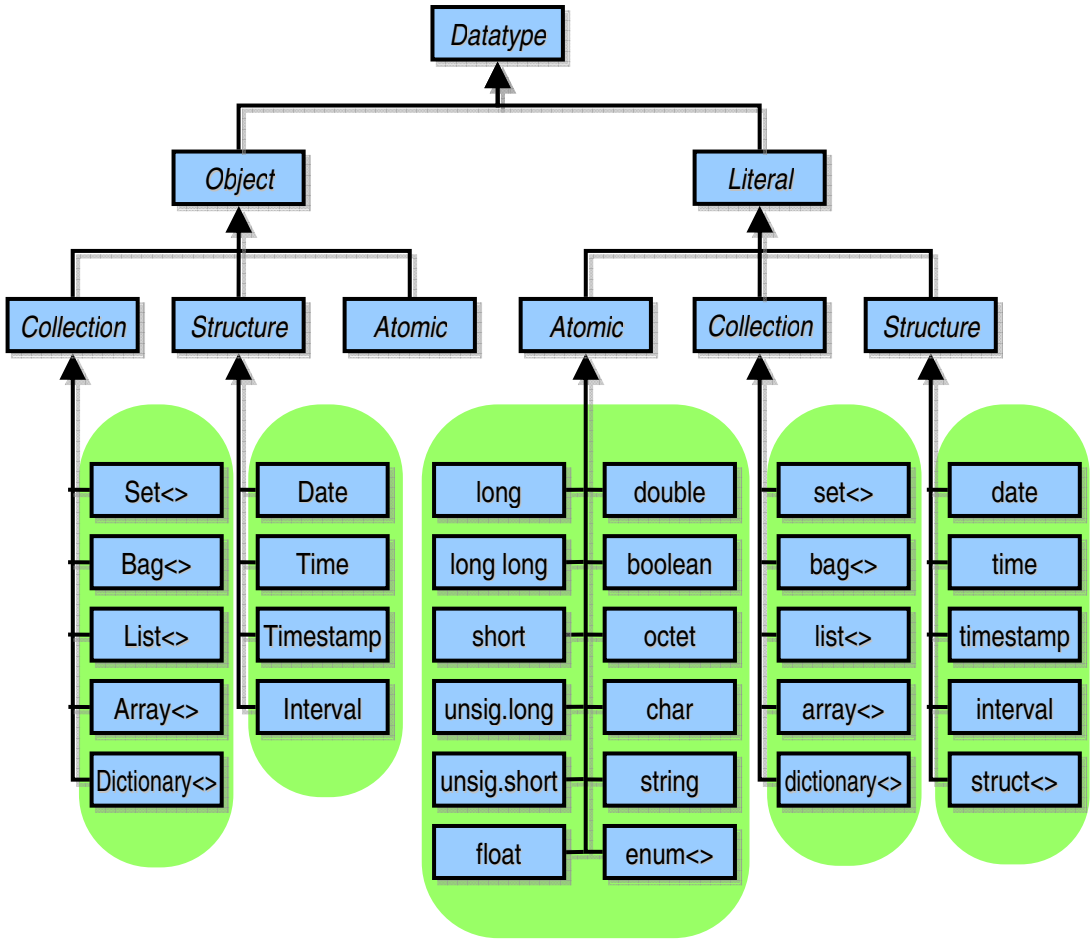


Figure 3.3 The full ODMG built-in type hierarchy (abstract classes italicized)

- **Literals.** Literals do not have unique identifiers, therefore they cannot be referenced and have to be embedded in objects. ODMG supports atomic, collection and structured literal types with characteristics similar to object types. Atomic literal types include types like numbers and characters, and an enumeration type generator enum<>. New structured literal types can be defined

using the struct<> generator. They contain a fixed number of variables with either a literal value or an object, and can be freely composed. For an overview of specific types, see the type hierarchy in Figure 3.3).

Comparison of Literals. Literals cannot be compared on the basis of identity using the same_as() operation; instead, the equals() operation compares them by value and type.

- **Extents and keys.** Extent is the set of all instances of a given type (and its subtypes). The database designer decides which types should have extents and which extents should be indexed.

  The value of a key is unique within an extent. Simple key corresponds to a single property, while a compound key is a set of properties.

- **Metadata.** The ODL Schema Repository stores persistent objects that define the schema of an ODMS. The ODMG metadata model describes the class assignments for this data — types like TypeDefinition, Property and Interface.

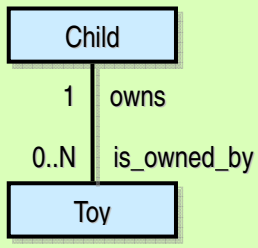
PROPERTIES AND OPERATIONS

- **Attributes.** The two kinds of properties defined by ODMG are attributes and relationships. Every attribute has a name and type and its value is either a literal or an OID. Attribute definitions in interfaces only define abstract behavior and a class can implement such interface either by a data member or by a pair of accessor methods.

- **Relationships** are always defined between two object types using two inverse traversal paths — either simple or collection ones. This allows the creation of 1:1, 1:N and M:N relationships like in the below example of a child owning toys. The ODMS is responsible for maintaining referential integrity of relationships, but not object-valued attributes. Relationship definitions in interfaces are purely procedural.

```
class Child { …
    relationship set<Toy> owns
        inverse Toy::is_owned_by;
}

class Toy { …
    relationship Child is_owned_by
        inverse Child::owns;
}
```



- **Operations** specify behavior of a single type using signatures with typed arguments and returned value, and any exceptions the operation may raise. ODMG does not define formal semantics of operations and it assumes their sequential execution. Operations may have side effects.

OBJECTS

- **Object Creation.** Objects are created by invoking the new() operation on an ObjectFactory interface. Every object must implement methods for copying, deleting, equality comparison and locking. All creation, access, modification, and deletion of persistent objects must be done within the scope of a transaction.

- **Object Identifiers** (OIDs) are automatically assigned to all objects, are unique within the scope of a given ODMS[1], and are immutable. They keep object identity through updates and connect objects in relationships.

- **Object Names** have the role of global OODB variables — any object can get a name for direct access to its data. Name definitions are independent of type interfaces. Named objects are identical with the notion of "root objects" or "entry points" in the database.

- **Object Lifetimes.** Objects can be either persistent or transient. *Transient objects* are managed by programming language runtime system and they only exist for the duration of a procedure, a thread or a process. *Persistent objects* are managed by ODMS runtime environment and they survive the termination of the process that created them. Object lifetime is orthogonal to its type — some instances of a type may be transient and others persistent.

This overview of the ODMG object model does not include the exception system, locking and concurrency control, transaction services and operations for managing the whole database.

3.2.3 State of the Art and Implementations

The ODMG standard started with version 1.0 in 1993, evolved in four releases to its final version 3.0 of 1999, and in 2001, the activity of ODMG was suspended. Some of its critics pointed out that the standard was designed by several object database vendors without regard to implementations on top of relational databases; others claim that it was not flexible enough (for example, it lacked support for multiple inheritance). ODMG also did not provide a test suite to verify implementation correctness, so differences in behavior of ODMG-compliant products became quite common. Not many OODB products came close to implementing the full specification, but some that strove to accomplish this were FastObjects, Poet, Jasmine, GemStone, Objectivity, and Versant.

3.2.4 Sources

The official site of the former ODMG activity is at http://www.odmg.org/. A book that contains the latest ODMG standard is [CB00], and an overview ODMG shortcomings and its comparison with JDO can be found at http://www.jdocentral.com/JDO_Commentary_DavidJordan_4.html.

3.3 Java Data Objects — JDO

Why Was it Chosen

JDO supersedes the ODMG standard in the Java world. It was shaped by a community process and it takes into account many recent advances in the field of IT (like three-tier applications, component architectures, or ubiquitous computing). It has gained wide acceptance and is being implemented and used in a wide range of solutions, including a number of object-oriented and object-relational databases.

[1] Object Data Management System

INTRODUCTION

The Java programming language has had two standard means of persisting objects — object serialization and JDBC (Java Database Connectivity). However, serialization is a simple mechanism that does not support transactions and queries, and JDBC is only useful with relational databases that support SQL.

In 1999, a task force was formed within the Java Community Process to specify a Java-centric model of persistent information. The Specification Request (JSR12) was approved in July 1999, official JDO 1.0 standard followed in March 2002, and the current release is JDO 1.0.1 from March 2003.

The two major objectives of the JDO architecture are:

- First, to provide application programmers with a transparent, Java-centric view of persistent information, including both enterprise and locally stored data
- Second, to allow pluggable implementations of datastores into application servers. Data from these datastores is presented in the Java Virtual Machine as instances of persistent classes.

JDO defines *interfaces* and *classes* for application programmers when using objects that are to be stored in persistent storage, and specifies the *contracts* between suppliers of persistence-capable (P-C) classes and the runtime environment (which is part of the JDO Implementation).

JDO also defines a *query language* JDOQL (JDO Query Language) that is similar to OQL, but simpler. It allows the execution of database queries in the form of simple function calls that specify the set of queried objects and a boolean condition (filter). These queries are internally translated to datastore query languages (like SQL). JDOQL is allowed to violate encapsulation and query private object state because no method calls (including calls to accessor functions) are allowed in queries.

JDO can run in a variety of environments ranging from three-tier application server architectures (EJB) and two-tier client-server environments to limited capability devices like cellular phones.

3.3.2 USING JDO

The JDO persistent object model of an application is determined by *a set of Java classes* and an *XML metadata file*. The XML metadata file contains modeling directives that either override the default semantics of the Java language or provide additional semantics not expressible in Java, thus augmenting it to get a full object DDL (data definition language).

Every P-C application class must be listed in the metadata file. Every P-C class must also be *enhanced* for processing by JDO – the most common approach is to use an *enhancer* that generates enhanced class files. One of the changes made to these classes is that they implement the PersistenceCapable interface. Most classes can be made persistent, except ones whose state depends on remote objects or is not accessible to Java (like classes that use Java Native Interface or most system-defined classes).

Every JDO datastore then provides the programmer with a PersistenceManagerFactory object that instantiates a PersistenceManager. This interface embodies the database connection, manages the object cache and acts as a factory for other JDO classes like Transaction and Query.

3.3.3 JDO DATA MODEL

- **Specifications and Implementations.** JDO runs on top of the Java framework of classes and interfaces. Only class state accessible to Java can be stored, class behavior (methods) is not subject to JDO persistence mechanisms.

- **Object types.** JDO supports the persistence of these types of fields: *Class types* include any P-C user-defined classes or interfaces. *Atomic types* correspond to Java ones, including Date, Locale and String. *Collection types* include arrays of values or references, plus standard Java collection interfaces and classes (maps, sets, lists, arrays, vectors, hashes, and trees). However, only HashMap is mandatory for all JDO implementations since some relational databases do not support richer schema semantics.

- **Inheritance.** In standard Java (single) inheritance hierarchies, any class can be made transient or persistent, but in the latter case, its immediate superclass must be explicitly mentioned in the XML metadata file. By default, all classes are transient.

- **Objects and Literals.** JDO has two types of persistent objects — *First Class Objects* (FCO) and *Second Class Objects* (SCO). Every FCO has a unique JDO identifier — it can be referenced by all other persistent instances. In contrast, a SCO has no identity and is either embedded in each of its "parent" objects at commit time, or is unowned and lost at commit time. SCO are defined in the metadata file for *fields* that reference objects, not for whole classes, which can result in some instances of a class being FCO and others SCO.

- **Extents.** Every P-C class can optionally have an extent that contains a collection of all its persistent instances. A class that uses the makePersistent method must have an extent.

- **Metadata.** The underlying Java object model supplies the programmer with a reflection mechanism — metamodel information and tools for its manipulation.

- **Attributes and Relationships.** In OODB terminology, FCO correspond to objects and SCO to literals. If the field references an FCO, a relationship is created, while for an SCO, the whole attribute is embedded. Arrays are always SCO and facilitate the creation of 1:N relationships.

  Treatment of null values depends on XML metadata information:

  None means that null fields are saved as null fields, unless the database has no support for them; in such case, an exception is thrown.

  Exception specifies that an exception be thrown each time a null field appears at transaction commit time.

  Default value of the attribute leaves the treatment of null fields to the database – they are converted to whatever default value the database uses for null values of the given type.

- **Operations.** JDO only works with the state of P-C class instances, not their behavior. The behavior has to be implemented in business classes that manipulate and query persistent JDO instances.

- **Object Creation and Lifetime.** Every object of a P-C class starts out transient but can be converted to persistent either explicitly (makePersistent method) or via *persistence-by-reachability*, which stores all P-C instances referenced by a persistent instance at the end of each transaction.

- **Object Identifers and Keys.** There are three types of object identifiers in JDO: application identity, datastore, and non-datastore. Class identity type is given in the metadata file.

  Application identity corresponds to the relational notion of primary key and is determined by the value of one or more class fields. For each class, a dual class that only contains the corresponding identity fields is automatically created.

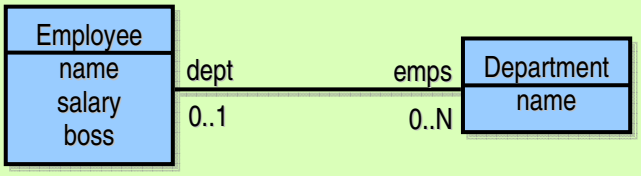
  Datastore identity uses a datastore-generated ID and is very similar to the OID in OODB terminology.

Non-datastore identity is managed by JDO itself for datastores that cannot save object identifiers, like log files or text files.

- **Class and Field Modifiers.** All of Java modifiers are supported: public, private, protected, static, transient, abstract, final, synchronized, and volatile. Access modifiers are enforced by converting the fields to private and adding accessor/mutator methods. Synchronized and volatile modifiers do not affect JDO, and although the static, final and transient fields are not persistent by default, this can be overridden in the metadata file.

3.3.4 Sample XML Metadata File

The following XML metadata file gives an example of definitions necessary for persisting two simple classes. Notice the tags for several vendor-specific model extensions.




```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
<package name="com.mycompany.myapp">
    <class name="Employee" identity-type="application" objectidclass="EmployeeKey">
        <field name="name" primary-key="true">
            <extension vendor-name="sunw" key="index" value="btree"/>
        </field>
        <field name="salary" default-fetch-group="true"/>
        <field name="dept">
            <extension vendor-name="sunw" key="inverse" value="emps"/>
        </field>
        <field name="boss"/>
    </class>
    <class name="Department" identity-type="application" objectidclass="DepartmentKey">
        <field name="name" primary-key="true"/>
        <field name="emps">
            <collection element-type="Employee">
                <extension vendor-name="sunw" key="element-inverse" value="dept"/>
            </collection>
        </field>
    </class>
</package>
</jdo>
```

3.3.5 State of the Art and Implementations

Sun Microsystems, Inc. provides reference implementations of JDO on top of JDBC and on top of the filesystem, and a reference class enhancer has also been developed. A publicly available suit of compatibility tests ensures that JDO implementations are correct even on the binary level.

The JDO standard was very well received and within months of its publication, both commercial and noncommercial JDO datastore implementations were available. The JDOCentral currently lists 28

existing implementations, both in object databases and object-relational bindings. Seven of these are open-source projects.

FUTURE EXTENSIONS TO JDO

The following list shows a selection of features that for upcoming releases of the JDO standard:

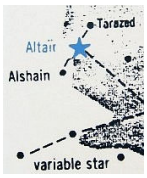
- Semantics for nested transactions, definition of savepoints with undo option within a transaction.
- Support for distributed objects and object relationships across datastores.
- Optional object-relational mapping information in XML metadata (table names, key information).
- Semantics for inverse (managed) relationships according to the EJB specification.
- Explicit support for java.sql.Blob and java.sql.Clob object types — binary/character large objects.

3.3.6 SOURCES

The full JDO specification is [Craig03], while a brief overview of its data model is given in [Jordan01]. A whole thesis on JDO is [vEchelpoel02], and a good Internet starting point is http://www.jdocentral.com/.

3.4 SET MODELS (O2)

WHY WAS IT CHOSEN



The set-theoretic paradigm is perhaps the most common for formal object-oriented data models. Although the first implementation of an OODB was GemStone [MOP86] in 1985, the O_2 model developed within the Altaïr project is described because it is also a representative of the first wave of object-oriented databases (near the end of the 1980s), it is strongly typed, and its structure reflects the OODB Manifesto.

INTRODUCTION

Toward the end of 1980s, following the rapid evolution of object-oriented languages (such as C++ [Stroustrup86] and Smalltalk [GR83]]), emerging object-oriented databases received much attention of the research community. Target applications for these systems were assumed to include office information systems (OIS), CAD/CAM systems and geographic information systems (GIS). Some of these early object databases included:

- **GemStone** [MOP86] which extended the Smalltalk language for database programming, data definition and queries. It was probably the first implementation of an object-oriented database.
- **Orion** [BKK87] used an extension of Lisp and allowed method programming using queries.
- **Ontos/Vbase** [Atwood85] provided the TDL language for data definition and extended C to COP (C Object Processor) for method definitions. Later came a query language similar to SQL.
- **Iris** [FBC87] had evolved from a functional to an object-oriented model, but still quite similar to the relational one. Its OSQL language (similar to SQL) allowed data definition, manipulation and queries. Complex tasks required implementing external functions in the C language.

- **Exodus** [CdWV88] used the Excess language, which had QUEL syntax and allowed database updates.

- **O2** [LRV92] used the O_2 language for data definition in context of other languages (C, Lisp, Basic). It provided own IQL and Reloop query languages.

From these, O_2 was chosen to show a conservative data model based on set theory. It was also the first successful effort to build a language-independent object database.

3.4.2 O2 DATA MODEL

- **Interfaces and Implementations.** Schema definition with method signatures is distinct from method implementations in the programming languages supported through special bindings.

- **Subtyping and Inheritance.** O_2 supports multiple inheritance, with Object implicitly at the top of the hierarchy. A subtype must contain every attribute of its supertype plus any new or refined ones. Late binding and overloading of methods are supported.

- **Object Types and Literals.** All objects, values and methods are strongly typed, although O_2 semantics allow tuples with extra attributes. Literals without an OID are called *complex values*, and they are composed using the set, tuple and list constructors. An object is a literal with an OID. Complex values belong in *types*, while objects belong in *classes* (composed of a type definition and methods). O_2 supports these atomic types: integer, float, string, char, Boolean and bits.

```
add class Monument with extension inherits Place
    type  tuple  (    name : string,
                      address : tuple (areacode : integer, city : City),
                      closing_days : list (string) )
```

- **Extents.** If a class is defined with an *extension*, the system creates a named set-value that contains all the objects of that class (and, therefore, makes them persistent).

- **Metadata.** Database schema is separate from database instances (or contents); sets of attribute names, object identifiers and class names are all pairwise disjoint and the programmer has no access to the metamodel.

  For schema evolution, a tool called *Interactive Consistency Checker* detects whether any inconsistencies (structural or behavioral) would occur after a proposed schema update. Changes include adding/modifying/dropping of classes/attribute/method names and attributes.

- **Attributes and Relationships.** The set/list elements and tuple elements can be either values or OIDs that build relationships between objects. 1:N relationships are constructed with sets or lists of OIDs. There is no explicit support for special kinds of relationships such as inverse ones.

```
add name Eiffel_Tower : Monument
Eiffel_tower = tuple (    name : “Eiffel Tower”,
                          address : tuple (areacode : 18341 city : Paris )
                          closing_days : list (“Sunday”, “Monday”, “Saturday”) )
```

- **Operations.** Methods can be attached to classes or to individual objects and they can be private or public. They can be programmed using CO_2, BasicO_2, Lisp O_2 or the IQL query language. These languages are extended by O_2 to gain object-oriented features, iterators and remote method

invocation. Method inheritance is possible under several conditions. The programmer also has full runtime control of distribution, i.e. whether the method runs on server or client.

A simple method signature and implementation for increasing the fee of Monument instances is shown below (in two different languages). The last line demonstrates method invocation including a distribution directive.

```
add method increase_fee (amount : integer) in class Monument
body increase_fee(amount : integer) in class Monument
     co2{ *self.admission_fee+= amount; }
     lispo2   (setq self.admission_fee (+ (get-field self. 'admission_fee) amount) )
[on server Eiffel_Tower increase_fee(3)]
```

- **Object Identifiers, Names and Lifetimes.** Both objects and values can be named. Persistence is based on names, so that all named objects and values are persistent, including their member objects and values. Objects are not explicitly deleted — they have *garbage-collection* semantics and disappear by becoming unreachable from named persistence roots.

3.4.3 Example of Formal Semantics

For a sample of the set-theoretical semantics of O_2 data model, the formal definition of types and classes is given below:

Let Bnames be the set of names for basic types, including nil and any; Cnames is a countably infinite set of constructed type names (disjoint with Bnames) and Tnames is their union. Meth is a set of methods.
A type is either basic or constructed.
A basic type is a pair (n,M) where $n \in$ Bnames and $M \in$ Meth.
A constructed type is one of the following:

1. A triple (s,t,M) where $s \in$ Cnames, $t \in$ Tnames, $M \in$ Meth. We denote such type by s = (t,M).
2. A triple (s,t,M) where $s \in$ Cnames, t is a finite partial function from attribute names to Tnames and $M \in$ Meth. We denote such type by $s = ([a_1:s_1,\ldots, a_n:s_n], M)$ where $t(a_k) = s_k$ and call it a tuple-structured type.
3. A triple (s,s',M) where $s \in$ Cnames, $s' \in$ Tnames and $M \in$ Meth. We denote such type by $s = (\{s'\}, M)$ and call it a set-structured type.

3.4.4 State of the Art and Implementations

More than five years (1986–1991) of research and implementation effort was invested in O_2 within the Altaïr project, first resulting in experimental prototypes and later in academic and commercial releases of usable systems. The whole project was backed by several Esprit, INRIA and CNRS grants, university of Paris-Süd, Siemens and Bull. In the following years, it was merged with other object-oriented systems.

Aside from the database server itself (consisting of Schema Manager, Object Manager and Disk Manager), O_2 also contained two ad-hoc query languages (Reloop and later IQL), a user interface server and generation tool (LOOKS), a programming environment (OOPE), persistent extensions of several programming languages, and other data management components (alphanumeric interface, modules for buffering, indexing, clustering, TCP/IP communication…).

Comparison with the most recent ODMG specification shows a number of corresponding concepts, including language independence, similar data definition languages, modeling concepts, typing, signatures, and more.

3.4.5 Sources

[LRV92] gives a formal definition of the data model. It is part of a whole book on O2 design and implementation. Other sources for object-oriented databases from around 1985–1990 include [MOP86], [BKK87], [Atwood85], [FBC87], and [CdWV88].

3.5 Concept Definition Language — CDL

Why Was it Chosen

The object model developed in this thesis is a subset of CDL (Concept Definition Language). Information about CDL development, design choices and implementation was readily available for research, because it was developed at the Faculty of Electrical Engineering of the Brno University of Technology. Its object model contains certain unique features in the areas of runtime behavior, namely dynamic inheritance using roles.

Introduction

Concept Definition Language started as an effort to implement the ODMG–93 specification for an object-oriented database. However, the ODMG has some limits on full use of different object-oriented concepts such as multiple and dynamic inheritance. Branching off from the ODMG standard let the new data definition language focus on some of these syntactic and semantic areas:

- CDL respects *common data definition patterns* — e.g. collections are defined in a natural way together with their respective member classes and can be easily extended by additional attributes.
- CDL tries to *simplify decisions* for the database designer — e.g. only a single collection type is used with an option to add ordering, duplicate instances and indices. Flexibility in multiple inheritance with abstract classes removes the need for interfaces.
- CDL provides a *uniform* approach to the whole model — e.g. enumeration types are created by means of simple structures with named instances, and relationships do not need to be differentiated from properties.

On the other hand, CDL only defines the object model and object specification language. It does not provide a query language, and guidelines for integration with programming languages are not part of CDL itself.

3.5.2 CDL Data Model

Types

- **Specifications and Implementations.** The CDL language only provides *interface specification* of concepts and modules. The choice of an actual programming language is independent of CDL itself, but a binding to Visual Basic was implemented.
- **Object Types.** The CDL system is strongly typed. The two foundational metatypes are *collections* and *structures* (tuples). All collections are ordered multisets with items of the same type (values, objects or collections) plus any additional properties — often indices or user-defined aggregate data, while ODMG has multiple collection types without additional properties. Structure types

may be either simple *literals*, or *objects* with an OID. For simplicity and with regard to common usage, at most one collection type can be associated with each structure type.

All objects of the class below have an OID, so their NextTo attribute is a collection that contains references to other houses. The HasRooms attribute may represent an aggregated collection of embedded objects without an OID, depending on its definition:

```
Concept House/Houses  [Data = Ref]
    Properties
        HasRooms : Rooms
        NextTo : Houses
End Concept
```

- **Subtyping and Inheritance.** CDL supports full multiple inheritance of both value and object types. In subtypes, properties may be added, modified and even removed. In place of interfaces, CDL uses *abstract classes* without an implementation that can be used to specify contracts for other classes (but collection types of abstract classes are not abstract to enable collections of subclass instances).

  The concept of *roles* allows objects to dynamically inherit additional types at runtime. For this purpose and to enable stronger type checking, subtypes are defined to be either *supplementary* or *alternative*.

For the Person class, subtypes Man and Woman are alternative — an object of one of these types may not gain the other type. On the contrary, subtypes like Student, Parent or Employee are supplementary because an object of type Person may gain or lose one or more of these types during its lifetime in the database.

- **Literals.** A literal has no OID and has to be embedded in an object. All collections and simple types (like numbers) are literals, while structures may be defined as either literals or objects.

- **Extents and keys.** For each object type, the designer may order the system to keep an extent. In that case, the associated collection type is used for storing the extent. An extent also contains all instances of subtype objects. Every persistent object in the system needs to in at least one extent.

- **Keys.** Every property can become a key (MatchCode). The system makes sure that all values of the key within its extents are unique. Another modifier may specify that the property is required (Mandat).

- **Metadata.** Information about types is contained in the Database object, which is stored along with all other data. All extents are also embedded in this object, along with other properties that describe the database.

The Car concept has a corresponding Cars collection with an AveragePrice attribute and an index of car types. These types are also used for object labels:

```
Concept Car/Cars   [Extent]
Properties
    Type : String [Matchcode, Label]
    Price : Integer [Where = Alone, Mandat]
    AveragePrice : Integer [Where = Col]
    IndexByType [Index = "Type"]
End Concept
```

PROPERTIES AND OPERATIONS

- **Attributes.** All attributes are named and strongly typed. They can be implemented algorithmically or as data members, and this can change in subtypes (ODMG does not allow algorithm implementations for properties of literals). An attribute can be a structure, a collection or an object (which creates a relationship), and it may be parameterized by a named typed parameter.

- **Relationships** are attributes whose type is an object (1:1) or a collection of objects (1:N). Additional information can include the name of an *inverse* relationship from the target (in ODMG, only these two-way links are called "relationships"); relationships can also be *subordinate*, which means that all targets are deleted upon deletion of the master object.

- **Constraints.** The *range* modifier can limit the value of a property. Range can be given by an enumeration of legal values, a numeric interval, and for collections (such as in 1:N relationships), by the set of objects contained in another collection.

- **Operations** include procedures and functions. They are similar to their ODMG counterparts.

In this concept, it is likely that TitledName will be a computed attribute without actual physical storage. HighestTitle must be one of employee's titles, AverageWage is different for each year and SendMessage is a function (probably with side-effects):

```
Concept Employee/Employees
    Properties
        Name : String [Kind = SetGet]
        Title : Titles [Kind = SetGet]
        HighestTitle : Title [range = ".Titles"]
        TitledName : String [Kind = Get]
        AverageWage (Year : Integer) : Float
        SendMessage (text : String) : Boolean [Kind = Function]
End Concept
```

OBJECTS

- **Object Identifiers** are unique within the database system, immutable and system-assigned. Since OIDs are not human-readable, each structure definition can appoint one of its properties to serve as a label for its instances in the graphical browser.

- **Object Names** can be defined for any structure type, even for values. This is done within the type definition; its extent must always contain these named objects. An empty structure with named values can represent an enumeration type.

In an enumeration type, only the names of instances are needed:

```
Concept KindsOfSteak  [Data = Enum]
    Instances
        Raw
        Medium
        WellDone
End Concept
```

3.5.3 Example of Formal Semantics

CDL has been partially formalized in [Güttner03] recently and its model closely resembles the O2 one, although it has certain categorical extensions. For this reason, the following example is only a brief excerpt from the model (and obviously builds on several previous definitions):

> A *database schema* is a tuple (S, L, D, OID, A) satisfying these conditions:
>
> D is the set of elementary domains, OID are OID sets, A are attribute names, L is the set of type labels, S(D, OID, A) is a function that assigns types.
>
> $\forall t \in T$: kind-of(t)= elem $\Rightarrow$ $t \in$ cod(S).
> All elementary domains are part of the schema.
> $\forall o \in$ OID $\exists! t \in$ cod(S): kind-of(t)= ref $\wedge$ t = (o,x).
> Fr every set of OID references, there is a corresponding structured object type.
> $\forall t \in$ cod(S): kind-of(t)= data $\Rightarrow$ is-def-in(t,S$\cup$OID).
> No data type is defined using some unknown type.
> $\forall t \in$ cod(S): kind-of(t)$\in$ {ref,col} $\wedge$ t=(u,v) $\Rightarrow$ is-def-in(v,S$\cup$OID).
> No object or collection type includes an unknown type in its definition.
> $\forall t \in$ cod(S): kind-of(t)=col $\Rightarrow$ $\exists! u \in$ cod(S): t=(Nu,v).
> For every collection type, there is a basic type for objects that it contains.
> $\forall t \in$ S: $\neg$(t is-part-of t).
> The is-part-of function helps indicate infinite levels of embedded data within objects.

A full RDF-compatible formal description of an object model closely resembling CDL is in chapter 6.

3.5.4 State of the Art and Implementations

CDL was implemented by Vema co. in its G2 object database several years ago. The implementation was deployed in a number of commercial applications. It was based on a high-level interpreted language (Microsoft Visual Basic), which enabled features such as dynamic inheritance and algorithm storage in the database. Another advantage was natural support for complex collection types, while some of the disadvantages involved performance issues. After several years, the G2 database development was discontinued.

3.5.5 Sources

[HM00] contains an introduction to CDL and [Güttner03] formalizes the core features of the language. The company that implemented G2 can be found at http://www.vema.cz/.

3.6 Categorical models

Why Was it Chosen

Category theory (CT) is a very general formalism with big potential for modeling complex systems in a declarative way. It is also radically different from the usual set- and logic-based approaches. Moreover, recent advances in graphical modeling (such as UML [OMG03]) are likely to find formal expression in CT's fully graphical representation. Several object-oriented data models based on category theory were proposed and attempts at their implementation exist.

INTRODUCTION

A category C is defined as a collection of *objects* obj(C), *arrows* arr(C) and a *composition function* comp: arr(C) × arr(C) → arr(C). Every arrow f∈ arr(C) has a source src(f)∈ obj(C) and a target tgt(f) ∈ obj(C) (arrows are written as f: src(f) →tgt(f)). Four additional conditions must hold:

> Composition of arrows:
> ∀ f,g∈ arr(C): tgt(f)=src(h) ⇒ ∃ h∈ arr(C): src(h)=src(f) ∧ tgt(h)=tgt(g) ∧ h=comp(f,g)
> Associativity of composition:
> ∀ f,g,h∈ arr(C): comp(f,comp(g,h)) = comp(comp(f,g),h) if one of the sides is defined
> Existence of identity arrow:
> ∀ x∈ obj(C) ∃ id_x∈ arr(C): src(id_x)=tgt(id_x)=x
> Function of identity arrow:
> ∀ f∈ arr(C): comp($id_{src(f)}$,f)=comp(f,$id_{tgt(f)}$)=f

Special types of arrows and objects are defined depending on the structure of a category. These notions are universal and can be applied to any category, no matter what the arrows and objects represent. For an example of categorical reasoning, see the subsection about formal semantics.

The manifests of Diskin and Goguen ([DC94] and [Goguen91]) indicate several advantages for using category theory for formal specification of object-oriented data models:

- Category Theory (CT) offers a *graphical*, yet *formal* language for modeling objects. Generic reasoning about any one object simplifies abstraction and proofs.
- CT provides *uniform constructs* that help discover interesting generalizations and see complex properties that would otherwise go unnoticed.
- CT uses a *declarative approach* that specifies objects only in terms of their relationships with other objects. Unlike in set or logic theories, the internal structure of objects does not need to be given.

3.6.2 LDM DATA MODEL

In this section, categorical LDM (Limit Data Model) of [Kolenčík98] is briefly compared to the ODMG model. Only a limited part of a full object model has been formalized, since the principal focus of LDM was to give a rigorous categorical background to several basic modeling concepts.

- **Types.** LDM is strongly typed — all of its objects have at least one type. Some types can be referenced by relationships (classes) while others need to be embedded in other objects (literals). All objects behave like tuples, because they are aggregated from simpler types and references.
- **Subtyping and Inheritance.** LDM models true multiple inheritance and even dynamic inheritance using roles. In this context, it defines the semantics of attribute access, virtual classes, polymorphism, and method dispatching.
- **Attributes and Relationships.** The semantics of attributes are identical to the ODMG standard. Relationships are always unidirectional, and can be either 1:1 or 1:N; in the latter case, a set of OIDs is modeled by power objects. LDM also ensures referential integrity.
- **Operations** are typed and their semantics is expressed by means of category theory using exponential objects. Method side effects are not part of the model.
- **Data manipulation.** Since the contents of a database are modeled by a functor from an intension category to an extension one, a transaction that manipulates data is a redefinition of the functor.

- **Object Identifiers** are strictly logical. They are represented by elements in the category of sets — each element is unique, and it can be connected to other elements by appropriate mappings.

- **Object Lifetimes.** All objects in the model are considered persistent.

- **Queries.** The LDM model formalizes a large part of OQL query language of ODMG–93, focusing on the select–from–where clause.

3.6.3 Example of Formal Semantics

Some approaches to formalizing the persistent object-oriented model using CT are given below:

- **Product model** of Nelson and Rossiter [NR95] focuses on queries, closure and views (using subcategories and natural transformations), but it also formalizes keys, relationships and aggregation. Each class is modeled as a separate category with methods and dependencies represented by arrows whose source and target represents persistent and in-memory variables. Keys correspond to initial objects, relationships to pullback categories, generalization to a coproduct.

- **Typegraph model** of Tuijn [Tuijn94] generates the database schema category from a graph (called *typegraph*). Instances are given by functors from this category to another one, not necessarily a category of sets and mappings. Tuijn's model is structurally equivalent to IQL with the query language defined only by universal constructions. Attributes, relationships and generalization are modeled as arrows.

- **Theory-based approach** of Fiadeiro et al. [FSM90] focuses on building an information system from a formal logical specification (a theory presentation) that describes properties and behavior of objects. Aggregation, inheritance and relationships are expressed by constructs on a diagram in the category of theory presentations.

- **Process-based approach** of Costa et al. [CSS94] is very similar to the previous one, but theory presentations are substituted by process models. This shows the unifying power of CT because instead of the usual representation of objects by sets and their structure by mappings, every object is represented by a Marcovian process and arrows denote partial mappings between these processes.

- **Limit Data Model (LDM)** of Kolenčík [Kolenčík98] works with a category whose objects represent classes, and arrows model attributes, methods, inheritance and relationships. Limits such as pullback or pushout are used to define additional constraints on the schema, such as virtual base classes and polymorphic behavior.

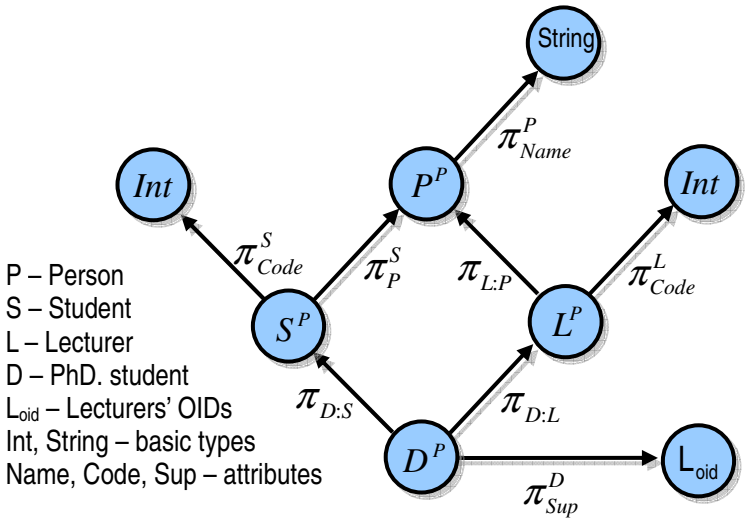


Figure 3.4 Example of an LDM schema category

Figure 3.4 shows a category that defines the schema of a simple database. Each arrow represents a projection of a product object. Such projections can represent simple string-valued attributes (person's name), OID-valued relationships (PhD. student's superior), or subclassing relationships (lecturer is a subclass of person). Since the graphical schema is a category, arrow compositions must be defined — these correspond to dotted naming conventions. The superscript in object names shows that the classes are "pure" — they do not contain objects of their subclasses, and specific arrows connect them to the respective "full" classes. Additionally, for Person to function as a virtual class, it is required that the D–S–P–L diamond is a *pullback*.

3.6.4 State of the Art and Implementations

Most categorical models focus on formal proofs for certain OODB concepts. Different authors use different categorical constructs for their models, without a standard way to express them. Most works only give a partial formalization of the data model. All of these factors contribute to the fact that implementations based on CT are scarce.

However, the Vision database ([FactSet00]) is based on categorical foundations — instead of storing objects as records, it works only with featureless sets and mappings. All properties and methods of an object are given through its mapping onto other objects, and database concepts such as memberships and keys are modeled by constraining these mappings (e.g. by injectivity). Families of appropriately constrained maps represent inheritance and polymorphism, and map transformations are powerful enough to model joins, projections, updates etc. to the point that they completely define a database query language.

3.6.5 Sources

The LDM model is presented in the PhD. thesis [Kolenčík98]. Manifests for categorical modeling of object-oriented concepts are [DC94] and [Goguen91]. Other models include [CSS94], [FSM90], [Tuijn94], and [NR95].

3.7 LOGIC MODELS (F–LOGIC)

WHY WAS IT CHOSEN

This section is an important example of modeling in logic. The new RDF-based OODB model proposed in this thesis is a logical one, and therefore this section is an introduction to some of its concepts. Moreover, many deductive object-oriented systems are currently being extended to handle Semantic Web data and provide entailment engines.

INTRODUCTION

Most of the previous work was focused on imperative and functional languages, but good reasons exist for bringing the object database paradigm into the world of *logic programming* [Kifer95]. The object-oriented paradigm provides a good way of specifying and manipulating structured objects, while logic and deduction offer the power of ad hoc reasoning and querying. This is increasingly important with growing database size, especially in the Semantic Web.

Some attempts have tried to integrate Prolog with object-oriented languages (C++), others define object-oriented features in existing logics. F-logic [KLW95] is an object-oriented logic designed for use in object-oriented logical (deductive) databases. Its main limitation is that it can only define and query object state, not change it through method side effects. For expressing the semantics of database updates, Transaction logic was designed by the same authors ([BK95]). F-logic has a sound and complete proof theory.

3.7.2 F-LOGIC DATA MODEL

- **Specifications and Implementations.** In F-logic, all types are classes — there are no equivalents of ODMG interfaces or literals. The semantics of F-logic do not address the question of separating interfaces from implementations, but if desired, signatures may be specified separately for practical encapsulation and reuse issues, although both use the same logic language[1].

- **Object Types.** Every object has a type and every operation requires typed operands. Type correctness is ensured by F-logic inference and consistency rules. Atomic, collection and structured objects are not distinguished — all objects are structured. The only collection tools of F-logic are set-valued object attributes. There is no direct equivalent of ordered sets, bags or collection types.

Schema information: The definition of class dept defines a *property* called name of elementary type string; *relationship* manager connects to an employee object; a *set relationship* of assistants (who are both students and employees); a *method* that takes a department and returns the number of its assistants; and a *class attribute* (label).

```
dept [dept_name ⇒ string;
      manager ⇒ empl;
      assistants ⇒⇒ (student, empl);
      assist_count @ dept ⇒ integer;
      class_label → "Department" ]
```

[1] In ODMG, several bindings are described for different implementation languages, but due to their high complexity, their semantics are not part of the data model

- **Subtyping and Inheritance.** F-logic groups semantically related objects in IS-A class hierarchies that support multiple inheritance. Any data expression in a class can be either inheritable or non-inheritable — an important distinction for class attributes. Semantically, inheritance in F-logic is set inclusion.

  Object myCar is an *instance* of class peugeot, which is a *subclass* of frenchCar — an instance of class carType (see the paragraph on metadata below). The last line is a query that retrieves all classes that peugeot belongs to; observe that carType is one of the types it returns.

```
myCar : peugeot
peugeot :: frenchCar
frenchCar : carType
?- peugeot : X
```

- **Literals.** While in many object-oriented models a distinction is made between entities with an OID (objects) and without one (literals), this is not the case in F-logic. Here, all elementary values are objects — "Mr. Wong" and 1772 are actually OIDs of a string and integer object, respectively.

  Database facts: This definition of the tom object specifies his attributes and an affiliation relationship along with his department's name, manager and a set of assistants:

```
tom [ name → "Tom";
      age → 40;
      affiliation → fit [   dept_name → "IT";
                            manager → tom;
                            assistants →» {john, mary} ] ]
```

- **Extents.** Extents as such do not exist in F-logic since its model theory requires that all data atoms are accessible to the query mechanism, and to preview an extent, a simple query returns all objects of a given type. From this point of view, maintaining extents is only a performance issue.

  Queries: This query returns a set of employees who work at the "IT" department along with their names and ages.

```
?– X : empl ∧ X [ name → Y; age → Z : midaged; affiliation → D [ dname → "IT" ] ]
```

- **Metadata.** F-logic uses higher-order syntax by eliminating the hard syntactic distinction between objects, classes and attribute names. This is desirable because the same language can be used for defining, manipulating and reasoning about classes and their instances; for defining virtual classes using deductive rules; and for schema exploration and browsing. A class can be an instance of another class, and queries may return sets of classes or attributes. This does not lead to computational problems because the semantics remain first-order (higher-order variables range over intensional representations of sets rather than the sets themselves).

- **Attributes, relationships and operations.** In F-logic, complex objects are defined by specifying logic functions from a set of all objects. These functions are partial since they are only defined for objects of a given type. *Operations* of an object take a given number of typed arguments and return a typed result. *Attributes* are simply functions that take the context object and return a constant value. Since everything is an object in F-logic, there is no difference between an attribute and a 1:1 *relationship*. Set-valued attributes and 1:N relationships are functions that return sets of objects.

  F-logic has no notion of inverse or subordination relationships, exceptions or operation side effects (since it has no update semantics). It is clear that unlike the ODMG specification, F-logic has exact method semantics. Another important point is that some facts are saved in the database (extensional ones), while others are deduced from database contents at runtime (intensional facts).

Deductive rules: The following rule says that an employee's boss is automatically derived from the manager property of his department.
E [boss → M] ← E : empl ∧ D: dept ∧ E [affiliation → D [manager → M :empl]]

- **Object Identifiers and Names** are considered "syntactic gadgets" needed to refer to objects in the physical representation of a database or in a programming language, respectively. F-logic uses logical OIDs in the form of tree-like first-order terms.

Some examples of F-logic OIDs are mark, 17, "A duck" and father(adam).

3.7.3 Example of Formal Semantics

Semantic structures in F-logic are called *F-structures*, and they are represented by tuples.

An F-structure is a tuple $I = (U, \in_U, <_U, I_F, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\rightarrow}, I_{\bullet\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rightarrow\Rightarrow})$, where
- U is the domain (a set of all actual objects) of a possible world I
- I_F is a mapping from syntactic object IDs in F, the F-language, onto actual objects in U
- $<_U$ is a semantic counterpart of the subclass relationship between class objects
- $\in_U$ models class membership; it need not be acyclic and even $s \in_U s$ is possible
- $I_{\rightarrow}: U \rightarrow \Pi_{k=0}^{\infty} \text{Partial}(U^{k+1}, U)$ where Partial denotes all partial functions, maps IDs to methods of all possible arities, defining both methods, attributes and relationships.
- $I_{\twoheadrightarrow}$, $I_{\bullet\rightarrow}$ and $I_{\bullet\twoheadrightarrow}$ have similar semantics for set-valued and inheritable attributes
- $I_{\Rightarrow}$ and $I_{\Rightarrow\Rightarrow}$ attach types to methods (and attributes) and take subclassing into account

Given a statement, called an F-molecule, with a corresponding variable assignment, rules are defined for its *satisfaction* by an F-structure that makes the statement true. This mechanism serves to define and verify facts about the F-structure data model environment that has been created. Properties of F-structures can be stated using axiomatic F-molecules: reflexivity of the subclassing relationship is expressed by p :: p is satisfied by I.

The *proof theory* for the satisfaction (or logical entailment) relation is sound and complete. It starts with the definition of substitutions and unifiers and proceeds to define a suite of 12 inference rules. In addition to common ones for logic systems — resolution, factoring and paramodulation — additional rules are defined to capture the semantics of an object-oriented system; these include behavior of the subclassing IS-A relationship, type inheritance, restriction of method input types and relaxation of output types, restriction of return values for scalar methods, merging and elimination of simple id-term tautologies.

The subclass inclusion rule states that an object of a class is a member of its superclasses:

$$\frac{(P : Q) \vee C,\ (Q' :: R) \vee C',\ \theta = \text{mgu}(Q, Q')}{\theta((P : R) \vee C \vee C')}$$

where mgu denotes the F-logic version of most general unifier for a pair of clauses C and C', and P, Q, Q', R stand for id-terms.

3.7.4 State of the Art and Implementations

One example of a system that is based on F-logic is Flora [YK00]. It represents a second generation of deductive and object-oriented database systems (DOOD) — the first one attracted attention in early 1990's but never gained wide acceptance due to performance problems and other difficulties. However, renewed interest was sparked by the interest in autonomous agents, Semantic Web, RDF and ontology languages.

Flora aims at being a practical system with high expressive power, strong theoretical foundations, competitive performance, and a convenient development environment. It is based on F-logic (for object-oriented model), HiLog [CKW93] (for higher-order programming) and Transaction Logic [BK95] (for implementing updates). Rather than implementing its own deductive engine, it translates the program at source level into predicate calculus and uses XSB [SSW94], which has the advantage of very high performance and optimizations such as tabling, trie-based indices and unification factoring.

Flora has been successfully used to implement a number of sophisticated Web-based information systems, including a model-based mediation system, a webbase, and a knowledge-based integration system. The current FLORA-2 implementation is being developed at http://flora.sourceforge.net.

3.7.5 SOURCES

A keynote on integrating object-oriented and logical languages is [Kifer95]. F-logic is defined in [KLW95], Transaction logic in [BK95], HiLog in [CKW93], and the implementation of Flora was presented in [YK00].

4 OVERVIEW OF THE SEMANTIC WEB

In 1989, Tim Berners-Lee wrote a proposal for an information management system at the CERN laboratory [BernersLee89]. This idea, which he called the *Mesh*, later became the World Wide Web. At its core was a set of graphs with (textual) objects connected by relationships that would serve as a central CERN library, help system, document repository, and reference manual. Depending on the type of graph, objects and relationships would represent different entities — people and subordination relationships, projects and dependence relationships, documents and reference relationships. This would enable automatic processing and reasoning with those graphs (see Figure 4.1).

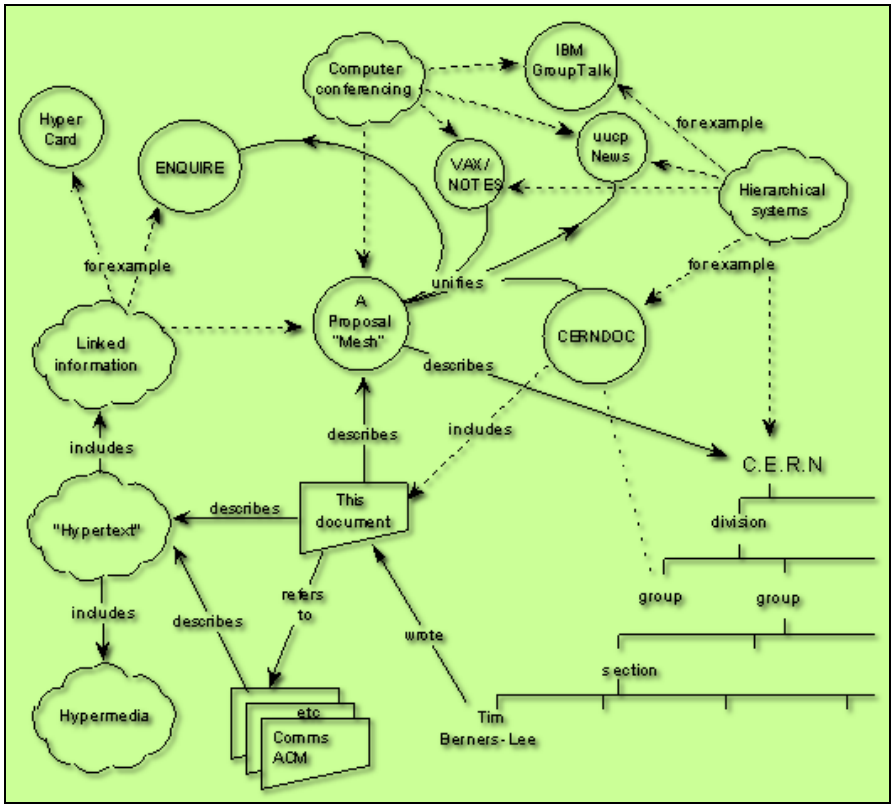


Figure 4.1 An example from Tim Berners-Lee's original WWW proposal [BernersLee89]

The evolution of the World Wide Web retained the principle of documents connected by references, but abandoned the idea of typing and classifying nodes and edges. The result is a web readable for

humans, but without formal semantics — an information jungle that is hard to search, syndicate, organize, and machine-process. Even the area of Web Services struggles to express precise meaning[1].

> For a computer program, it is hard to tell that a HTML, or even an XML document is someone's CV — in order to recognize it, text mining techniques dependent on specific vocabulary and language have to be employed. Web pages also lack formal machine-readable meaning, so it is almost impossible to automatically find out which of the numbers contained in the CV is the author's date of birth, and which of the many links leads to the company where the person works, as opposed to links leading to their favorite e-shops.

SEMANTIC WEB

The Semantic Web is a W3C (World Wide Web Consortium, http://www.w3.org/) activity that aims at extending the current World Wide Web with information that has well-defined meaning for both humans and computers. This would create an environment where *"software agents roaming from page to page can readily carry out sophisticated tasks for users"* [BHL00]. The means to this end are in:

- Presenting information in a structured and unambiguous way in contrast to today's Web pages that only define the form of their content, not its meaning.
- Providing automatic inference abilities for software agents in order to enable intelligent reasoning, browsing, searching, and interconnecting the data.

> A software agent was asked to set up an appointment with the doctor. Today the clinic's Web page might just contain keywords such as "treatment, medicine, physical, therapy", but the Semantic Web makes it possible to find out that Dr. Hartman works at this clinic on Mondays, Wednesdays and Fridays and that the appointment script takes a date range in yyyy-mm-dd format and returns appointment times. These semantics were encoded into the Web page using off-the-shelf software along with standard terms listed on the Physical Therapy Association's site. The agent "knows" that some of the terms are used by its calendar program so it sets up the appointment automatically. (Adapted from [BHL00])

At the core of the Semantic Web lies the RDF — *Resource Description Framework* that stores information in graphs where each edge represents a binary predicate. For uniqueness across the whole Web, both edges and nodes are labeled with URI references, or *urirefs* [BFM98]. RDF and RDFS (RDF Schema) also provide other features — see the following sections for more details. Semantics of new urirefs can be formally specified using existing sets of references, or model theory and RDF closures.

SEMANTIC WEB RESEARCH

Most of the current Semantic Web research activities focus on the following areas:

- Machine inference that selects relevant information, finds associations between facts, alerts to inconsistencies and uses rules to automatically draw conclusions — [PHH04], [Hayes04].
- Web document annotation, intelligent searching and cataloguing — [DC03], [Karger04].

---

[1] see WWW'03 conference keynote (SemWeb and Web services), http://www.w3.org/2003/Talks/0521-www-keynote-tbl/

- Integration and communication of heterogeneous systems using ontologies — [Rohner04].
- Creating an environment for intelligent autonomous agents — [CvHH01], [FIPA02].

4.2 Model–Theoretic RDF Semantics

RDF (Resource Description Framework) is the foundational element of the Semantic Web. It provides a way of describing and connecting resources on the Web in graph form. On one hand, it is easy to understand, yet on the other, it has precise semantics that allow the possibility of automatic entailment between these graphs. The following explanation briefs the official W3C specifications, namely [Hayes04].

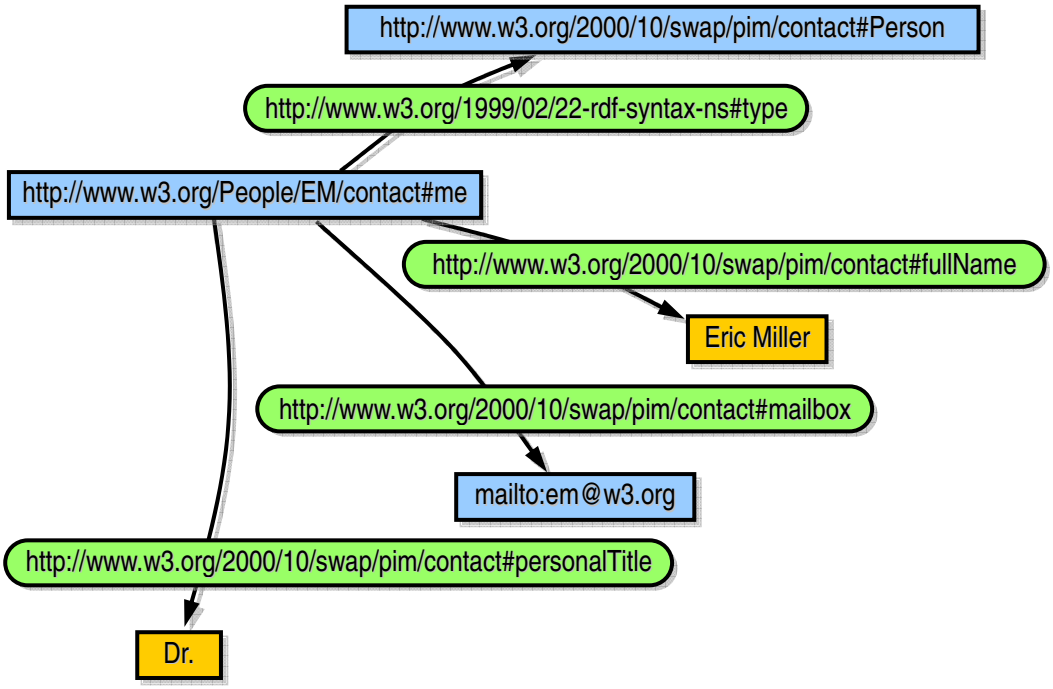


Figure 4.2 An example of a RDF graph (description of Eric Miller) [MM04]

4.2.1 Basic RDF Concepts

RDF semantics is built on top of *model theory* based on stating facts (*statements*) about the world (or *domain of discourse*). In order to model what these statements mean, they need to be *interpreted* into the world. For every set of syntactic statements, there generally exists more than one *interpretation* that assigns a set of *possible worlds* to that set of statements. It is basically impossible to say everything about a world, describing it so there is no doubt about every possible detail within it; therefore, every elementary statement says something about the world, limiting the set of possible worlds that satisfy the model. To impose structure on the world we are describing, it is necessary to state enough facts to limit the set of possible worlds to those that satisfy our requirements.

In case of RDF, an elementary statement is a *triple* subject-predicate-object, where each of the three parts is assigned an identifier — a URI reference (for example http://www.w3.org/1999/02/22-rdf-syntax-ns#type).

> A statement that gives an e-mail address for a person might look like this:
> http://www.fit.vutbr.cz/people/guttner
> http://www.w3.org/2000/10/swap/pim/contact#mailbox
> mailto:guttner@fit.vutbr.cz .
> The notation used here is called N-Triples. The URI for the subject denotes a person here — not a document reachable through the http protocol. Name of the predicate is taken from a working W3C standard, and the e-mail address is standard mailto: URI reference.

Such statement can be modeled by an edge of a directed graph where a predicate labels the edge from a node denoted by the subject to the node denoted by the object. A set of statements forms a RDF graph. A set of possible worlds for such graph can be found by *interpretation*. Interpretation can either be a simple assignment of URI references to things in the domain of discourse, or, by defining additional semantic constraints on some of the URI labels, it can take into account some specific requirements. A set of urirefs that influences the semantics of a given interpretation is usually called a *dictionary*.

RDF also allows two extensions to the above-mentioned triple structure:

- Both the subject and the object of a triple can be nodes that do not have an associated URI reference — *blank nodes*. Such nodes do not have any special meaning and the only requirement is that there exists an element in the domain of discourse that they can be mapped to (so all the statements in which they appear are consistent). In other words, blank nodes have the role of RDF existential quantifiers.

- An object of a triple can also be a *literal*. This can either be an untyped string that denotes itself in any interpretation, or a typed literal — one example is "13"^^xsd:Integer. Typed literals are used in places where the meaning of a value can be unambiguously specified by a string ("13") and a uriref (xsd:Integer). The semantics of these datatype urirefs is given by means external to RDF[1].

4.2.2 A Formal Description

Description of RDF semantics [Hayes04] states that for a set of URI references (a vocabulary) V that includes the RDF and RDFS vocabularies, and for a datatype theory T, a *T-interpretation* of V is a tuple I = < IR, IP, IEXT, IS, IL, LV > (elements of this tuple stand for Resources, Properties, Extensions, Semantic mapping, Literal mapping and Literal Values).

- IR is called the *universe* and represents the world described by a RDF graph. It is a set of arbitrary elements that contains the denotations of urirefs from V. Another way to state this is that the semantics of URI references is given by elements in the universe that they map onto.

- IP is a subset of IR and contains the denotations of all the properties from V. Properties are urirefs that are used to label the edges of RDF graphs, or predicates. In RDF interpretations, these are the extension of rdf:Property class.

- IEXT is a mapping from IP, the set of properties, into $2^{IR \times IR}$, and defines the extensions of properties. For every property in V, IEXT gives the set of all object-subject pairs that make this predicate true.

[1] A very common datatyped interpretation is based on XML Schema Datatypes [BM00]

- IS is the foundational semantic mapping IS: V → IR that assigns denotations from the domain of discourse to URI references.

- IL is the mapping from the set of typed literals to the set LV that is a superset of all untyped literals and a subset of IR.

The RDFS vocabulary interpretation also defines *class extensions* – for every c, it is the set ICEXT(c) = {x ∈ IR | <x,c> ∈ IEXT (IS(rdf:type))} and also the set of all classes IC = ICEXT(IS(rdfs:Class)). The rdf: and rdfs: prefixes mean http://www.w3.org/1999/02/22-rdf-syntax-ns# and http://www.w3.org/2000/01/rdf-schema#, respectively. Such definitions are examples of special treatment of certain vocabulary terms.

Apart from these requirements, there are several more conditions that need to be true for every T-interpretation — two examples are the requirement that extensions of all datatype classes must be subsets of LV and that IEXT(IS(rdfs:subClassOf)) is a transitive relation.

4.2.3 RDF/S VOCABULARIES

The meaning of certain urirefs is fixed by the RDF and RDFS standards. For some of these, direct model-theoretic semantics are given to express their intended interpretation in a formal way, while others are described only informally (and therefore not used in subsequent chapters of this thesis). This section presents a brief overview:

RDF VOCABULARY

- **Semantic vocabulary:** These urirefs must satisfy certain formal requirements that partially define their actual meaning — rdf:type is used for assigning individuals to their types, rdf:Property is a type of all urirefs used as predicates, rdf:nil and rdf:List serve as types for list structures, and rdf:XMLLiteral is for typing XML documents embedded in RDF graphs.

- **Reification vocabulary:** RDF model theory is not expressive enough to be able to give formal background to this group of statements, although their meaning is quite clear. rdf:Statement is a type for individuals that represent or *reify* individual triples; rdf:subject, rdf:predicate and rdf:object are properties that denote the three elements of such triples.

- **Container vocabulary:** Several types of containers can be constructed using standard RDF, but since this vocabulary is so limited, most "natural" assumptions concerning RDF containers are not formally sanctioned by the model theory. rdf:Seq is a type for ordered collections, rdf:Bag is used for unordered ones and rdf:Alt offers a choice among several alternatives. Members are connected to these containers with membership properties rdf:_1 rdf:_2 etc.

- **Collection vocabulary:** RDF contains vocabulary for creating linked lists (of type rdf:List). Each node in such list contains an individual denoted by rdf:first and a connection to the rest of the list, rdf:rest. Lists are terminated by inserting an empty list at the end — rdf:nil. Note that in contrast to containers, linked lists can be constructed to only contain a given number of items. However, since this definition is only informal, a node that has several first and rest properties is still formally correct.

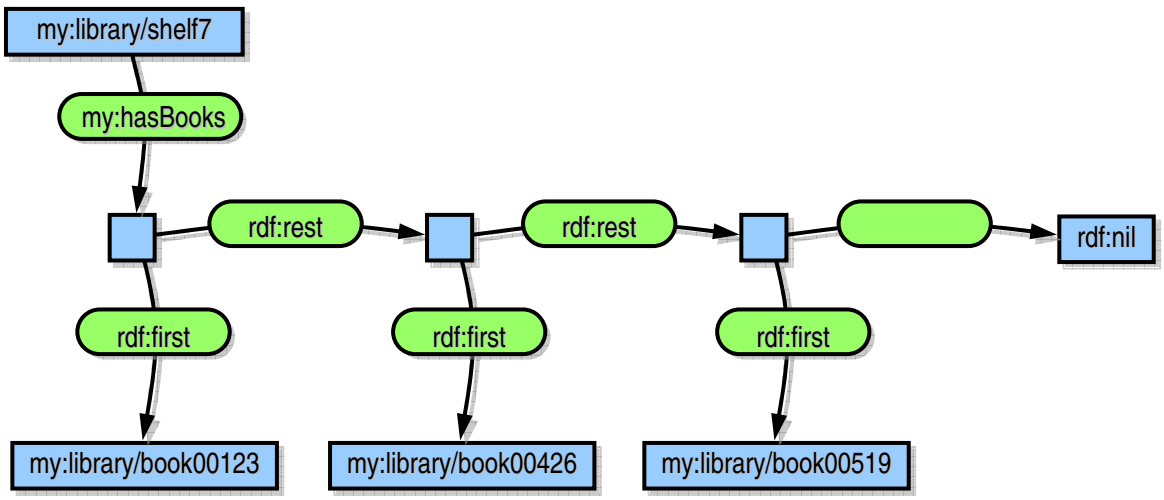


Figure 4.3 Example of a RDF list collection — books on a library shelf

- **rdf:value** is intended to designate a single characteristic value when there is more than one possibility, but its precise meaning must be usually derived from the context.

RDFS Vocabulary

RDF Schema (RDFS, [Hayes04]) is a natural extension of RDF that adds a number of modeling constructs and increases expressing power to form a lightweight ontological language. Many RDF applications actually use RDFS and in many cases, including W3C specifications, these two standards are discussed together. Formally, RDF Schema defines several vocabulary terms mostly connected with classes and offers a new kind of interpretation that takes these into account.

- **Basic typing:** With the introduction of classes came several ones that can be used in any RDFS graph — rdfs:Resource is the class for all things in the universe; rdfs:Class contains all classes (it is therefore true that rdfs:Class rdf:type rdfs:Class.); rdfs:Literal denotes literal values from LV; and rdfs:Datatype contains all literal values in datatyped interpretations.

- **Domains and ranges:** Properties are associated with classes by rdfs:domain and rdfs:range restrictions. The subject of every triple must be a member of the domain class specified for the predicate, and its object must be a member of the class given by its range restriction.

- **Subclassing:** Multiple inheritance of classes is expressed using rdfs:subClassOf. The formal definition is one of a subset: all members of a subclass must be members of its superclass.

  In many ontological languages, properties exist independent of classes and have their own inheritance hierarchy. In case of RDFS, this is accomplished with rdfs:subPropertyOf. Both of the specialization concepts are transitive.

- **Containers:** RDFS has properties and classes that contain the RDF container vocabulary. Thus, rdfs:Container is a superclass of rdf:Bag, rdf:Seq and rdf:Alt. Properties like rdf: 1 are subproperties of rdfs:member and members of the rdfs:ContainerMembershipProperty class (which itself is a subclass of rdf:Property).

- **Other urirefs:** These urirefs actually lack any formal model-theoretic definition. Their intended meaning is as follows: rdfs:comment is used to insert arbitrary notes into RDF documents; rdfs:seeAlso may contain a link to another RDF document and rdfs:isDefinedBy suggests that a given

document may contain the definition of a given uriref. Finally, rdfs:label is used to designate human-readable labels for urirefs.

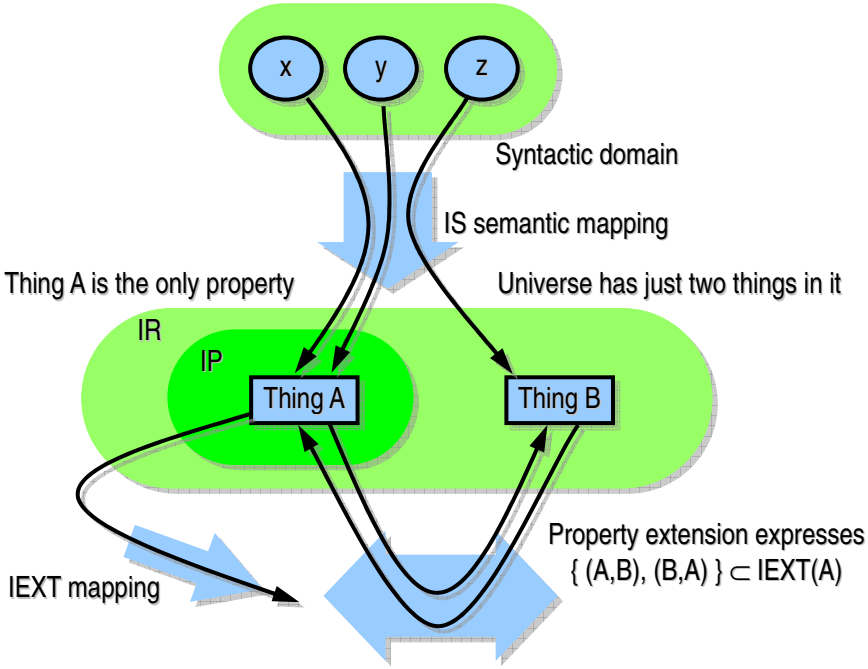


Figure 4.4 An example of RDF model theory

4.3 OWL

Several standards for defining data models of object-oriented databases were overviewed in the previous chapter; however, complex data models can be found even within the scope of the Semantic Web in the form of *ontological languages*.

According to a common definition from [GO94], *"ontology is a formal specification of a shared conceptualization"*. Compared to object models, there are at least two important distinguishing aspects:

- Ontologies attempt to capture a *shared* model that can be used, specialized, and exchanged by multiple parties, whereas object models mostly aim at a specific application and its implementation.

- Ontologies are meant to be used for defining semantics of information about the real world that can be used for automatic inference of facts, while object models mostly focus on defining how to store and structure data, not what the data means [Fensel01].

Despite these differences, ontologies and object models actually use many identical or similar concepts. That is why this section describes the object model of an ontological language from the same point of view as OODB standards earlier in the thesis.

INTRODUCTION

OWL (Ontology Web Language) recently became a W3C recommendation that evolved from DAML+OIL [CvHH01], the connection of American DAML

(DARPA Agent Markup Language) and European OIL (Ontology Inference Language). Ontologies and knowledge bases built with OWL consist of definitions of classes, properties, individuals and constraints, and have the form of RDF graphs. As a whole, OWL is actually a collection of three languages with increasing complexity:

- **OWL Lite**, a simple language that mainly supports class hierarchies and several simple constraints such as cardinality of 0 and 1.

- **OWL DL** (corresponding to description logics) includes the whole OWL vocabulary and is interpreted under several minor constraints, primarily type separation — the sets of individuals, classes and properties must be pairwise disjoint. Thanks to these constraints, OWL DL entailments are decidable.

- **OWL Full** has the same vocabulary as OWL DL, but interpreted more broadly — like in RDF, a class can simultaneously be an individual. This and other differences are primarily of interest to the advanced user.

4.3.2 OWL DATA MODEL

- **Specifications and Implementations.** As an ontological language, OWL does not include implementation details. No behavior is specified and the semantics of types end at the class level.

  > An example of this is the treatment of elementary data types — their semantics is neither given nor enforced by OWL. A property can have type xsd:integer from XML Schema Datatypes but the fact that integers are numbers is given by semantics outside and beyond the scope of OWL itself.

- **Object Types.** Individuals (instances, objects) can be grouped into classes. An object may belong in zero or more classes. These classes are used for classification, and their instances do not need to conform to any structural requirements. OWL DL also introduces intensional class definitions where a class is automatically populated with all individuals that have a given value of some property, and classes given by direct enumeration of their members.

  Complex Classes. OWL DL allows the definition of classes using set operators like intersection, union and even complement. A restriction may require a pair of classes to be disjoint.

- **Subtyping and Inheritance.** OWL supports unlimited multiple inheritance relationships with semantics that require the extent of a subconcept to be a subset of the superconcept's extent. Inheritance hierarchies can be specified among classes and properties alike.

- **Literals.** Most objects or individuals are identified using a unique ID and assigned to a class, but *datatypes* with meaning specified outside of OWL can be used to represent property ranges like strings, numbers or evaluation types. A common set of datatypes is defined by XSD [BM00] — for example 123^^xsd:Integer in XSD denotes a number.

- **Metadata.** In OWL, there is no difference between defining individuals and classes. Furthermore, OWL Full does not require data and metadata to be disjoint. Some metadata can also be stated about the whole ontology — its version, URI and the list of ontologies it imports (and extends).

- **Attributes and Relationships.** These are called *properties* in OWL. Properties are defined independent of classes, although the domain and range of each property can be given or inherited by a subproperty, and property ranges for individual classes can be restricted. Properties can also have specific cardinality constraints or be an *inverseOf* another property.

There are four kinds of properties in OWL — *datatype* properties correspond to attributes with elementary values, *object* properties correspond to relationships between objects, i.e. things with a unique ID. *Annotation* properties are used to attach arbitrary notes to things, and *ontology* properties connect and describe the ontology as a whole.

Furthermore, properties can have additional characteristics typical for ontologies. A *transitive* property denotes a transitive binary relation between individuals and the same is true of *symmetric* and *functional* (or *inverseFunctional*) ones.

- **Object Identity.** Urirefs are used as unique IDs for objects, properties and individuals, same as in RDF. When serializing OWL into RDF, the following namespace is used: http://www.w3.org/2002/07/owl#. Restrictions are defined using *anonymous* classes, properties with specific simple features (like minimum cardinality), and named restrictions then inherit from these.

  Object Equivalence. Since ontologies are meant to contain consensual information intended for sharing, reuse and extending, OWL provides the means to define that a pair of classes, properties or individuals with different urirefs has ientical (or different) semantics.

- **Object Lifetimes.** OWL has no semantics for changing or updating ontologies; however, a versioning tag is provided in every OWL ontology as a hook for version control applications.

4.3.3 Example of Formal Semantics

As an ontological language, OWL has direct model-theoretic semantics. However, it also uses the RDF/S vocabulary and has RDF-compatible model-theoretic semantics, which enables it to function as an extension of the RDF/S framework. These are shown in the following example:

> The following account gives the semantics for the complementOf vocabulary term.
> The triple xxx owl:complementOf yyy. is interpreted as follows:
>
> $\forall$xxx, yyy: <IS(xxx),IS(yyy)> $\in$ IEXT(IS(owl:complementOf)) $\Rightarrow$
> {IS(xxx),IS(yyy)} $\subseteq$ ICEXT(IS(rdf:Class) $\wedge$ ICEXT(IS(xxx)) = IR – ICEXT(IS(yyy))

4.3.4 State of the Art and Implementations

In 2001, DAML+OIL was submitted to the W3C as a basis for creating OWL, and a WebOnt working group was founded. After several drafts, the OWL specification reached the final stage and became a W3C recommendation in February 2004. A suite of tests for determining the (in)correctness of a number of OWL entailments was also released and its results for existing OWL implementations are dynamically updated in RDF online form. A month later, the OWL working group was dismissed with the claim that its goals have been accomplished.

At the same time, OWL is being implemented in a number of academic and commercial projects. These include plugins for ontology design tools (such as Protégé), editors, support tools APIs and ontology suites. The W3C website mentions 13 existing reasoners in different stages of maturity with support for all three versions of OWL. Other developments can be found among syntax checkers, parsers, and complete ontologies (such as NCI cancer data).

4.3.5 Sources

The official OWL Semantics document is [PHH04], and its creator, the WebOnt Working Group (now closed), can be found at http://www.w3.org/2001/sw/WebOnt/, where the other components of the official

OWL standard can be accessed. A new page dedicated to OWL adoption and implementations was established by W3C at http://www.w3.org/2004/OWL.

4.4 SOME RDF APPLICATIONS

This section contains an overview of several applications for the Semantic Web based on RDF. Some of these are used in section 6.4 to demonstrate the model presented in this thesis on an example, and others are interesting because they work with RDF from the database perspective. Several other applications, which are only interesting if the reader wants to find out about how the Semantic Web was adopted by the industry, can be found in Appendix B.

4.4.1 CUSTOM RDF VOCABULARIES

One of the main objections to the Semantic Web is that it has not gained wide adoption by the industry and there is no motivation for companies to publish RDF data. This section gives examples of several RDF vocabularies that are gaining wide acceptance as of the time of writing.

DUBLIN CORE METADATA INITIATIVE

The Dublin Core Metadata Initiative (DCMI, http://www.dublincore.org/) is a set of *"elements"* (properties) for describing information resources (and hence, for recording metadata). It is now a NISO standard that has become widely used in documenting Internet resources, partly because it is very simple (the standard itself is about 3 pages long). Many new RDF vocabularies use or extend Dublin Core properties, which can be seen in the sections below.

The most recent set of Dublin Core elements is defined in [DC03]. Element definitions include recommendations for unambiguous typing of their values (in parentheses), which provides the elements with clear semantics.

Dublin Core contains definitions for the following properties:

- **Title, Identifier**: A formal name of the resource and a unique identifier for the resource within its context (URI, URL, DOI, ISBN).
- **Subject, Description, Type**: A list of keywords or phrases (ideally from some controlled vocabulary); an overview of the resource in form of free-text, abstract or a table of contents; and the nature or genre of the content (DC–types vocabulary).
- **Creator, Publisher, Contributor**: Name of an entity — person, organization, or service — primarily responsible for making, publishing and contributing to the content of the resource.
- **Source, Relation**: A reference to a resource from which this one is derived; a reference to a related resource (in the same form as identifiers).
- **Language, Coverage, Rights**: A language of the resource (ISO 3066); extent of its content, typically spatial information (TGN, Thesaurus of Geographic Names), time period (ISO 8601) or jurisdiction; and information about rights held over it such as copyright or intellectual property rights.
- **Format, Date**: The digital or physical manifestation of the resource (MIME for Internet media types) and a date, typically of its creation or publication (ISO 8601).

Information using the Dublin Core elements may be represented in any suitable language (e.g., in HTML meta elements); however, RDF is an ideal representation for Dublin Core information, and a widely used one. Moreover, basically all of these controlled vocabularies exist in form of RDF URI identifiers or XML Schema Datatypes, and can be formally expressed in RDF.

A description of a World Wide Web page in DC/RDF could look like this:

```
<rdf:RDF   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:dc="http://purl.org/dc/elements/1.1/">
    <rdf:Description rdf:about="http://www.fit.vutbr.cz/">
        <dc:title>SODA – Semantic Object–Oriented Database Program</dc:title>
        <dc:publisher>Faculty of Information Technology, VUT Brno</dc:publisher>
        <dc:date>2003-05-19</dc:date>
        <dc:subject>  <rdf:Bag>
            <rdf:li>Semantic Web</rdf:li>  <rdf:li>Object–Oriented Databases</rdf:li>  <rdf:li>SODA</rdf:li>
        </rdf:Bag>  </dc:subject>
        <dc:format>text/html</dc:format>
        <dc:language>en</dc:language>
    </rdf:Description>
</rdf:RDF>
```

RSS — Remote Site Syndication

People need to access many small pieces of information on the Web on a day-to-day basis — schedules, to-do lists, news headlines, search results, "What's New", etc. As the sources and diversity of the information on the Web increases, a language was created to provide lightweight information description format for timely, large-scale distribution and reuse.

RSS 1.0 ("RDF Site Summary" RDF vocabulary, [BBD00]) is perhaps the most widely deployed RDF application on the Web; it is used in many news syndication sites and desktop readers. It keeps evolving and the recent 2.0 version is not a RDF application anymore — instead, it contains a module (Simple Semantic Resolution) that can translate its simple XML files into XML/RDF.

RSS defines properties that describe news sources and news items, including their uriref, name and short description, along with many optional features such as publication date, category, time to live or ratings. These descriptions can be transparently extended by additional modules. An example of RSS 1.0 feed from [MM04] is:

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns="http://purl.org/rss/1.0/",  xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <channel rdf:about="http://www.w3.org/2000/08/w3c-synd/home.rss">
        <title>The World Wide Web Consortium</title>
        <description>Leading the Web to its Full Potential...</description>
        <link>http://www.w3.org/</link>
        <dc:date>2002-10-28T08:07:21Z</dc:date>
        <items>
            <rdf:Seq>
                <rdf:li rdf:resource="http://www.w3.org/News/2002#item164"/>
                <rdf:li rdf:resource="http://www.w3.org/News/2002#item168"/>
                <rdf:li rdf:resource="http://www.w3.org/News/2002#item167"/>
            </rdf:Seq>
        </items>
    </channel>
    <item rdf:about="http://www.w3.org/News/2002#item164">
        <title>User Agent Guidelines Become a W3C Recommendation</title>
        <description> Written for developers of user agents, the guidelines lower barriers
            to Web accessibility for people with disabilities </description>
```

```
        <link>http://www.w3.org/News/2002#item164</link>
        <dc:date>2002-10-17</dc:date>
    </item>
    ...
</rdf:RDF>
```

4.4.2 XMP – Extensible Metadata Protocol

"XMP is an important piece that brings the Semantic Web closer to realization" Eric Miller, W3C Semantic Web Activity Lead

XMP [Adobe04] is a labeling technology for embedding metadata information in files. It was developed by Adobe Corp., is now implemented in all Adobe applications (including Photoshop, Acrobat or Illustrator) and includes a downloadable kit for embedding the technology in other software packages and file types. Current version of the specification provides the means for inserting XMP data into the following filetypes: TIFF, JPG, JPG 2000, GIF, PNG, AI (Adobe Illustrator), PSD (Adobe Photoshop), and SVG/XML images; and HTML, PDF, Postscript, XML and EPS documents.

The XMP data model is based on RDF/XML, although it has several limitations — the top-level resources it describes are always documents or their portions, serialization to XML only has limited syntax, some RDF tags are ignored (rdf:ID, rdf:parseType='Literal', rdf:aboutEach). On the other hand, XMP supports most RDF concepts, including advanced ones like blank nodes, complete container vocabulary, and even reification. For each property, it also defines its range and type (among these are Boolean values, several integer and real formats, vocabulary choices, dates, MIME types, and dimensions).

XMP uses RDF vocabularies from other sources (Dublin Core and a RDF version of EXIF schemas [EXIF02]) and defines several new vocabulary extensions for areas like:

- **Basic metadata** like the unique identifier of the resource (including the name of identification system), timestamps, thumbnails, and information about the tool that created the document
- **Rights management**, including copyright owners, usage terms, and a reference to an online rights management certificate or a WWW page with human-readable information about the rights
- **Workflow automation**, which includes versioning information, history, references to corresponding management systems and their user interfaces, and references to any jobs that this document participates in
- **Application-specific properties** that include page and document size, PDF version and keywords, Photoshop metadata etc.

Adobe XMP serves as a good example of a large company adopting and promoting the RDF standard as a foundation for the emerging Semantic Web.

4.4.3 Jena

The Semantic Web is not only standards. For any kind of development activity, *tools* are also necessary. *Jena 2.0* [CDD03], an open-source effort supported by Hewlett–Packard laboratories is one of them —

a "Semantic Web framework" that provides the user with tools for developing Semantic Web applications. It is written in Java and available at http://jena.sourceforge.net. Jena consists of:

- **RDF API** that can manipulate RDF graphs (triple-oriented or resource-oriented), and includes import and export for RDF/XML (with its own ARP parser), N3 [BernersLee02] and N-Triples [MM04]. It has full support for RDF constructs (containers, typed literals) and allows the application to extend the behavior of resources, for example by adding method calls.

- **Persistence subsystem** provides RDF graphs with back-end database storage in relational databases (MySQL, Oracle and PostgreSQL, portable to any SQL DBMS). The user can influence the degree of denormalization for tables and set the optimum balance between speed and storage size. While other RDF databases often include optimizations for RDFS classes (as standalone tables), this is not the case with Jena, which leads to more flexibility in case of schema changes.

- **Reasoning Subsystem** provides a generic rule based inference engine extensible with arbitrary rule sets. Current configurations available for the engine include a stable implementation of RDFS and a development version of OWL/Lite.

- **Ontology Subsystem** to support programmers who are working with RDF-based ontology data, specifically OWL, DAML+OIL and RDFS. The Java classes for RDF Resource and Property are extended to model more directly classes, properties and relationship found in ontologies. The ontology API works closely with the reasoning subsystem to derive additional information from ontologies. The subsystem also includes a document manager for imported ontologies.

- **Query language** (RDQL) for RDF data whose implementation is coupled to relational database storage so that optimized query is performed over data held in a Jena relational persistence store. Some of these optimizations include FastPath support and partial translation of RDQL queries into direct SQL queries.

4.4.4 TAP

TAP [GmC03] is a platform developed by IBM in cooperation with Stanford. It aims at resolving different concerns that appear as the Semantic Web is enabled in form of a worldwide virtual database. With the growing amount of RDF data, it becomes necessary to concentrate on not only RDF logical foundations, inference engines and ontological extensions, but also issues that enable the networking of Semantic Web data in a simple and unified way. TAP tries to address the following issues:

- **A query language** called GetData that could be provided by all Semantic Web servers. While numerous RDF query languages already exist, GetData is intended to be very simple and predictable so that query processing is limited to using just a small amount of server's computational resources (unlike, for example, SQL), and queries are easy to formulate and evaluate.

  A GetData query simply states the *subject* and *property name* and returns the *object* of a corresponding RDF triple. Additional commands include *searching* resource names by a substring, getting *all properties* of a given object, and querying for the *object* of a triple. GetData runs as a Web service and communicates using the SOAP protocol [BEK00]. In addition, it retrieves data globally, not just from one given server — in a way, this is similar to the distributed DNS domain name service, only instead of retrieving an IP address for a host name, it retrieves RDF triples.

- **Semantic Negotiation** (earlier called *reference by description*) tries to overcome the problem of reasoning about resources that have the same meaning but different parties use different unique identifiers for them. Resources are matched based on the values of their properties instead of their ID itself. When two parties communicate with each other, they start out with a small shared

vocabulary (like Dublin Core metadata [DC03]) and continue to agree on a growing number of vocabulary terms.

- **Trust management** is necessary to make the Semantic Web useful for business purposes. Since anyone can publish any RDF information, not all sources on the Internet can be trusted. TAP creates a *web of trust* by building a network of registries that contain lists of other trusted registries, so the user just lists several trusted registries and they, in turn, only provide information that can be found within *their own* web of trust.

The TAP system currently consists of: *TAPache*, a module for the Apache HTTP server[1] that implements the GetData interface for RDF files located in a special directory, much like what the server does with HTML pages; a *client library* with and application programming interface that allows user programs to obtain and consume data obtained through GetData; and a *registry* that contains a mechanism similar to the DNS registry, implements the tools for the distributed Web of Trust and caches query results.

4.4.5 The Mozilla Project

Mozilla [SA00][2] is a popular open-source communicator that includes a Web browser, e-mail and news client, and HTML composer. It is less known that it also provides a complete framework for development of cross-platform distributed applications, and that its internal data representation format is based on RDF. This gives a very interesting example of using the Semantic Web data format within an environment of a large software system that stretches the common understanding of "Web resources" to the practical level of application development.

The Mozilla framework is based on a cross-platform runtime platform (NSPR), a distributed component model (XPCOM), a language for defining and extending user interfaces based on XML and CSS (XUL/XBL), JavaScript (or C++) programming, and RDF.

In Mozilla, RDF *datasources* are transparently accessible through the component model or through a custom implementation. The component framework encapsulates them and adds some unique manipulation possibilities — RDF graphs can be stored and used both locally and remotely, portions of RDF graphs can easily be exchanged or integrated between components or different computers, and every application can define its own way of handling graph updates. Displaying RDF data is possible through XUL templates that match property names to template expressions, automatically display the graph structure and provide means for conditional formatting and layout of different parts of the RDF graph.

In Mozilla, RDF datasources are currently used to store information about bookmarks, remote site maps, local filesystems, networks of "related links", Web search engines, browsing profiles, address books etc.

---

[1] http://www.apache.org/

[2] http://www.mozilla.org/

5 SEMANTIC WEB AS AN OBJECT DATABASE

> *"Resource Description Framework (RDF) is a framework for representing information in the Web."*[KC04]

Although the Semantic Web is mostly mentioned from the viewpoint of ontologies or artificial intelligence, managing a body of information is a typical *database problem*. From this perspective, the Semantic Web can and should be viewed as a worldwide database[1]. Obviously, it is widely distributed and not centrally managed, often incomplete or inconsistent and very loose in its format — but it still is a database, albeit not a relational one. There are, however, other types of databases as well, ones whose structure is very close to the graph nature of RDF and the Web.

This section considers the Semantic Web as an object-oriented database. At the core of an OODB is a set of uniquely identifiable objects connected by relationships; more OODB principles are elaborated in chapter 3 and below. This chapter lists similar and corresponding concepts in object-oriented databases and the RDF-based Semantic Web, overviews design choices for object models and RDF, and discusses how the two can enrich each other by applying database concepts to the Semantic Web.

5.1 SIMILARITIES

When comparing the areas of the Semantic Web, ontologies and agents with the areas of object databases and their application layers, it turns out that there are some important similarities. Certain structural elements are almost identical in the two models. Some of them include:

- **Unique identifiers** — unique object identifiers are a strong requirement in every OODB — they allow objects to be connected in relationships. It is obviously important to note the scope of the word "unique". Several years ago, this would have probably meant "unique within an application". Today the information systems community strives for open and interoperable solutions. Uniqueness across an application, all installations of a given database product or even an operating system is not enough. What object databases need is some kind of resource identifiers with uniform structure and *worldwide* validity. Uniform Resource Identifiers [BFM98] are one such system in wide use — and they also underlie the Semantic Web RDF framework. With a suitable data model, the ability to uniquely identify and publish data and their relationships would practically constitute a loosely bound, worldwide, distributed object-oriented database.

- **Object-oriented schemas** — compared to relational schemas, object-oriented ones are much richer and closer to the way humans think about problems. This is also true of designing knowledge-based schemas (ontologies). The result is that these two paradigms have many

[1] see [GmC03] or its description in section 4.4.4

concepts in common and share identical semantics (see Figure 5.1) and it would be beneficial for the ontological and object design communities to cooperate more closely.

- **Graph-theoretic foundation** — In 1990, [Atkinson90] admits that the OODB community has not reached a consensus on a common theoretical foundation. While relational databases firmly stand on relational algebra, the situation in object databases has not changed. However, many models contain elements from graph theory, either as a supplement to a set-theoretical model ([LRV92]), a pure graph model or category theory ([NR95], [Tuijn94], [Schewe95], [Kolenčík98]). Graph theory is suitable for expressing relationships or inheritance hierarchies.

  The underlying paradigm for the RDF layer of the Semantic Web is model theory that interprets graphs. An opportunity is open for laying a shared theoretical foundation for both the Semantic Web and object-oriented databases, which would essentially connect their two worlds together.

- **Description logics** — Deductive object-oriented databases use description logics for areas ranging from describing the database schema through type checking and query languages to update semantics (F-logic, Transaction logic, HiLog [Kifer95]). Deductive databases are being extended to embrace the Semantic Web [YK00]. Since RDF model theory is also built on existentially quantified first order logic, yet another connection appears between the two areas.

- **RDF databases** — research that focuses on this emerging area works to understand how RDF data can be organized, stored, queried, and processed. All of these efforts are more natural in an object-oriented or object-relational database than in a purely relational one.

5.2 SHARING CONCEPTS

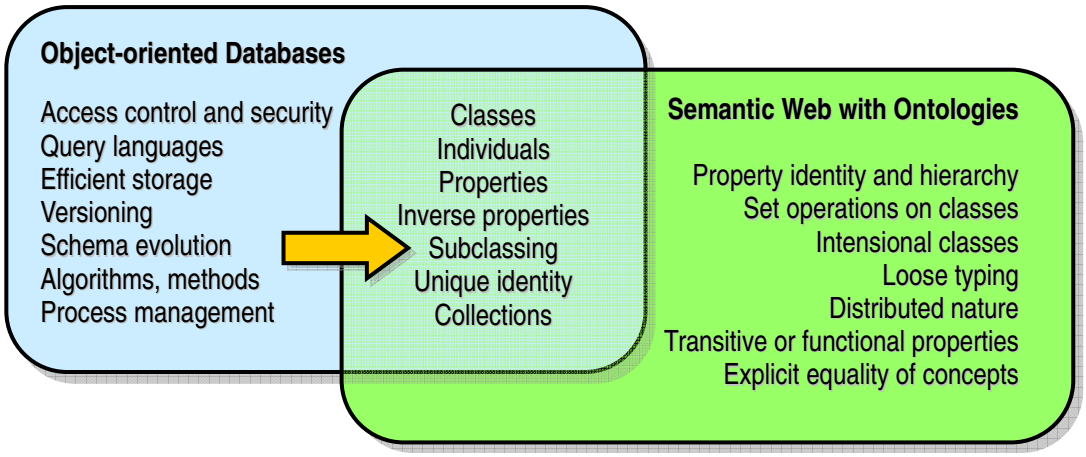


Figure 5.1 The worlds of object-oriented databases and ontologies

Figure 5.1 shows that the structure of an object model is similar to that of simple ontology languages (like RDFS), despite some semantic nuances. This inevitably suggests that some well-developed concepts from the area of object-oriented databases could be adapted and used in the context of the Semantic Web. Some examples are:

- **Access Control** — the growing Semantic Web is about to include sensitive data that cannot be accessed by everyone. Object databases offer several ways of restricting access to data graphs — permissions, views, roles, and other mechanisms. Some of these could be used for RDF graphs.

- **Query Languages** — multiple query languages for RDF data exist (e.g. RDFSuite RQL [KCA02], Sesame SeRQL [MKvH02], and Jena RDQL [Seaborne02], see appendix C). Some of them stem from SQL, others from graph formalisms; it would be interesting to see how object query languages such as OQL [CB00] or JDOQL [Craig03] adapt to RDF/S data.

- **Efficient Storage** — Today the Semantic Web still operates in small and manageable scale and there serious performance concerns are rare, but storing RDF in flat files is slowly becoming an obstacle. RDF databases try to address this problem, but they struggle with storing triples in relational tables with too many joins, no support for reification, complex data types, or inheritance [WSK03]. The Sesame RDF server already uses PostgreSQL, an object-relational system [MKvH02]. When storing RDFS data, object-oriented databases that cluster data by objects and types offer more efficiency.

- **High Level Processing** — manipulating RDF data by triples is too fine-grained for some applications. Aggregating multiple edges into objects and finding dependencies between these objects moves RDF data to a higher level of abstraction, making them easier to understand and manage. This allows object-oriented languages to work with RDF objects[1], including typing, method invocations, and custom behavior.

5.3 Data Modeling From a RDF Perspective

This section discusses the main concepts of object-oriented database models, gives a deeper rationale for their use, mentions some limitations they have in database implementations and outlines their correspondence to RDF concepts together with possible advantages this might offer.

The herein mentioned concepts are derived from two sources:

- If humans are to understand the object model, it is necessary to tie it closely to human thinking and reasoning — that is the domain of cognitive psychology [BvSvS87].

- The concepts also correspond to the four fundamental object-oriented principles as proposed at a public meeting of experts in object technology standards and SQL3 development [Sutherland93].

5.3.1 Atomic, Collection and Structure Nodes

A system is defined as a set of related *objects*. Human thinking is based on finding out about the qualities of these objects (theory of exploration and concepts) and abstracting from them (theory of meaning and classification) [BvSvS87]. Therefore, an object model should be able to express *properties* of objects.

Assigning properties to objects has two aspects. First one is that every specific feature needs to have a given place, role or meaning within the parent object. This is modeled using properties and the parent

[1] a concept called RDF objects is being developed within the Jena platform [CDD03]

object is then a *tuple*, also called a structure or a record. The number of properties is constant because each one of has distinct meaning. The second construction deals with several properties that have the same role. This is usually not modeled by having many values of the same property, but by aggregating these values within a new subobject that represents the many values of the property — a set, bag, list, array, or generally, a *collection*. The difference between these different collection types is whether they are indexed and whether multiple occurrences of the same object are allowed. Since objects cannot be decomposed infinitely, we need some *atomic objects,* also called elementary or simple objects.

In the world of relational databases, cells in tables are required to be *atomic objects*, rows are *tuples* with named fields, and tables represent *collections*.

RDF COUNTERPARTS

In RDF, it is natural to identify atomic objects with typed literals — a good and widely used choice for their typing is XML Schema Datatypes [BM00]. Tuples are naturally modeled by triples that have the same subject and specify property values, while collections can use RDFS collection syntax with some additional semantic restrictions.

5.3.2 UNIQUE IDENTIFIERS

In [Sutherland93], the first principle is: *"A first class object has unique, immutable identity within its scope in a distributed environment."* Databases in a networked world need to avoid the problem of synonyms and homonyms in natural language, and uniquely identify the objects they describe.

In the object-oriented database world of nodes, a sharp distinction exists between *objects* (reference concepts, instances) and *literals* (data concepts, values). While objects have a unique object identifier (OID) that allows them to enter into relationships and keep their distinct nature without regard to their structure, literals are always parts of objects and the only relationship that targets them comes from their parent object. From the modeling point of view, this reflects the fact that some information does not make sense alone; it should only be considered within the parent object. All literals with the same value are essentially identical.

RDF COUNTERPARTS

In RDF graphs, objects naturally correspond to individuals with *urirefs* since these denote resources that can be referenced. Literals correspond either to RDF (typed) *literals* or, in case of complex data types like collections and structures, to *blank nodes*. RDF literals cannot be subjects of triples and blank nodes do not have universal identifiers, therefore they cannot be globally referenced. Additional condition has to ensure there is only a single reference to a blank node, one from the parent object[1].

5.3.3 ATTRIBUTES AND RELATIONSHIPS

The second principle of [Sutherland93] says: *"First class links occur only between first class objects."*

There are two principal kinds of properties on the instance level of the OODB world — *attributes* and *relationships*. Properties are used to structure and relate things in tuples and collections. Attributes connect nodes to literals, and relationships connect nodes to objects. The attribute property is also called the part-of relation or aggregation. N-ary relationships are transformed to binary ones and relationships of cardinality 1-to-N or M-to-N are expressed using collections. In object-oriented

[1] see also Second Class Objects in JDO, section 3.3.3

practice, properties have unique names only within a given object, which causes name conflicts when extending objects with new properties in case of multiple inheritance [Kolenčík98].

RDF COUNTERPARTS

In RDF, properties are not bound to a specific class of objects and their naming has to be just as unique as node urirefs. In an object-oriented RDF-based database, connecting property domains and ranges to specific classes is obligatory but the advantage of unique property names is kept. In cases where a property is used in a sense more general than just a specific class (Dublin Core metadata), a strongly typed property can be subclassed from the more general one.

5.3.4 TYPING AND THE DATABASE SCHEMA

The third [Sutherland93] principle states: *"A first class object always knows what type(s) it is."* Human thinking depends on concept abstraction — D. Frege, semiotics and the theory of meaning [BvSvS87].

Another way to understand a complex system is through *categorization*, or *typing*. An object can be better understood by users and used by the system when it's known that its structure corresponds to a certain pattern or prototype. The question "what is it?" is answered in terms of object category. Computers exploit typing for efficient data storage, integrity checking, but perhaps the most importantly, binding algorithms to objects. A given algorithm works for many different inputs, but the inputs are required to conform to a certain pattern. In traditional object-oriented data models without reflection, every object has exactly one type and exactly the structure prescribed by the type.

But rigid typing can also become a limitation. Types need to evolve when our understanding of objects changes, they need to be flexible when objects start playing new roles in the system [HM00]. It is sometimes desirable to add extra properties, construct or transform types dynamically or work with an object that does not provide all the information required by its type. All these properties are rare in strictly typed object-oriented systems but inherent in RDF.

RDF COUNTERPARTS

RDF allows objects to have additional properties and does not place any restrictions on properties of typed nodes. However, it makes sense to define the expected structure using a subproperty of rdf:type that would indicate full structural conformance of an object with its type. It is also important to distinguish between classes with objects that can enter into arbitrary relationships and types that are parts of these objects without separate identification whose extensions are collections, tuples and atomic types.

5.3.5 SUBTYPING AND INHERITANCE

The fourth of the [Sutherland93] principles says: *"An instance of a subtype is always an instance of its supertype."* Cognitive psychology calls this principle in human thinking *hierarchization* [BvSvS87].

There are several views of what inheritance actually is. The one used here is based on two principles: extension of a subclass is a subset of the superclass extension, and therefore all individuals of the subtype can be used in place of the supertype. This helps organize the system hierarchically, work with multiple inheritance, and handle large numbers of object in a unified way while preserving their differences (polymorphism).

RDF COUNTERPARTS

In RDFS, the notion of subclassing is specified identically with the first part of the definition, and hierarchies of classes and properties are common in Semantic Web applications.

6 DATABASE MODEL ON TOP OF RDF/S

6.1 FEATURES OF THE MODEL

The vocabulary given in this section contains foundational concepts from the area of OODB as specified by the ODMG [CB00] or JDO [Craig03] standards. Moreover, the semantics of this model is practically identical with the CDL (Concept Definition Language) of the G2 database [HM00]. Some important features of the model[1] are:

- Contents of the database are separate from its schema. This is not required in RDFS interpretations, but required by most databases for efficient schema management. Formally, ICEXT(IS(soda:Thing)) ∩ ICEX`T(IS(soda:Concept)) = ∅.[2]
- Instances are strongly typed using the soda:type property. Apart from classes and literals, other fundamental types include collections (strongly typed ordered multisets) and tuples. All property values of an instance are mandatory, except where an empty collection is allowed, with the option to use an undefined type value.
- Some instances defined by the model have their own unique OID, while others are represented by literal values or blank nodes. Collections, tuples, and literals are embedded in an enclosing object, while classes have unique identifiers.
- The model supports multiple inheritance.

LIMITATIONS OF THE MODEL

- The model is limited to a static structural description, and does not formalize the dynamic aspects of a database, including updates, methods and side effects, or transactions.
- Some structural features that are commonly used for reasons of user comfort or performance but are marginal from the theoretical point of view were omitted — the model does not contain collection indices, named objects, matchcodes, or labels.

---

[1] a formal description of the model can be found in the following section. The model was published in [Güttner03c].

[2] see section 4.2.2 for a detailed description of RDF/S semantics

6.2 Obtaining the OODB Graph

In RDF, not all information about a certain vocabulary term is known to us since anyone can publish assertions about anything.

> If the Semantic Web keeps growing, different companies will publish urirefs, for example ones that denote cars — http://www.ford.com/rdf/2006/mustang/2.6TDI. When someone works with such widely known URI, there is a host of triples on many different servers around the whole planet that mention this subject — stores, repair services, car magazines, newspaper ads or customer reviews. Many of these are not relevant for a given goal, cannot be easily found or trusted.

On the other hand, in the database world it is desirable to have a single RDF graph that is coherent and represents one logical set of data that conforms to a certain schema. Without this, some important questions could not be answered in the database — for example, whether an object's structure corresponds to its type.

> An RDF graph specifies a type called my:CarType and requires that all cars have certain maximum speed and average fuel consumption. One cannot be sure if only this is required or whether some other RDF source adds more required attributes like mileage or price. In such case, the complete definition of a type could not be obtained, which would prevent the database system from enforcing strong typing constraints.

For some applications, limiting the extent of an RDF graph to the triples that are immediately accessible is necessary. In the case of a single organization that publishes its data (ranging in size from a small e-shop to a whole government) it is only natural to seek consistency and stronger typing in its own limited RDF graph or the graphs of its trusted business partners.

The complete database graph for a given task can be obtained by connecting several RDF sources (graphs) using properties similar to rdfs:seeAlso or owl:imports. In RDFS, the semantics of rdfs:seeAlso is only informal, while in OWL, the owl:imports uriref has stricter semantics that requires an OWL ontology to be present at a specific URL address [PHH04].

6.3 A Formal Description

This section presents model-theoretic semantics of a RDF vocabulary that supports the elementary notions of an object-oriented database (see chapter 3). The semantics is specified for a vocabulary comprised of urirefs that use the soda prefix (Semantic Object–oriented DAtabase) defined as xmlns:soda=http://www.fit.vutbr.cz/~guttner/soda. These semantics do not influence the semantics of core RDF or RDFS vocabularies, which allows the OODB parts of RDF graphs to have special semantics without influencing the rest of an RDF graph[1].

Every formal statement in this section is given using two types of notation with an empty line in between. One is a specification using the N-Triples syntax [MM04] where the triples must be true in an RDFS interpretation. Sometimes this is not expressive enough so quantifiers, grouping or logical

[1] This feature is especially useful for mining object data from raw RDF/S graphs, see section 7.1

conjunctions are used although they are not part of N-Triples — in these cases, the corresponding part of the description is only informative.

The second type of notation, however, is strictly formal and exact even in places where the first one goes beyond N-Triples semantics. It uses the RDF model theory explained in section 4.2.

6.3.1 Division of the Domain of Discourse

The SODA vocabulary is:

> { soda:Concept, soda:Class, soda:Tuple, soda:Collection, soda:Thing, soda:Attribute, soda:MemberAttribute, soda:Meta } ∈ IC
> { soda:type, soda:collectionOf, soda:subTypeOf, soda:member, soda:_1, soda:_2,... } ∈ IP

The soda:Meta uriref denotes a class that contains all SODA vocabulary metaexpressions.

> { soda:Meta, soda:Concept, soda:Class, soda:Tuple, soda:Collection, soda:Thing, soda:Attribute, soda:MemberAttribute, soda:Meta, soda:type, soda:collectionOf, soda:subTypeOf } rdf:type soda:Meta .
>
> {soda:Meta, soda:Concept, soda:Class, soda:Tuple, soda:Collection, soda:Thing, soda:Attribute, soda:MemberAttribute, soda:Meta, soda:type, soda:collectionOf, soda:subTypeOf} ∈ ICEXT(IS(soda:Meta))

This interpretation is very strict as far as typing goes, which is similar to strongly typed programming languages. Different levels of abstraction in the domain of discourse are strictly separate. The areas of instances (Thing), model (Concept), metamodel (Meta), tuple attributes and collection membership attributes are divided into pairwise disjoint sets (Figure 6.1).

> ICEXT(IS(soda:Meta)), ICEXT(IS(soda:Thing)), ICEXT(IS(soda:Concept)), ICEXT(IS(soda:Attribute), ICEXT(IS(soda:MemberAttribute) are pairwise disjoint.

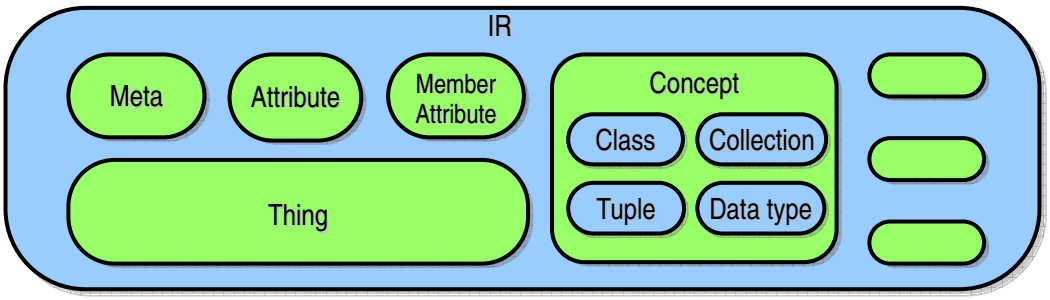


Figure 6.1 Domain of discourse division

It will be convenient to define a mapping, I2EXT, from metaclass types to extensions of classes. For example, for soda:Collection, this mapping returns all collection objects.

> I2EXT(z) = { x | ∃ y: <x,y> ∈ IEXT(IS(soda:type)) ∧ <y,z> ∈ ICEXT(z) }

6.3.2 Concepts: Classes, Collections, Tuples and Datatypes

In the following text, what is often called a *type* or a *class* will be called *concept* to convey the idea of conceptual modeling, since in the world of OODBs, *class* is a special kind of type/concept — one that

can be the target of relationships based on its uriref (OID), as opposed to collections, tuples, and literals. Semantics is defined only for part of the RDF world so the meaning of vocabulary outside of SODA remains unchanged.

```
soda:Concept rdfs:subClassOf rdfs:Class .
soda:Class rdfs:subClassOf soda:Concept .
soda:Tuple rdfs:subClassOf soda:Concept .
soda:Collection rdfs:subClassOf soda:Concept .

ICEXT(IS(soda:Concept)) ⊂ ICEXT(IS(rdfs:Class))
ICEXT(IS(soda:Class)) ⊂ ICEXT(IS(soda:Concept))
ICEXT(IS(soda:Tuple)) ⊂ ICEXT(IS(soda:Concept))
ICEXT(IS(soda:Collection)) ⊂ ICEXT(IS(soda:Concept))
```

The following subclassing statement has no semantic meaning since RDF does not impose any conditions on rdf:Bag. The statement is supposed to indicate that the informal understanding of a bag is preserved (and actually, as show later, formally imposed) in SODA. It also helps to infer object data from RDF structures.

```
soda:Collection rdfs:subClassOf rdf:Bag .
soda:Datatype rdfs:subClassOf soda:Concept .
soda:Datatype rdfs:subClassOf rdfs:Data type .

ICEXT(IS(soda:Collection)) ⊂ ICEXT(IS(rdf:Bag))
ICEXT(IS(soda:Datatype)) ⊂ ICEXT(IS(soda:Concept))
ICEXT(IS(soda:Datatype)) ⊂ ICEXT(IS(rdfs:Datatype))
```

This division of the following class extents is depicted in Figure 6.1.

```
ICEXT(IS(soda:Class)), ICEXT(IS(soda:Tuple)), ICEXT(IS(soda:Collection)), ICEXT(IS(soda:Datatype)) are
pairwise disjoint and their union is ICEXT(IS(soda:Concept))
```

6.3.3 Typing as a Mandatory Individual–Type Relationship

The notions of typing and subtyping are introduced here. They connect individuals to their types and types among each other, imposing structure on the individuals that corresponds to the attributes and other properties of their types and supertypes. Structural restrictions on instances of different kinds of concepts are described in the following sections.

SODA typing is a special case of RDF typing that has additional semantics but does not alter the original rdf:type semantics.

```
soda:type rdfs:subPropertyOf rdf:type .
soda:type rdfs:domain soda:Thing .
soda:type rdfs:range soda:Concept .

IEXT(IS(soda:type)) ⊂ IEXT(IS(rdf:type))
<x,y> ∈ IEXT(IS(soda:type)) ⇒
    x ∈ ICEXT(IS(soda:Thing))
<x,y> ∈ IEXT(IS(soda:type)) ⇒
    y ∈ ICEXT(IS(soda:Concept))
```

Every instance in the SODA world must have at least one type, and everything that is SODA typed is a SODA instance.

```
xxx rdf:type soda:Thing . ⇔ ∃ yyy: xxx soda:type yyy .
```

```
x ∈ ICEXT(IS(soda:Thing)) ⇔
    ∃ y: <x,y> ∈ IEXT(IS(soda:type))
```

Blank nodes cannot represent class instances since they do not have a universal OID — uriref.

```
_xxx soda:type yyy . ⇒ yyy rdf:type (soda:Tuple ∪ soda:Collection) .

_xxx is a blank node ∧ <IS+A(_xxx),y> ∈ IEXT(IS(soda:type)) ⇒
    y ∈ ICEXT(IS(soda:Tuple)) ∪ ICEXT(IS(soda:Collection))
```

SODA subtyping is a special case of RDFS subtyping.

```
soda:subTypeOf rdfs:subPropertyOf rdfs:subClassOf .

IEXT(IS(soda:subTypeOf)) ⊂ IEXT(IS(rdfs:subClassOf))
```

Subtyping is a transitive property. This requirement needs to be restated for SODA subtyping since the transitivity of rdfs:subClassOf could have been lost by application of rdfs:subPropertyOf.

```
xxx soda:subTypeOf yyy . ∧ yyy soda:subTypeOf zzz . ⇒ xxx soda:subTypeOf zzz .

{<x,y>,<y,z>} ⊂ IEXT(IS(soda:subTypeOf)) ⇒ <x,z> ∈ IEXT(IS(soda:subTypeOf))
```

The only semantic difference between the RDFS and SODA versions of subtyping are that the latter is more strictly typed — it can only be applied to concepts. According to this definition, it is for example possible to assert that some collection is a subtype of some class, but it is then required that the collection have an empty extension.

```
soda:subTypeOf rdfs:domain soda:Concept .
soda:subTypeOf rdfs:range soda:Concept .

<x,y> ∈ IEXT(IS(soda:subTypeOf)) ⇒
    x ∈ ICEXT(IS(soda:Concept))
<x,y> ∈ IEXT(IS(soda:subTypeOf) ⇒
    y ∈ ICEXT(IS(soda:Concept))
```

A subtype needs to conform to the structure of a supertype, but this requirement is implicitly given by the subset definition of RDFS subtyping. This means that no special rules are required for the subtype to obtain the type of all its supertypes and all their attributes.

6.3.4 COLLECTIONS

In this section, a subset of RDF collection concepts is given additional semantics that correspond to collection as an (ordered) multiset.

```
soda:MemberAttribute rdfs:subClassOf rdfs:ContainerMembershipProperty .
soda:member rdf:type soda:MemberAttribute .
∀ n ∈ N: soda:_n rdfs:subPropertyOf soda:member .
∀ n ∈ N: soda:_n rdfs:subPropertyOf rdf:_n .

ICEXT(IS(soda:MemberAttribute)) ⊂
    ICEXT(IS(rdfs:ContainerMembershipProperty))
IS(soda:member) ∈ ICEXT(IS(soda:MemberAttribute))
∀ n ∈ N: IEXT(IS(soda:_n)) ⊂ IEXT(IS(soda:member))
```

$\forall$ n $\in$ N: IEXT(IS(soda:_n)) $\subset$ IEXT(IS(rdf:_n))

The following two rules specify the domain and range of membership properties.

xxx soda:member yyy . $\Rightarrow$
$\exists$ ccc, ddd: xxx soda:type ccc . $\wedge$ ccc rdf:type soda:Collection .
$\wedge$ yyy soda:type ddd . $\wedge$ ddd rdf:type soda:Concept .

<x,y> $\in$ IEXT(IS(soda:member)) $\Rightarrow$
x $\in$ I2EXT(IS(soda:Collection)) $\wedge$ y $\in$ I2EXT(IS(soda:Concept))

Every collection concept has exactly one property soda:collectionOf, which indicates the type of objects contained in the collection.

xxx rdf:type soda:Collection . $\Rightarrow$ $\exists$! yyy: xxx soda:collectionOf yyy .
soda:collectionOf rdfs:domain soda:Collection .
soda:collectionOf rdfs:range soda:Concept .

x $\in$ ICEXT(IS(soda:Collection)) $\Rightarrow$
$\exists$! y: <x,y> $\in$ IEXT(IS(soda:collectionOf))
<x,y> $\in$ IEXT(IS(soda:collectionOf)) $\Rightarrow$
x $\in$ ICEXT(IS(soda:Collection))
<x,y> $\in$ IEXT(IS(soda:collectionOf) $\Rightarrow$
y $\in$ ICEXT(IS(soda:Concept))

All elements of a collection need to have the prescribed type.

xxx soda:member yyy . $\wedge$ xxx soda:type ccc . $\wedge$ ccc soda:collectionOf ddd . $\Rightarrow$
yyy soda:type ddd .

<x,y> $\in$ IEXT(IS(soda:member) $\wedge$ <x,c> $\in$ IEXT(IS(soda:type))
$\wedge$ <c,d> $\in$ IEXT(IS(soda:collectionOf)) $\Rightarrow$
<y,d> $\in$ IEXT(IS(soda:type))

6.3.5 Object Attributes

Attributes of an instance point to different parts of the instance's data — references to other objects, aggregated tuple values or collections and elementary data types, or literals. The attributes of an object are given by its type, although an object can also have additional attributes.

Both classes and tuples can have attributes, and an attribute must indicate its domain.

aaa rdf:type soda:Attribute . $\Rightarrow$
$\exists$ ccc: aaa rdfs:domain ccc . $\wedge$
ccc rdf:type (soda:Class $\cup$ soda:Tuple) .

a $\in$ ICEXT(IS(soda:Attribute) $\Rightarrow$
$\exists$ c: <a,c> $\in$ IEXT(IS(rdfs:domain)) $\wedge$
c $\in$ ICEXT(IS(soda:Class)) $\cup$ ICEXT(IS(soda:Tuple))

An attribute of a given instance can either be a reference to an object, or a tuple or collection, or a literal value. Every attribute has to specify its range.

aaa rdf:type soda:Attribute . $\Rightarrow$ $\exists$ ccc: aaa rdfs:range ccc . $\wedge$ ccc rdf:type soda:Concept .

$$a \in ICEXT(IS(soda:Attribute)) \Rightarrow \exists\, c: \langle a,c\rangle \in IEXT(IS(rdfs:range)) \wedge c \in ICEXT(IS(soda:Concept))$$

This rule is at the core of strict SODA typing. A given instance must have all the attributes that its type prescribes. An exception to this is an undefined reference to a class instance.

$$xxx\ soda:type\ ccc\ .\ \wedge\ ccc\ rdf:type\ (soda:Class \cup soda:Tuple)\ .\ \wedge\ aaa\ rdf:type\ soda:Attribute\ .\ \wedge\ aaa\ rdfs:domain\ ccc\ .\ \wedge\ ccc\ rdf:type\ (soda:Collection \cup soda:Tuple \cup soda:Datatype)\ .\ \Rightarrow\ \exists\, yyy: xxx\ aaa\ yyy\ .$$

$$x \in I2EXT((IS(soda:Class)) \cup I2EXT(IS(soda:Tuple)) \wedge a \in ICEXT(IS(soda:Attribute)) \wedge \langle x,c\rangle \in IEXT(IS(soda:type)) \wedge \langle a,c\rangle \in IEXT(IS(rdfs:domain)) \wedge c \in ICEXT(IS(soda:Collection)) \cup ICEXT(IS(soda:Tuple)) \cup ICEXT(IS(soda:Datatype)) \Rightarrow \exists\, y: \langle x,y\rangle \in IEXT(a)$$

The following rule formalizes what was already mentioned elsewhere — there should not be multiple attribute triples with the same subject and predicate. Such cases must be transformed to collections.

$$xxx\ aaa\ yyy\ .\ \wedge\ xxx\ aaa\ zzz\ .\ \wedge\ aaa\ rdf:type\ soda:Attribute\ .\ \Rightarrow IS(yyy) = IS(zzz)$$

$$\langle x,y\rangle \in IEXT(a) \wedge \langle x,z\rangle \in IEXT(a) \wedge a \in ICEXT(IS(soda:Attribute)) \Rightarrow y = z$$

6.3.6 Reference Semantics for Tuples and Literals

The difference between classes and the other concepts is that collections, tuples and literals are always parts of an aggregated object — they do not have OIDs and cannot participate in relationships.

The expression of this fact is that in a RDF graph, there is always one and only one reference to the instances of these non-class concepts, because within an object, the structure of its data is always a tree graph. In most cases, this can be conveniently expressed using a blank node since beside the enclosing object, nothing needs to refer to it.

$$xxx\ soda:type\ ccc\ .\ \wedge\ ccc\ rdf:type\ (soda:Tuple \cup soda:Collection \cup soda:Datatype)\ .\ \wedge\ yyy\ aaa\ xxx\ .\ \wedge\ zzz\ bbb\ xxx\ .\ \wedge\ \{a, b\}\ rdf:type\ (soda:Attribute \cup soda:MemberAttribute)\ .\ \Rightarrow IS(xxx) = IS(yyy)$$

$$\forall n \in N: x \in I2EXT(IS(soda:Tuple)) \cup ICEXT(IS(soda:Collection)) \cup ICEXT(IS(soda:Datatype)) \wedge \langle y,x\rangle \in IEXT(a) \wedge \langle z,x\rangle \in IEXT(b) \wedge \{a, b\} \subset ICEXT(IS(soda:Attribute)) \cup ICEXT(IS(soda:MemberAttribute)) \Rightarrow x = y$$

Collections and tuples (and, obviously, literals) cannot exist in the database without an enclosing object (an instance of a class) that refers to them.

$$\forall\, n \in N: xxx\ soda:type\ ccc\ .\ \wedge\ ccc\ rdf:type\ (soda:Tuple \cup soda:Collection \cup soda:Datatype\ )\ .\ \Rightarrow \exists\, yyy, aaa: yyy\ aaa\ xxx\ .$$

$$\forall\, n \in N: x \in I2EXT(IS(soda:Tuple)) \cup I2EXT(IS(soda:Collection)) \cup I2EXT(IS(soda:Datatype)) \Rightarrow \exists\, y, a: \langle y,x\rangle \in IEXT(a)$$

Collections, tuples and literals also need to avoid being parts of each other because that would create something like an infinite data recursion. Since there is only one attribute referring to each of them, there needs to be a "path" of attributes from some class concept to these concepts (Figure 6.2).

xx_0 soda:type yy . $\wedge$ yy rdf:type (soda:Collection $\cup$ soda:Tuple $\cup$ soda:Datatype) . $\Rightarrow$
$\exists$ n $\geq$ 1, $xx_1 ... xx_n$, $yy_1 ... yy_n$, $aa_1 ... aa_n$: $\forall i \in \{1...n\}$:
xx_n aa_n xx_{n-1} . $\wedge$ … $\wedge$ xx_2 aa_2 xx_1 . $\wedge$ xx_1 aa_1 xx_0 . $\wedge$ aa_i rdf:type (soda:Attribute
$\cup$ soda:MemberAttribute) . $\wedge$ xx_n soda:type zzz . $\wedge$ zzz rdf:type soda:Class .

$x_0 \in$ I2EXT(IS(soda:Collection)) $\cup$ I2EXT(IS(soda:Tuple))
$\cup$ I2EXT(IS(soda:Datatype)) $\Rightarrow \exists$ n$\geq$1, $x_1 \ldots x_n$, $a_1 \ldots a_n$: $\forall i \in \{1 \ldots n\}$: $\langle x_i, x_{i-1} \rangle \in$
IEXT(a_i) $\wedge$ $a_i \in$ ICEXT(IS(soda:Attribute)) $\cup$ ICEXT(IS(soda:MemberAttribute))
$\wedge$ $x_n \in$ I2EXT(IS(soda:Class))

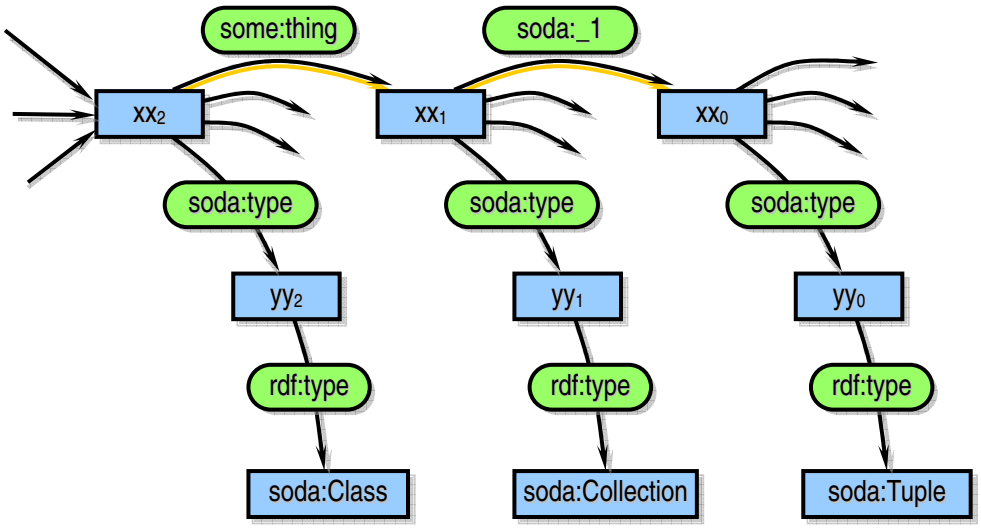


Figure 6.2 Attribute path from a tuple to class instance

6.4 EXAMPLE OF THE SODA MODEL

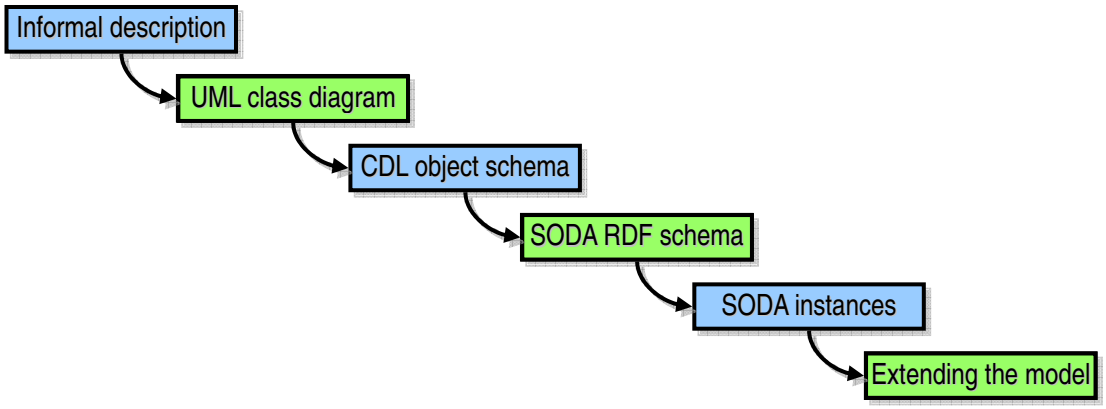


Figure 6.3 Overview of the example — development stages

This example is intended to demonstrate the whole process of building a RDF-based object-oriented database, and contains the following steps (see Figure 6.3). **First**, a UML class diagram shows an overview of the problem domain. **Second**, the data model is clearly defined using the CDL language to show what the schema would look like in a normal object-oriented database. **Third**, the description is transformed to a RDF-based schema of a corresponding SODA model (with semantics essentially identical to the CDL description). **Fourth** part contains part of RDF graph with several instances of the data that conform to the given schema. **Fifth** part concludes the whole example with extending the model to encompass several useful vocabularies for the Semantic Web.

6.4.1 DESCRIPTION OF THE ONLINE MAGAZINE DOMAIN

We need to create an information system for a simple content management application — an online magazine that publishes on the Web. The publishing team is made up of several *authors* who contribute *articles* and *photographs*. We need to store some basic information about these authors (like their *names* and *addresses*) and some metadata about the files they produced (*URL* of the files, article *abstracts* and image *thumbnails*). All of this information should be accessible to the managers of the whole project as well as other computers that can use some of the metadata to search, index or publish our content.

The selection of elements for implementing this example is aimed at demonstrating a number of concepts defined in the SODA model, while some relationships and attributes were intentionally omitted to keep the whole section simple.

6.4.2 UML DIAGRAM OF THE OM SCHEMA

The UML class diagram [OMG03] in Figure 6.4 shows the model in a way that should be simple to understand, but there are several semantic nuances and unusual features that deserve more explanation and are properly formalized in the CDL and RDF schemas below: the hasAddress aggregation relationship is simply an attribute of Author, but in this form the structure of this attribute is clearly displayed; the choice of attribute names corresponds with the RDF way of naming associations (hasName) instead of roles (Name); and many attributes which naturally belong in the model were omitted for the sake of brevity. The Products association class is actually a collection that represents a 1:N relationship from Author to Product, physically embedded in the Author class, with a hasImgNum attribute that says how many of the products in this collection by one author are images.

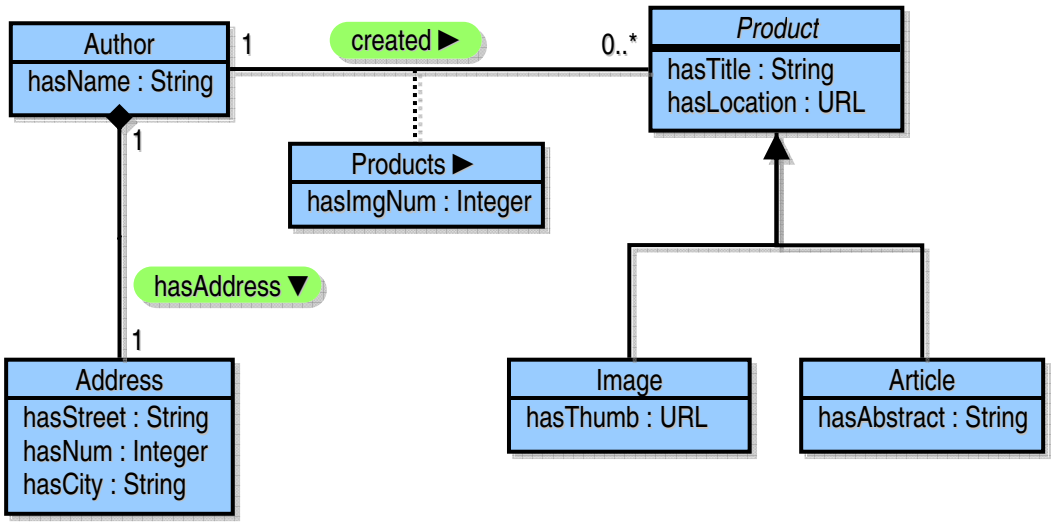


Figure 6.4 UML class diagram of the online magazine[1]

6.4.3 CDL Definition of the OM Schema

Since the SODA model is based on CDL (see section 3.5), this CDL definition captures the exact semantics of the resulting schema. Namespaces for concepts, properties and types are omitted for brevity, along with the [Mandat] (mandatory) directive for all properties:

```
Concept Author [Extent]                          // a class type for all authors in the system
Properties                                       // all properties in this model are mandatory
    hasName : String                             // name and surname of the author
    created : Products [Inverse = "createdBy"]   // a collection of references to author's products
    hasAddress : Address                         // author's correspondence address
End Concept

Concept Address [Data = Value]                   // a tuple type for storing address information
Properties
    hasStreet : String                           // street name
    hasNum : Integer                             // house number
    hasCity : String                             // city name
End Concept

Concept Product / Products [Abstract, Extent]    // a class and collection type for products
Properties
    hasTitle : String                            // title or name of the product
    hasLocation : URL                            // address of the file containing the product
    createdBy : Author [Inverse = "created"]     // a reference to the author of this product
    hasImgNum : Integer [Where = Col]            // a collection attribute of Products that says >>
End Product                                      //      >> how many of the products are images
```

[1] For a plain text explanation of what these classes, attributes and relationships represent, please look below at the annotated CDL schema definition.

```
Concept Image                          // a class for storing information about images
Inherits Product                       // every Image is a Product
Properties
    hasThumb : URL                     // address of the file containing image thumbnail
End Concept

Concept Article                        // a class for storing information about articles
Inherits Product                       // every Article is a Product
Properties
    hasAbstract : String               // a short textual description of the article
End Concept
```

6.4.4 SODA Model of the OM Schema

This section presents the RDF/S schema of the example using N-Triples notation, with the usual regular expression abbreviation for sets of similar triples. When creating a SODA model, the designer always needs to choose a namespace for representing the concepts he is creating. This example will use the following namespace: xmlns:om="http://www.example.com/om#". The rdf: and soda: namespaces were already defined in chapter 6.3.

The following description defines typing and subclassing information for all OM concepts and attributes, assigning their urirefs clear semantics. Author and Product are classes — their instances will have a system-assigned uriref, which is also required of Image and Article because these concepts subclass Product. Instances of Address will be tuples represented by blank nodes with three attributes each. The Products collection used to model 1:N relationships must also specify the type of items it stores (soda:collectionOf).

```
{ om:Author, om:Product } rdf:type[1] soda:Class .
om:Address rdf:type soda:Tuple .
om:Products rdf:type soda:Collection .
om:Products soda:collectionOf om:Product .
{ om:Image, om:Article } soda:subTypeOf om:Product .
{ om:hasName, om:hasAddress, om:created, om:createdBy, om:hasStreet, om:hasNum, om:hasCity,
om:hasTitle, om:hasLocation, om:hasThumb, om:hasAbstract } rdfs:subClassOf soda:Attribute .
```

All the urirefs that were defined as SODA attributes must also give their domains and ranges. From this point the concepts are completely defined with typed member attributes and relationships. The ranges of literals are typed by XML Schema Datatypes ([BM00]).

```
{ om:hasName, om:hasAddress, om:created } rdfs:domain om:Author .
{ om:hasStreet, om:hasNum, om:hasCity } rdfs:domain om:Address .
{ om:hasTitle, om:hasLocation, om:createdBy } rdfs:domain om:Product .
om:hasImgNum rdfs:domain om:Products .
om:hasThumb rdfs:domain om:Image .
om:hasAbstract rdfs:domain om:Article .
{ om:hasName,  hasStreet, om:hasCity, om:hasTitle, hasAbstract } rdfs:range xsd:String .
{ om:hasLocation, om:hasThumb } rdfs:range xs:anyURI .
{ om:hasNum, om:hasImgNum } rdfs:range xsd:Integer .
om:hasAddress rdfs:range om:Address .
om:created rdfs:range om:Products .
```

[1] The typing properties used here belong in the RDF/S vocabulary — their stricter SODA counterparts are only used for connecting SODA instances to classes.

```
om:createdBy rdfs:range om:Author .
```

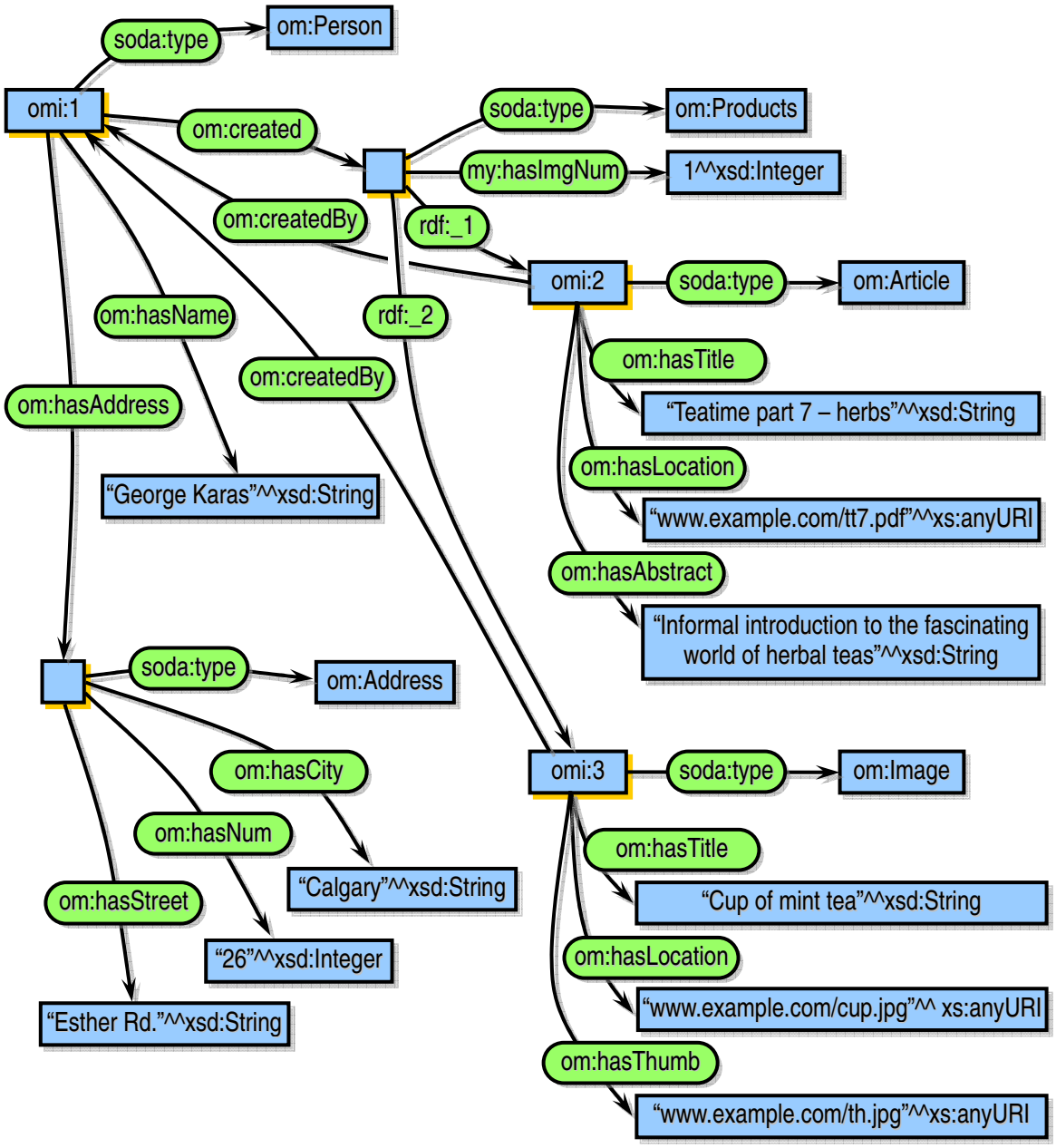


Figure 6.5 RDF database contents (instances) for the online magazine

6.4.5 Example of OM Database Contents

Figure 6.5 shows three objects in the OM database that correspond to the given schema — one person, an article and an image. The emphasized nodes of the graph show SODA concepts — two class instances, one collection and one tuple. They are all SODA typed and belong in the soda:Thing class, therefore they must have all properties prescribed by their type definition. Moreover, the values of these properties must be correctly typed. One important detail is that the collection of products is polymorphic and stores both article and image objects.

Another feature to notice is the new omi: namespace that serves for storing machine-generated urirefs for OIDs. These are automatically assigned to any class instances created by the user. Collection and tuple concepts are represented by blank nodes without unique identifiers.

Notice that from the RDF perspective, the pair of inverse relationships created and createdBy is redundant, since RDF triples can be read in both directions (indexed by subject and by object). On the other hand, in an object-oriented system they improve the performance and allow better traversability of the object graph.

6.4.6 Extending the Example

Now the model can be made to cooperate with other RDF vocabularies in a very simple yet powerful way through several subclassing declarations that connect the om concepts in global class hierarchies:

```
xmp:Nickname rdfs:subClassOf om:hasTitle .
xmp:Thumbnail rdfs:subClassOf om:hasThumb .
om:createdBy rdfs:subClassOf dc:creator .
om:hasTitle rdfs:subClassOf { dc:title, rss:title } .
om:hasAbstract rdfs:subClassOf { dc:description, rss:description } .
om:Product rdf:type rss:item .
om:hasLocation rdfs:subClassOf rss:link .
    where
    xmlns:rss="http://purl.org/rss/1.0/"
    xmlns:xmp="http://ns.adobe.com/xap/1.0/"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
```

If the maintainer of our example database decided to install a simple RDF reasoner and a RDF server, the above statements would result in opening the online magazine database to the following systems:

- **XMP – Extensible Metadata Protocol:** The hasTitle and hasThumb properties can now be imported from PDF or Photoshop files upon insertion in the database since Adobe XMP metadata is embedded in those files and the values of two different properties are identified by the OM system.

- **RSS – Remote Site Syndication:** RSS information is now automatically generated by every Product object in the database because it also becomes a RSS Item object. If the RDF server took the result, other sites could display brief descriptions and titles of the images and articles and other users could download the RSS feeds into their desktop readers to check for updates in their favorite magazine.

- **DC – Dublin Core Metadata:** By publishing Dublin Core information about the creator, title and description of products in the magazine, the whole database opens up to third party semantic searches and indexing, because it suddenly uses a widely accepted metadata protocol.

7 EXTENDING THE MODEL

7.1 MINING OBJECTS FROM RDF/S

In this context, mining objects means finding object-oriented structures in arbitrary RDF/S graphs, extracting objects from RDF graphs and storing them in an object database. This can be accomplished by either accepting the existing RDF/S schema, or building a new schema based on regularities encountered in the graph, or taking an existing OODB schema. This section focuses on the first approach and contains guidelines that are also useful in the other two approaches.

7.1.1 MOTIVATION

What are some reasons for mining objects from RDF data?

- **Structured processing** — object-oriented databases like Zope (http://www.zope.org/) contain application and Web servers so once the data is stored in such systems, it can be very easily extended with methods, published and manipulated. Relational databases can also structure data, but cannot take full advantage of RDF Schema information.

- **Object-oriented storage** — RDF graphs could be directly stored in OODB systems. Performance problems with storing objects in tables have been addressed by several approaches, like using sparse arrays (Caché object database).

- **Type checking** — once the object-oriented schema is defined, it is possible to automatically validate the required structure of arbitrary RDF data. Algorithms that process the result do not need to do so much type checking — this is similar to having a DTD for XML files.

- **Deductive extensions** — looking at the advantages from a database viewpoint, an object database that can access RDF data can use RDF entailments and therefore gains certain deductive features.

7.1.2 OVERVIEW OF EXTRACTED STRUCTURES

Most objects can be extracted automatically by imposing the SODA-interpretation on RDF data, converting RDF Schema statements into corresponding SODA statements and employing machine entailment but several minor adaptations of the RDF graph can extend this process and extract even more objects, although the resulting schema can be quite unstructured.

To extract object data from RDF, a list of basic object-oriented concepts and their RDF counterparts is needed. These concepts, corresponding to the ODMG standard [CB00] and G2 Concept Definition Language [HM00], are listed below.

- **Objects** — All data in an OODB are stored in objects. An object has a unique OID (uriref in RDF) and it contains a tuple or a collection of attributes or references.

  All RDF nodes with a uriref are considered objects.

- **Datatypes** — Most OODB models build their complex types on top of a set of elementary datatypes. These cannot be further decomposed and their semantics is fixed. This approach is equivalent to RDF T-interpretations.

  In both RDF and an object database, an instance of a datatype is bound to one type and its value is physically contained in the data node. Datatype nodes are converted to atomic values.

- **Attributes and relationships** — A collection or a tuple can either contain or reference a value. This is important for modeling, data storage and update semantics.

  When the value of a RDF attribute is a uriref node, it is referenced, and when its value is either a datatype or a blank node with a single reference to it, its value is embedded within the parent object. Blank nodes with multiple references are handled in a way similar to JDO Second Class Objects [Craig03] — an identical copy of all edges starting in this node is stored for every incoming edge.

- **Tuples** — A tuple is an elementary modeling concept; a structure with attributes labeled by property urirefs.

  Every blank RDF node that has at least one non-collection attribute can be considered a tuple. Adding a uriref to the node makes it a tuple object.

  In cases where the RDF graph contains several triples that assign different values to the same attribute of a node, an appropriate collection of the most general type is substituted for these attributes in the object-oriented schema.

- **Collections** — RDF has collection vocabulary without adequate formal semantic restrictions. In OODBs, collections are often typed and they are used to model extents or 1-to-N relationships.

  Any blank RDF node with collection attributes (eg. rdf:_1) can be considered a collection. A RDF collection with a uriref is reified to become a class instance with an embedded collection. RDF can have objects that act as both tuples and collections, similarly to G2 CDL [HM00], and the formal SODA model respects this.

- **Types** — In most OODBs, an object has exactly one type that specifies its internal structure.

  However, a RDF node can have no types or multiple types. Having no type is equivalent to having the most general type (soda:Thing) and having multiple types is similar to the concept of multiple inheritance and object roles. Nodes that conform to the structure prescribed by their types (through rdf:type, domain and range properties of object attributes) are labeled as "strongly typed" (soda:type) in the model and their class becomes a SODA concept.

- **Inheritance** — In both RDF and OODBs, inheritance is a fundamental tool for type hierarchies.

  The RDF definition of inheritance as a subset relationship on elements of the universe is suitable for its database counterpart, because it allows multiple inheritance and preserves the notion of "strongly typed" objects. SODA inheritance is applied to SODA concepts connected by the rdf:subClassOf attribute.

With these guidelines an object-oriented structure can be extracted from any RDF graph. Depending on the schema and organization of the graph, the format of the resulting data can be either quite loose

with many extra attributes or strongly typed — the SODA model allows both. Many custom RDF vocabularies are quite simple and place strong restrictions on their data, so most practical Semantic Web applications provide strictly structured RDF graphs suitable for OODB processing.

7.2 ACCESSING GRAPH DATA

7.2.1 ACCESSING THE RDF GRAPH

For the purposes of reasoning, a RDF graph is often presented as a set of facts (triples — binary predicates). When retrieving binary predicates for deductions, the most common request is to find a set of triples with a constant predicate — in Prolog syntax, this would be something like parent(X,oid1) or supervises(X,Y). This means that from the viewpoint of retrieving data, a RDF graph can be accessed randomly — the access point is *the label of an edge*, therefore the graph does not need to be connected and it is not important whether all its nodes are reachable. For representing knowledge bases in a database, this approach leads grouping the data by predicates.

While this is common from the deductive point of view, it presents an obstacle for viewing the RDF graph as an object database that usually has a different structure.

7.2.2 ACCESSING AN OBJECT DATABASE

Object database (without a query language) are accessed differently. In an OODB, information is physically grouped by objects. An object is accessed as a whole, and in contrast to the deductive approach, it is unusual to retrieve all occurrences of a given attribute. In Prolog syntax, accessing the whole object can be expressed as X(oid1,Y), which can be translated as "find all attributes X of object obj1 and read their values Y". The access point into the database graph is *the label of a node*. In a typical object database, only a limited number of nodes can be accessed directly and other nodes need to be reached by traversing the edges (references/attributes). Most common entry points are *extents* — for a given type, an extent is an object that stores a collection of its instances.

The practical result is that an object database is accessed from several nodes by traversing the edges of the object graph. In contrast to the Semantic Web approach, the direction of an edge is important and the whole graph must be reachable from its access points.

7.2.3 A COMMON ACCESS MODEL

To access the RDF graph as an object database, the whole graph needs to be reachable from certain access points. Figure 7.1 shows part of a RDF graph ("Nathaniel majors in philosophy"). To make such graph fully reachable, four changes to the original structure need to be made (three of the changes are emphasized and my: namespace contains user defined objects and attributes):

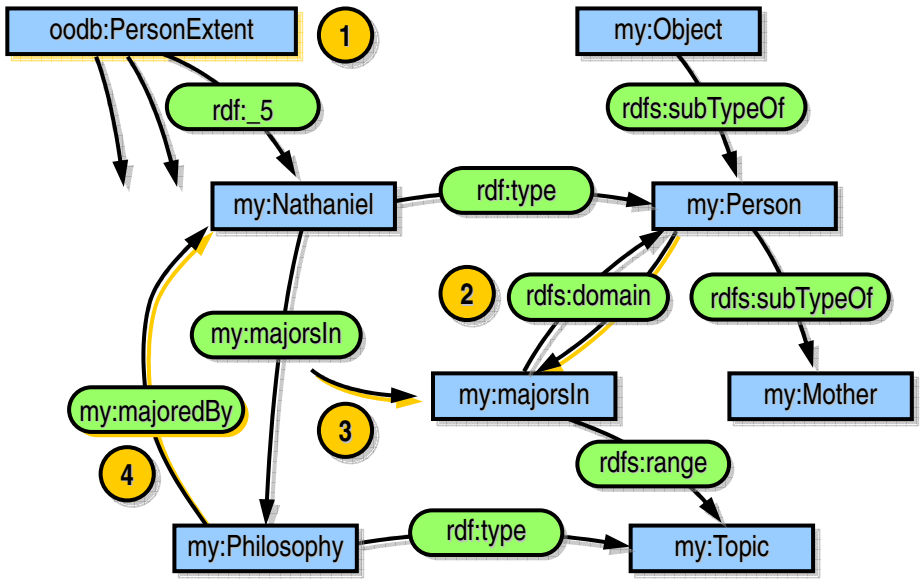


Figure 7.1 Reachability in part of a RDF graph

1. **Adding type extents.** In a typical OODB, every object needs to be part of at least one extent. For types in the database, extents are collections that are directly reachable from the system catalog (the only initial access point into the database), which in effect makes all the objects in the database accessible. In RDF setting, extent nodes could actually be identified with type nodes (oodb:PersonExtent and my:Person).

2. **Reversing the direction of rdfs:domain.** When finding out information about a certain type, it is useful to ask a question like X(my:Person,Y) to find out all the attributes of my:Person and their types together with subclassing information and other specifications of the class. For this reason, it would be useful to define a predicate that has the same meaning as rdfs:domain, but its object and subject are exchanged (see the curved arrow in Figure 7.1).

3. **Connecting edges to corresponding nodes.** In RDF, it is natural that an object can act both as a predicate and as subject/object. To find out information about an attribute in an object database, one can usually examine the type of an object that the attribute is connected to. However, the RDF model does allow extra object attributes unspecified by the class of the object, therefore it should be possible to find out about the specifics of an attribute elsewhere. In Figure 7.1, straight arrow indicates a new connection that needs to be made for the database to correctly traverse to the needed information — one practical way to do this is to create property extents that serve as access points to the property hierarchy.

4. **Inverse relationships.** With random access to any triple in the RDF graph, attributes and relationships can be traversed in both directions. However, in an object-oriented graph, a new inverse link needs to be added for every two-way relationship. On one hand, this adds extra triples, but on the other, it allows the clustering of triples by objects and lowers performance requirements for navigating the graph.

Limiting access to several entry points together with removing property-driven random access into the graph certainly limit the ease of random access to data in the RDF graph. On the other hand, data can be physically organized by objects and the usual OODB optimizations can be applied to things like attribute storage or type information. Accessing the RDF graph in a normal OODB way is made possible, which is useful for large RDF databases that need to retrieve whole RDF objects rather than process complex queries. The difference is similar to one between OLTP (online transaction processing) and OLAP (online analytical processing) architectures in relational databases.

7.3 IMPLEMENTATION NOTES

The proposed SODA model would obviously be very slow if built on top of existing RDF datasource implementations. Most of the existing implementations are programmed in high-level scripting languages and used for smaller-scale purposes rather than large data storage. However, RDF databases store triples in relational or postrelational databases which significantly boosts the performance. Thanks to the structure imposed on RDFS data, an implementation similar to a traditional object-oriented database could be chosen.

Numerous optimizations can be achieved for object-oriented RDF data — for example, the database can group data by objects and only store literal values instead of full triples; similarly, blank nodes of collection and tuple types can stored with class instances, forming in-memory records; assembling objects that inherit properties can be done without multiple joins on database tables; and some triples, such as soda:type or soda:_n become redundant thanks to the default organization of the database schema.

Why build on the semantic foundation of RDF instead of converting data to some other representation and storing them in an existing object-oriented database? The main advantages of having a sound RDF-compatible semantics are:

- **For the Semantic Web** — from the conceptual point of view, all of the data, their structure and semantics are RDF-compatible and therefore easily accessible within the Semantic Web by other computers, and ready for automatic processing by software agents.

- **For system flexibility** — RDF-based solutions are very flexible, which makes them ideal for prototyping, restructuring and further semantic extensions. It is often desirable to prefer flexibility to performance, and the SODA model is open to such modifications.

- **For object databases** — laying a solid theoretical foundation for an object-oriented data model is important because consensus has not yet been reached on the structure of OODB systems, and the Semantic Web offers a widely accepted standard with many object-oriented features.

8 CONCLUSION

The Semantic Web is an emerging paradigm for sharing semantically rich data on the Web and using them for cooperation among software agents. As the Semantic Web grows, it needs to address issues that have been traditionally researched in the database community — security, transaction management, efficient data storage, embedded business logic.

For many years, the area of object-oriented databases has lacked a unifying formal foundation. OODBs have been closely integrated with several object-oriented languages and independent of each other, and the ODMG effort at providing a common specification has only had partial success.

This thesis showed many similarities between the RDF-based Semantic Web and object-oriented data models. It highlighted some of the areas where these two can enrich each other — the area of OODB models and ontologies, and the area of formal specification of flexible RDF-based databases. A vocabulary extension for RDF was developed that structures data according to object-oriented database principles. This showed how to map the foundational OODB elements onto the world of the Semantic Web, and how to provide the Semantic Web with some important database concepts.

8.1 MAIN CONTRIBUTIONS

- The correspondence between a wide range of object-oriented data models and the Semantic Web RDF/S model was shown and analyzed.
- An object-oriented data definition language (similar to the G2 CDL language) based on RDF/S model theory was designed and formally described.
- Several extensions to the model were developed, such as the process of mining object data from arbitrary RDF/S graphs, and the modifications needed for providing reachability to RDF/S graphs based on access points.

Results of this research were presented to the international database and Semantic Web community at conferences in Canada [Güttner03c] (co-organized by Maebashi, Japan), Italy and Croatia [Güttner03b], and several ones in the Czech Republic — [GH02], [Güttner03] (won the Best PhD. Paper Award), [GH03], [Güttner04].

8.2 DIRECTIONS FOR FURTHER RESEARCH

There are many areas that open up for future investigation. Much research activity is currently aimed at RDF servers and databases (see section 4.4) and the object-oriented data model presented in this thesis could facilitate the implementation of an object-based RDF database. Some of the topics that could be addressed within the scope of the Semantic Web are:

- **Access Control** for portions of RDF graphs. This idea is discussed in a recent article ([Güttner04]) that shows the emerging need for access control based on RDF data, which returns the appropriate RDF subgraph. A simple access model was suggested but the issue needs further investigation.

- **RDF object query language**. OQL [CB00] and JDOQL [Craig03] are standard languages for querying object databases that have been extensively tested and used in a variety of applications. One interesting topic is exploring how these languages could be adapted to work with RDF data objects, and what limitations would need to be overcome.

- **Storage optimizations** gained by storing and subsequently navigating RDF data by objects. A fruitful topic would be to explore the tradeoffs of boosting navigation and storage capacity at the expense of limiting random access to triples — see section 7.2 for an introduction to the topic.

- **Active RDF databases** with embedded algorithms that work on RDF objects, dynamically generating RDF data are a counterpart of storing methods in object databases. Developing a formal model of such system is another direction for further research.

A REFERENCES[1]

[Adobe04] Adobe Systems Inc.: *XMP Specification.* http://www.adobe.com/products/xmp/pdfs/xmpspec.pdf, USA 2004.

[ACK01] Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., Tolle, K.: The ICS-FORTH RDFSuite: Managing Voluminous RDF Description Bases. In: *Proceedings of 2nd Int'l Workshop SemWeb'01 at WWW'01*, Hongkong 2001.

[Atkinson90] Atkinson, M. et al: The Object-Oriented Database Systems Manifesto. In: *Proceedings of DOOD '90 – Deductive and Object-Oriented Databases*, Elsevier Science Publishers, USA 1990.

[Atwood85] Atwood, T.: *An object-oriented DBMS for design support applications.* Ontologic, Inc. report, 1985.

[BKK87] Banerjee, J., Kim, W., Kim, K.C.: *Queries in object-oriented databases.* MCC Technical Report, no. DB 188-87, USA 1987.

[BBD00] Beged-Dov G., Brickley D., Dornfest R., et al.: *RDF Site Summary (RSS 1.0)*, http://purl.org/rss/1.0/spec, USA 2000.

[BvSvS87] Benesch, H., Von Saalfeld, H., Von Saalfeld, K.: *Encyclopedic Atlas of Psychology.* Deutscher Taschenbuch Verlag GmbH&Co, Germany 1987.

[BernersLee89] Berners-Lee, T.: *Information Management: A Proposal.* CERN, Switzerland 1989.

[BernersLee02] Berners-Lee, T.: *Primer: Getting into RDF & Semantic Web using N3.* http://www.w3.org/2000/10/swap/Primer, W3C, USA 2002.

[BHL00] Berners-Lee, T., Hendler, J., Lassila, O.: Semantic web. In: *Scientific American May 5/02*, USA 2000.

[BFM98] Berners-Lee, T., Fielding, R., Masinter, L.: *Uniform Resource Identifiers (URI): Generic Syntax (RFC 2396).* IETF Standard, http://www.ietf.org/rfc/rfc2396.txt, USA 1998.

[BM00] Biron, P.V., Malhotra, A. (eds.): *XML Schema Part 2: Datatypes.* W3C Recommendation, http://www.w3.org/TR/xmlschema-2/, USA 2000.

[BK95] Bonner, A.J., Kifer, M.: *Transaction Logic Programming (or a Logic of Declarative and Procedural Knowledge).* Technical Report CSRI-323, ftp://ftp.cs.toronto.edu/pub/bonner/papers/transaction.logic/iclp93.ps, University of Toronto, Canada 1995

[BEK00] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., Winer, D.: *Simple Object Acceess Protocol.* W3C Recommendation, http://www.w3.org/TR/SOAP/, USA 2000.

[BKvH02] Broekstra, J., Kampman, A., Van Harmelen, F.: Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In: *Proceedings of 1st Int'l Semantic Web Conference*, http://www.openrdf.org/doc/papers/Sesame-ISWC2002.pdf, Italy 2002.

[CdWV88] Carey, M., DeWitt, D., Vandenberg, S.: A data model and query language for Exodus. In: *Proceedings of the 1988 ACM SIGMOD conference,* ACM Press, USA 1988.

[1] All World Wide Web links in this thesis have been valid as of June 24, 2004.

[CDD03] Carroll, J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: *The Jena Semantic Web Platform: Architecture and Design*. HP Laboratories Technical Report HPL-2003-146, USA 2003.

[CastroLeon04] CastroLeon, E.: The Web Within the Web. In: *IEEE Spectrum* 2/41, IEEE Press, USA 2004.

[CB00] Cattel, R.D.D., Berry, D.K. (eds.): *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers San Francisco, USA 2000.

[CKW93] Chen, W., Kifer, M., Warren, D.S.: HiLog: A foundation for higher-order logic programming. In: *Journal of Logic Programming,* 15/3, USA 1993.

[CODASYL80] Committee on Data Systems and Languages, Data Base Task Group: *CODASYL Network Data Model*. CODASYL, USA 1980.

[Codd70] Codd, E.F.: A Relational Model of Data for Large Data Banks, In: *Communications of the ACM* 13(6), ACM Press, USA 1970.

[CvHH01] Connolly, D., Van Harmelen, F., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: *DAML+OIL Reference Description*. W3C Note, http://www.w3.org/TR/daml+oil-reference, USA 2001.

[CSS94] Costa, J., Sernadas, A., Sernadas, C.: Object Inheritance Beyond Subtyping. In: *Acta Informatica*, Germany 1994.

[Cowan02] Cowan, J.: Metadata, Reuters Health Information, and Cross-Media Publishing. In: *Proceedings of Seybold New York 2002 Enterprise Publishing Conference*. http://seminars.seyboldreports.com/seminars/2002_new_york/files/presentations/014/cowan_john.ppt, USA 2002.

[Craig03] Craig, R. (ed.): *Java Data Objects Specification 1.0.1*. http://jcp.org/aboutJava/communityprocess/final/jsr012/index2.html, Sun Microsystems, USA 2003.

[Dataquest99] Dataquest, a Gartner group company: *Dataquest market analysis*. http://www.intersystems.com/cache/analysts/reviews/dataquest.html, England 1999.

[DC94] Diskin, Z., Cadish, B.: *Algebraic Graph-Oriented = Category Theory Based. Manifesto of categorizing database theory*. Technical Report 9406 – Frame Information Systems, Latvia 1994.

[DC03] *Dublin Core Metadata Element Set, Version 1.1: Reference Description*. NISO Standard Z39.85-2001, http://dublincore.org/documents/2003/06/02/dces/, USA 2003.

[EXIF02] Technical Standardization Committee on AV & IT Storage Systems and Equipment: *Exchangeable image file format for digital still cameras: Exif Version 2.2*. http://www.exif.org/Exif2-2.PDF, JEITA Association, Japan 2002.

[FactSet00] FactSet Inc.: *The Vision Technology*. http://www.factset.com/files/whitepapers/vision_tech.pdf, FactSet White Paper, USA 2000.

[FBC87] Fishman, D., Beech, D., Cate, H.P. et al.: Iris: an object-oriented database management system. In: *ACM TOIS 5 (1)*, ACM Press, USA 1987.

[Fensel01] Fensel, D., *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce*. Springer Berlin, Germany 2001.

[FSM90] Fiadeiro, J., Sernadas, C., Maibaum, T., Saake, G.: Proof-theoretic semantics of object-oriented specification constructs. In: *Object-Oriented Databases: Analysis, Design and Construction,* North-Holland, USA 1990.

[FIPA02] Foundation for Intelligent Physical Agents: *FIPA Standard Specifications*, http://www.fipa.org/repository/standardspecs.html, Switzerland 2002.

[Goguen91] Goguen, J.: A Categorical Manifesto. In: *Mathematical Structures in Computer Science*, 1(1): 49–67, Spain 1991.

[GR83]] Goldberg, A., Robson, D.: *Smalltalk 80: The language and its implementation*. Addison–Wesley, USA 1983.

[GmC03] Guha, R., McCool, R.: *TAP: A Semantic Web Platform*. IBM Research, USA 2003.

[GO94] Gruber, T.R., Olsen, G.: An Ontology for Engineering Mathematics. In: *Proceedings of 4th Int'l Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kaufmann, Germany 1994.

[Güttner03] Güttner, J.: Object Attributes As Functions. In: *Proceedings of 6th International Conference ISIM '03*, MARQ Ostrava, Czech Republic 2003.

[Güttner03b] Güttner, J.: Objects in the Semantic Web. In: *SoftCOM 2003 – International Conference on Software, Telecommunications and Computer Networks*, FESB Split, Croatia 2003.

[Güttner03c] Güttner, J.: Object Database on Top of the Semantic Web. In: *Proceedings of the WI/IAT 2003 Workshop on Applications, Products and Services of Web-based Support systems*, Halifax, Canada 2003.

[Güttner04] Güttner, J.: RDF Access Control — an Object Database Viewpoint. In: *Proceedings of Znalosti '04*, MU Brno, Czech Republic 2004.

[GH02] Güttner, J., Hruška, T.: Dynamic Object Model In Interpreted Systems. In: *Proceedings od 5th International Conference ISM 02 – Information System Modelling*, MARQ Ostrava, Czech Republic 2002.

[GH03] Güttner, J., Hruška, T.: Semantic web as a flexible database. In: *Datakon 2003 – Proceedings of the Annual Database Conference*, MU Brno, Czech Republic 2003.

[Hayes04] Hayes, P.: *RDF Semantics*. W3C Recommendation, http://www.w3.org/TR/rdf-mt/, USA 2004.

[HM00] Hruška, T., Máčel, M.: Object-Oriented Database System G2. In: *Proceedings of the Fourth Joint Conference on Knowledge-Based Software Engineering*, IOS Press Brno & Ohmsha Publishing, Czech Republic 2000.

[IDC03] IDC White Paper: *Breaking the Relational Barrier: User Data Management Triumphs with Intersystems' Caché*. IDC, USA 2003.

[Jordan01] Jordan, D.: The JDO Object Model, In: *Java Report* 6/2001, http://www.objectidentity.com/images/jdo_model82.pdf, USA 2001.

[Karger04] Karger, D.: Haystack — Per User Information Environments. In: *Proceedings of 3rd Int'l Semantic Web Conference*, Hungary 2004.

[KCA02] Karvounarakis, G., Christophides, V., Alexaki, S., Plexousakis, D., Scholl, M.: RQL: A Declarative Query Language for RDF. In: *Proceedings of 1st Int'l Semantic Web Conference*, http://athena.ics.forth.gr:9090/RDF/publications/www2002/www2002.pdf, Italy 2002.

[Kifer95] Kifer, M.: Deductive and Object Languages: Quest for Integration. In: *Proceedings of DOOD '95 – Deductive and Object–Oriented Databases,* Springer, Singapore 1995.

[KLW95] Kifer, M., Lausen, G., Wu, J.: Logical Foundations of Object-Oriented and Frame-Based Languages. In: *Journal of the ACM*, ACM Press, USA 1995.

[KC04] Klyne, G., Carroll, J.J., (eds.): *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C Recommendation, http://www.w3.org/TR/rdf-concepts/, USA 2004.

[Kolenčík98] Kolenčík, P.: *Categorical Framework for Object-Oriented Database Model*. PhD. Thesis, Brno University of Technology, Czech Republic 1998.

[LRV92] Lécluse, C., Richard, P., Vélez, F.: O2, an Object-Oriented Data Model. In: *Building an Object–Oriented Database System: The Story of O2*, Morgan Kaufmann Publishers, USA 1992.

[MOP86] Maier, D., Otis, A., Purdy, A.: Development of an object-oriented DBMS. In: *Quarterly Bulletin of IEEE on Database Engineering 8(4),* IEEE Press, USA 1986

[MM04] Manola, F., Miller, E., (eds.): *RDF Primer*. W3C Recommendation, http://www.w3.org/TR/rdf-primer/, USA 2004.

[MRB04] Miles, A.J., Rogers, N., Beckett, D.: *SKOS-Core 1.0 Guide*. SWAD-Europe, EU 2004.

[NR95] Nelson, D., Rossiter, B.: Prototyping a categorical database in P/FDM. In: *Second Int'l Workshop on Advances in Databases and Information Systems ADBIS'95*, Slovakia 1995.

[OMG03] Object Management Group: *Unified Modeling Language (UML), Version 1.5*. http://www.omg.org/technology/documents/formal/uml.htm, OMG, USA 2003.

[PHH04] Patel-Schneider, P.F., Hayes, P., Horrocks, I. (eds.): *OWL Web Ontology Language Semantics and Abstract Syntax*. W3C Recommendation, http://www.w3.org/TR/owl-semantics/, USA 2004.

[PRISM02] PRISM Working Group: *Publishing Requirements for Industry Standard Metadata 1.2*. http://www.prismstandard.org/Pam_1.0/PRISM_1.2h.pdf, IDEAlliance, USA 2003.

[Rohner04] Rohner, R.: Commercializing RDF: Semantic Software Solutions for Enterprise Web Management. In: *Proceedings of 3rd Int'l Semantic Web Conference*, Hungary 2004.

[SSW94] Sagonas, K., Swift, T., Warren, D.S.: XSB as an Efficient Deductive Database Engine. In: *Proceedings of ACM SIGMOD Conference on Data Management*, ACM Press, USA 1994.

[Schewe95] Schewe, K.-D.: *Fundamentals of Object Oriented Database Modeling*. Clausthal Technical University, Germany 1995.

[Seaborne02] Seaborne, A.: A Programmer's Introduction to RDQL. In: *Proceedings of First Int'l Semantic Web Conference*, http://www.hpl.hp.com/semweb/iswc2002/JenaTutorial.Alpha/RDQL/TutorialRDQL.html, Italy 2002.

[SA00] Shaver, M., Ang, M.: *Inside the Lizard: A Look at the Mozilla Technology and Architecture*. USA 2000.

[Stroustrup86] Stroustrup, B.: *The C++ programming language*. Addison–Wesley, USA 1986.

[Sutherland93] Sutherland, J.: *Dialogue on Objects and SQL3*, Meeting Notes X3H7-93-057, USA 1993.

[ŠSK04] Šváb, O., Svátek, V., Kavalec, M., Labský, M.: Querying the RDF: Small Case Study in the Bicycle Sale Domain. In: *Proceedings of the Dateso 2004 Workshop*. http://www.cs.vsb.cz/dateso/2004/, Vydavatelství Univerzity Palackého Olomouc, Czech Republic 2004.

[Tuijn94] Tuijn, C.: *Data Modeling from a Categorical Perspective*. PhD thesis, Antwerpen University, Netherlands 1994.

[vEchelpoel02] Van Echelpoel, K.: *Java Data Objects: A Revolution in Java Databanking?* Final thesis, http://www.vanechelpoel.be/thesis/, Universiteit Antwerpen, Netherlands 2002.

[WSK03] Wilkinson, K., Sayers, C., Kuno, H., Reynolds, D.: Efficient RDF Storage and Retrieval in Jena2. In: *Proceedings of 1st Int'l Workshop on Semantic Web and Databases*, Germany 2003.

[YK00] Yang, G., Kifer, M.: *FLORA: Implementing an Efficient DOOD System Using a Tabling Logic Engine*. Dept. of Computer Science, SUNY at Stony Brook, USA 2000.

AUTHOR'S OWN PUBLICATIONS[1]

[1] Güttner, J.: RDF Access Control — an Object Database Viewpoint. In: *Proceedings of the Znalosti '04 Conference*, Ostrava, CZ, VSB-TUO, 2004, ISBN 80-248-0456-5

[2] Güttner Jakub, Hruška Tomáš: Semantic web as a flexible database, In: *Datakon 2003 - Proceedings of the Annual Database Conference*, Brno, CZ, MU Press, 2003, p. 217-226, ISBN 80-210-3215-4

[3] Güttner Jakub: Can a Computer Think? (A Hypothetical Discussion between L. Wittgenstein and A. Turing), In: *Pokroky matematiky, fyziky a astronomie*, Vol. 2003, No. 48, CZ, p. 105-114, ISSN 0032-2423

[4] Güttner Jakub: Object Attributes As Functions, In: *Proceedings of 6th International Conference ISIM '03*, Ostrava, CZ, MARQ, 2003, p. 169-177, ISBN 80-85988-84-4

[5] Güttner Jakub: Object Database on Top of the Semantic Web, In: *Proceedings of the WI/IAT 2003 Workshop on Applications, Products and Services of Web-based Support systems*, Halifax, CA, 2003, p. 97-102, ISBN 0-9734039-1-8

[6] Güttner Jakub: Objects in the Semantic Web, In: *SoftCOM 2003 - International Conference on Software, Telecommunications and Computer Networks*, Split, HR, FESB, 2003, p. 19-23, ISBN 953-6114-64-X

[7] Güttner Jakub, Hruška Tomáš: Dynamic Object Model In Interpreted Systems, In: *Proceedings od 5th International Conference ISM 02 - Information System Modeling*, Ostrava, CZ, MARQ, 2002, p. 103-110, ISBN 80-85988-70-4

[1] sorted in inverse chronological order

B OTHER SEMANTIC WEB TOOLS AND APPLICATIONS

B.1 LSID IN BIOINFORMATICS

Structured metadata using controlled vocabularies play an important role in modern medicine, enabling efficient literature searches and aiding in the distribution and exchange of medical knowledge.

Until recently, many bioinformatics databases had proprietary access protocols, data formats, and naming conventions for objects and their relationships. Users had to use many different tools for a number of sites, which always gave them partial information about the topic of interest. Today, RDF-compatible LSID (Life Sciences Identifiers) with SOAP-enabled servers running on LSID protocol have become the standard in most public bioinformatics databases — they provide RDF data either directly or through proxy LSID authorities. Some providers of such data include GenBank, Gene Ontology or PDB [Cowan02]. This allows seamless integration of sources in projects like BioHaystack [Karger04].

> One example of LSID identifiers is urn:lsid:ncbi.nlm.nih.gov.lsid.i3c.org:genbank:nm_001240

Gene Ontology Consortium (http://www.geneontology.org/), for example, provides its own controlled vocabularies to describe specific aspects of gene products. Collaborating databases annotate their gene products with terms in RDF/XML. Genes are annotated by molecular function, biological process, and cellular component. These annotations are directed acyclic graphs of inheritance and aggregation relationships with semantics similar to RDF Schema. Every statement in the database also needs to be backed up by evidence information, such as traceable author statement, direct result of a publication, or automatically inferred fact. All of this information is then connected to LSIDs and published.

> An excerpt from a Gene Ontology file describes how two gene products are associated:

```
<go:term rdf:about="http://www.geneontology.org/go#GO:0016209">
    <go:accession>GO:0016209</go:accession>
    <go:name>antioxidant</go:name>
    <go:isa rdf:resource="http://www.geneontology.org/go#GO:0003674" />
    <go:association>
        <go:evidence evidence_code="ISS">
            <go:dbxref>
                <go:database_symbol>fb</go:database_symbol>
                <go:reference>fbrf0105495</go:reference>
            </go:dbxref>
        </go:evidence>
        <go:gene_product>
            <go:name>CG7217</go:name>
            <go:dbxref>
                <go:database_symbol>fb</go:database_symbol>
```

```
                <go:reference>FBgn0038570</go:reference>
            </go:dbxref>
        </go:gene_product>
    </go:association>
</go:term>
```

B.2 SESAME

Sesame [MKvH02] is an architecture that allows the storage and querying of large amounts of RDF and RDF Schema data. It was developed within the joint European On-to-knowledge project and the whole project is available for downloading by noncommercial users.

- **Storage** layer of data in Sesame is based on using multiple stackable Repository Abstraction Layers with custom API. In newer versions of Sesame, these were extended to SAILs — Storage and Inference Layers. These can interface the Sesame core with database management systems such as PostgreSQL, RDF files, specialized RDF stores, or RDF network services.

- **Administration** module takes care of incrementally adding RDF/S information by statements. As triples are added, some basic RDF Schema entailments are made based on types, subclassing information and property domains/ranges, and checked against the consistency of the data store.

- **Export** module simply exports schema or data in RDF/XML.

- **Query** module interfaces with the user through the SeRQL query language. For the PostrgreSQL database, queries are optimized as they are translated to SQL. Further performance improvements come from the storage structure that organizes the relational tables by RDF Schema information.

- **Interface** layer makes the modules accessible through HTTP, SOAP and RMI protocols.

B.3 HAYSTACK

Haystack [Karger04] is a tool intended for casual users, developed by MIT in cooperation with IBM. It addresses the fact that people know a lot that they are willing to share, but too lazy to publish. The environment based on IBM's open-source Eclipse IDE gathers that knowledge in RDF form for exchange and analysis without interfering with the user. This leads to support for intelligent searching, mail filtering, automatic categorization and context-sensitive support for user tasks.

Activities supported by Haystack include text processing, e-mail management, calendar and planner, Internet browsing, searching and bookmarking, content annotation and categorization. The whole environment heavily relies on context and tries to present the user only with information that is relevant to the task at hand. All information is decoupled from physical location and graphical representation and is displayed when needed through persistent *views* that can be customized and even exchanged among users.

Apart from personal management interfaces, another existing Haystack application is BioHaystack that integrates LSID data from various bioinformatics databases and lets the user annotate, discuss and analyze their content.

B.4 SKOS AND SWAD-EUROPE

SKOS (Simple Knowledge Organization System, http://www.w3.org/2004/02/skos/core.rdf) is one of the activities of SWAD-Europe (Semantic Web Advanced Development, http://www.w3.org/2001/sw/Europe/), EU IST-supported project and part of the W3C Semantic Web Activity. The SKOS-Core specification is intended for connecting various thesaurus activities and the Semantic Web.

SKOS provides a basic framework for building concept schemes, but it does not carry the strictly defined semantics of OWL, which makes it ideal for representing those types of knowledge organization systems, such as thesauri, that cannot be mapped directly to an OWL ontology. SKOS is also easier to use, and harder to misuse than OWL, providing an ideal entry point for those wishing to use the Semantic Web for knowledge organization [MRB04].

SKOS-Core contains a RDF/S vocabulary for linking *concepts* to the *words and phrases* (labels) used by people to refer to them — this includes support for definitions, scope notes and examples; several degrees of subsumption (broader/narrower, generic, instantive and partitive relationships), relatedness (related, PART-OF related); high-level categories and internationalization. Other parts of the activity involve work integrating existing RDF solutions and providing frameworks for multilingual thesauri and inter-thesaurus mappings.

B.5 PRISM

PRISM (Publishing Requirements for Industry Standard Metadata, [PRISM02]) was developed by a working group in the publishing industry. Its goal is to specify metadata for reusing existing content in a variety of situations, including online versions of magazine articles, licensing to aggregators, retrospective articles, book compilations or licensing to outsiders. In order to meet these needs, the standard focuses on: *discovery* of content within the industry, which includes metadata about multiple subject taxonomies, topics, origins and context; *rights tracking* such as needed for stock photo agencies which defines issues like royalty payments or limitations on use in different industries; and *end-to-end metadata* that can carry over multiple stages of the production pipeline without being lost in processing the files in question.

RDF was chosen as a carrier because DTDs for plain XML cannot adequately describe the many possibilities of defining PRISM metadata (from plain text attributes to structured vocabularies). Apart from using controlled vocabulary such as Dublin Core or ISO standards, PRISM also defines three new RDF vocabularies:

- **PRISM core vocabulary** (PRISM) subclasses several Dublin Core terms and gives them a more specific meaning — an example is dc:date, which breaks down to prism:publicationTime, prism:releaseTime or prism:expirationTime.

- **PRISM Rights Language** (PRL) is an interim standard in an environment struggling to come up with an agreement in the area of digital rights management. It helps encode basic information about how content may or may not be used depending on information about industry, time and place.

- **PRISM Controlled Vocabulary** (PCV) helps build shared lists of controlled keywords. It assists with defining terms, relations between them and alternate names.

> This example says that the image (Nat.jpg) cannot be used (#none) in the tobacco industry (code 21 in SIC, the Standard Industrial Classifications):

```
<rdf:RDF    xmlns:prl="http://prismstandard.org/namespaces/prl/1.0/"
            xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:dc="http://purl.org/dc/elements/1.1/">
    <rdf:Description rdf:about="http://my.pictures.com/2004/01/Nat.jpg">
        <dc:rights rdf:parseType="Resource"
                xml:base="http://prismstandard.org/vocabularies/1.0/usage.xml">
            <prl:usage rdf:resource="#none"/>
            <prl:industry rdf:resource="http://prismstandard.org/vocabs/SIC/21"/>
        </dc:rights>
    </rdf:Description>
</rdf:RDF>
```

C A SURVEY OF RDF QUERY LANGUAGES[1]

C.1 GETDATA

GetData[2] is a very simple query language intended to provide basic RDF retrieval with as little load on the server as possible, making it useful for large-scale deployment. It is also very easy to implement, although its capabilities are extremely limited. Given the object (or subject) and predicate of a given triple, it provides its subject (or object). For data exploration, a query can also return all triples with a given subject.

C.2 RQL

RQL [KCA02] was first proposed as part of RDF Suite [ACK01]. It is a declarative language that works with RDF and RDF Schema. Its basic building blocks are *functions* that query RDF Schema (such as Class, Property, domain and range) and *path expressions*. RQL contains a SELECT–FROM–WHERE statement that differs from its SQL counterpart in several aspects. The SELECT part gives a list of variables that are returned as a result. FROM specifies a filter in form of path expressions that allows querying a RDF subgraph, and WHERE gives additional conditions such as comparison of literals, variables and results of embedding queries; these can be composed using logical connectives.

> This query returns the first names of all Person objects with surname that starts with "D":
>
> ```
> SELECT first
> FROM {X : my:Person} my:hasSurname {surname},
>     {X} bike:hasFirstName {first} .
> WHERE name like "D*"
> ```

[1] this overview was inspired by [ŠSK04]

[2] see section 4.4.4 for a description of TAP that also mentions GetData as one of its components

C.3 SeRQL

SeRQL[1] [MKvH02] used in the Sesame framework is a declarative language similar to RQL, but with several important extensions. First, it supports the CONSTRUCT statement that is identical to SELECT but it returns a portion of RDF graph because a structure of triples replaces the variable list. Second, schema information that cannot be obtained from the RDF graph due to automatic entailments is supported by predicates like serql:directSubClassOf or serql:directType. Third, literals in the WHERE clause must be assigned XSD types; and fourth, SeRQL supports optional path expressions that may or may not return results depending on whether they are found.

This query returns triples of names and possible titles of all PhD students called "Smith":

```
CONSTRUCT {name} <rdf:type> {<my:Smith>}, {name} <my:studentTitle> {title}
FROM {X} <serql:directType> {<my:PhDStudent>};
     <my:hasSurname> {surname};
     [ <my:hasTitle> {title} <rdf:type> {<my:EnumTitles>} ];
     <my:hasFirstName> {name},
WHERE surname = "Smith"^^<xsd:String>
```

C.4 RDQL

RDQL was originally developed for the Jena [CDD03] tool. It works with RDF Schema, and supports path expressions. It also supports the SELECT statement, but the FROM part is missing; the subgraph filter is given in WHERE clause as a list of triples, and partial paths can be bound together by variables. RDQL also supports an AND part of Select, where the results are filtered by value through comparisons and logical connectives.

The following query returns information about all adults called "Smith".

```
SELECT ?person, ?name, ?surname, ?age
WHERE (?person, <rdf:type>, <my:Person>) ,
     (?person, <my:hasName>, ?name) ,
     (?person, <my:hasSurname>, ?surname) ,
     (?picture, <pict:hasAge>, ?age)
AND ( ?age > 18 && ?surname eq "Smith")
```

C.5 PerlRDF Queries

PerlRDF Suite was developed by Ginger Alliance (http://www.gingerall.com/), and contains an array of tools for the Perl programming language, including a query processor. Again, the SELECT–FROM–WHERE statement is available with support for namespace use, functions, path expressions, comparisons and logical connectives.

[1] Sesame Query Language, pronounce as "circle"

This query retrieves names and surnames of Person objects. It demonstrates several ways of returning values, using class membership (::) and namespace with or without abbreviations:

```
SELECT ?person, ?name, ?person -> [http://www.example.com/people#hasSurname]
FROM my:Person::?person -> my:hasName { ?name },
     ?person -> bike:hasAge { ?age }
WHERE ?person -> [http://www.example.com/people#hasAge] > '18'
```

D LIST OF FIGURES

E GLOSSARY OF ABBREVIATIONS

CDL	Concept Definition Language, language that describes schemas in the G2 object database
CT	Category Theory, a very general and abstract formalism for declarative modeling
DC	Dublin Core Metadata, a standard for specifying basic metadata about various resources
DTD	Document Type Definition, a context-free grammar based definition format for XML files
JDO	Java Data Objects, Java standard for object-oriented and object-relational database models
JDOQL	Java Data Objects Query Language, part of JDO specification, a language similar to OQL
ODMG	Object Data Management Group, a standard for object-oriented models of persistent data
OID	Object Identifier, mandatory unique identification of objects in object-oriented databases
OODB	Object-Oriented Database, a database that stores objects connected by relationships
OQL	Object Query Language, SQL-like query language specification in the ODMG standard
OWL	Ontology Web Language, a high-level ontology language (now a W3C recommendation)
P-C	Persistence Capable classes, classes in the JDO framework that are stored in the database
RDF	Resource Description Framework, a W3C standard for structuring Semantic Web data
RDFS	RDF Schema, a lightweight ontological standard for further structuring RDF graphs
RSS	Remote Site Syndication 1.0 is a popular RDF-based format for syndicating online content
SemWeb	Semantic Web, a W3C standard for next-generation Web with computer-readable semantics
SODA	Semantic Object-Oriented Database, model presented in this thesis, based on CDL+RDF
URI	Universal Resource Identifier, a widely used standard for assigning global identifiers
uriref	URI reference, used as a unique object identifier (OID)
XML	EXtensible Markup Language, a standard for exchanging syntactically structured data
XMP	Extensible Metadata Protocol, an Adobe standard for embedding RDF metadata in files
XSD	XML Schema Datatypes, a standard for typing XML data and forming new datatypes
W3C	World Wide Web Consortium, an organization that develops Semantic Web standards