A Layered Approach to Information Modeling and Interoperability on the Web

Sergey Melnik * **and Stefan Decker**

Database Group, Stanford University
{melnik,stefan}db.stanford.edu

Revised version: Sep 4, 2000

Abstract. On the Semantic Web, the target audience is the machines rather than humans. To satisfy the demands of this audience, information needs to be available in machine-processable form rather than as unstructured text. A variety of information models like RDF or UML are available to fulfill this purpose, varying greatly in their capabilities. The advent of XML leveraged a promising consensus on the encoding syntax for machine-processable information. However, interoperating between different information models on a syntactic level proved to be a laborious task. In this paper, we suggest a layered approach to interoperability of information models that borrows from layered software structuring techniques used in today's internetworking. We identify the object layer that fills the gap between the syntax and semantic layers and examine it in detail. We suggest the key features of the object layer like identity and binary relationships, basic typing, reification, ordering, and n-ary relationships. Finally, we examine design issues and implementation alternatives involved in building the object layer.

1 Introduction

A wealth of novel Web applications are aimed at providing integrated access to data and enabling comprehensive Web services. Examples of such applications include data portals and warehouses, mediation and e-business services. For many of these applications, data interoperation is essential. For example, an information portal needs to integrate data from different sources, whereas business-to-business communication requires bridging differences between incompatible business documents like purchase orders and invoices. Unfortunately, Web data sources and services deploy a variety of information models and data encoding syntaxes. Hence, exchange of information between Web applications is limited and requires expensive engineering.

Factors like design independence, competition, purpose-tailoring, and the increasing installed base of software that uses diverse data models suggest that a variety of alternative information models will always be around. Although increasing use of XML has simplified data exchange, the problem of information interoperability remains largely unresolved. For the same kind of data, independent developers often design XML syntaxes that have very little in common. For example, `biztalk.org` lists a number of XML schemas used to encode purchase orders. In the schema by LCS International Inc., the issue date of a purchase order is specified as:

```
<PurchaseOrder>
  <orderDate>...</orderDate>
</PurchaseOrder>
```

The encoding chosen by the Open Application Group (OAG) looks rather like

```
<PROCESS_PO_004>
  <DATAAREA>
    <PROCESS_PO>
       <POORDERHDR>
          <DATETIME>...</DATETIME>
       </POORDERHDR>
    </PROCESS_PO>
  </DATAAREA>
</PROCESS_PO_004>
```

whereas the schema by the NxTrend Technology Inc. requires yet another incompatible format:

* *Permanent address:* Database Group, Leipzig University, Germany

```
<PurchaseOrder>
  <OrderHeader>
    <POIssueDate>...</POIssueDate>
  </OrderHeader>
</PurchaseOrder>
```

Current approaches to interoperation

Enabling interoperation between say the Noris ERP system used by LCS Inc. and an OAGIS-compliant system is a laborious task. Multiple strategies are used for enabling data interoperability. One possible solution is translating directly between two different kinds of XML schemas. Such translation can be done, for example, using a declarative language like XSLT. A serious obstacle for this approach is that a mapping between two XML representations needs to be carefully specified by a human expert. In the example, the expert needs to understand both the encoding of LCS Inc. and OAG schemas and the semantics of the schema elements. Since the representations can be very diverse, the mappings created by the expert are often complex. This complexity makes them hard to maintain when the original data schemas change and hinders efficient execution of mappings.

An alternative strategy that is used for reconciling XML data is based on intermediate conceptual models [DMvH+00]. In this case, a human expert is needed to reverse-engineer the underlying conceptual model for every XML schema, and to specify formally how the original schema maps onto the corresponding conceptual model. After this step, the differences between conceptual models can typically be bridged with less effort. Although more elegant, this approach has, however, difficulties comparable to the first one. That is, intervention of a human expert is required, and the mappings need to be maintained.

Today's information exchange resembles a group of people communicating by means of encrypted data without disclosing the keys needed to decipher it. A way of reducing the tremendous effort needed for data interoperation is to supply metadata needed to interpret the exchanged information. However, the semantics of XML elements used by Web applications is hard-coded into the applications and is typically not available in machine-processable form. In fact, explicit and comprehensive encoding of metadata is prohibitive for all but rare application scenarios. It is not even clear how much metadata is sufficient, and how it should be encoded. Thus, establishing interoperation is a complex task, with many special-case solutions. Solving the interoperability problem on a broad scale requires novel techniques.

Computer networks and data models

In this paper we introduce some initial ideas targeted at facilitating data interoperation using a layered approach. Our approach resembles the layering principles used in internetworking. Consider a client application that wants to establish a reliable connection with a server application separated by multiple heterogeneous networks. If attacked directly, this problem is virtually unfeasible. Even in case of a direct physical connection between the client and the server, both sides have to be prepared to deal with different protocols, addressing schemas, packet sizes, error handling, flow control, congestion control, quality of service etc. The amount of control information and mutual agreement needed to deliver and correctly interpret the bits across multiple networks is enormous. To reduce its design complexity, most network software is organized as a series of layers. The purpose of each layer is to offer certain services to the higher layers, shielding those layers from the details of how the offered services are implemented. Internetworking is achieved by a common understanding of protocols. The transmitted data, flowing through the different layers, is enriched with control information necessary for the correct interpretation of the data in a particular layer. From a data perspective, this control information can be viewed as metadata used for routing the original data package in the desired way through the network and interpreting the data by the recipient.

Analogously, if attacked directly, data exchange requires tremendous effort. Any two applications have to be prepared to deal with various encoding syntaxes, different ways of representing objects, ordered and n-ary relationships, aggregation, specialization, cardinality constraints, ontology languages etc. To harness the complexity of data interoperation, we suggest a software structuring technique similar to that used in internetworking. We noticed that existing and emerging data models also tend to be organized in a layered fashion. For example, the Resource Description Framework [LS98] uses XML as its serialization syntax. RDF itself can be deployed as an object model for carrying UML data [Mel00]. Finally, UML is used as a basis for the Open Information Model (OIM) developed by the Meta Data Coalition. Currently, such *ad hoc* layering approaches lack well-defined separation between data modeling layers, have redundant features on different layers, and are generally characterized by a very low granularity of layering. We believe that clean separation between the layers can significantly improve data interoperation and facilitate applicability of well-established internetworking principles like bridges and gateways.

In this paper we focus on the question how to identify and distinguish data modeling layers from each other. We start with an analysis of existing data modeling languages and try to extract modeling primitives used in those languages. We roughly organize these modeling primitives into three major layers, the syntax, object and semantic layer, and examine the object layer in more detail. We do not claim that the resulting organization is perfect and definitive. Rather, it is a first incremental

step in our effort to build a comprehensive data interoperation architecture.

The next section describes our layered reference model, which we call *Information Model Interoperability* (IMI) Reference Model. We highlight the design choices that need to be made in providing a clean separation between the data modeling layers and sublayers. In Sect. 3 we review some popular data models used for data exchange on the Web and justify our design choices. In the rest of the paper we focus on the features and implementation of the object layer.

2 Information Model Interoperability (IMI) Reference Model

To illustrate the principles of layered data modeling, let us briefly review the fundamental concepts of network layers. Imagine two network applications that exchange data using the TCP/IP protocol (see Fig. 1). A protocol like TCP or IP is an agreement between the communicating parties on how communication has to proceed. The entities comprising the corresponding layers on different machines are called peers. No data is directly transferred between the peers. Instead, each layer passes data and metadata (header or control information) to the layer immediately below it, until the lowest layer is reached.

In the figure, the application layer passes the data to the transport layer. The transport layer arranges the data into segments and appends a TCP header to each segment. The header contains metadata about the segment like its sequence and acknowledgement number, source and destination port etc. The TCP header and data are passed further down to the network layer. The network layer arranges segments into packets and appends its own headers to them. Finally, the data link layer sends the data over a physical medium as a sequence of bits, preceded by a frame header. The frame header contains, for instance, a delimiter, number of bits and the checksum of the frame. This information is required to identify frames in a bit sequence. On the other side of the wire, the process is reversed. Each layer receives data from the layer below it, and evaluates the header containing information on how to interpret the data field.

Every pair of adjacent layers exchange information using an interface. The interface defines which primitive operations and services the lower layer offers to the upper one. Clean-cut interfaces make it simpler to replace the implementation of one layer with a completely different implementation and minimize the amount of information that must be passed between layers. In our example, the data link layer may equally well be implemented using the Ethernet or the Token Ring protocol. This design choice does not affect the upper layers. It is not even necessary that the interfaces on all machines in a network be the same, provided that each machine can correctly use all the protocols. For instance, a UNIX application may use Berkeley sockets, whereas a Windows application may use the Winsock library. The only important issue for a successful multilayer communication is an agreement on the protocol stack, i.e. which kind of protocol is used on every layer.

Now, let us turn to data modeling. Imagine two applications that need to exchange complex data. Instead of forcing every application to deal directly with the details of semantics, structure and serialization of data, we can organize the data exchange software in a layered fashion, similarly to the approach taken in the internetworking. Consider a sample set of layers depicted in Fig. 2. In the application layer, the data may be accessed using high-level primitives like "`employee.setEmployer(boss)`". As in the networking architecture, the data is not exchanged directly between two peer layers. Instead, every layer appends metadata needed to correctly interpret the data, and passes them to the layer below it.

In the example depicted in the figure, the semantic layer creates an object graph representing the entities and their relationships. It appends to the object graph the metadata needed to determine which ontology languages and ontologies are used, how they are implemented, how cardinality, aggregation etc. are expressed, and passes both the data and metadata to the object layer. The metadata appended at the object layer describes e.g. how ordered relationships or n-ary relationships are implemented, or how typing of nodes is represented in the object graph. This information is forwarded to the syntax layer, which generates, for example, an XML document containing the object graph, and appends to it an XML schema needed to extract the object graph from the document.

On the other end of the communication link, the process is reversed and a high-level data structure is delivered to the application layer. Similarly to internetworking, every data modeling layer relies on a number of rules and conventions to exchange information with its peer, just as two corresponding networking layers deploy a specific communication protocol. For example, the syntax layer may require that the metadata be represented using the XML Schema standard. We call a list of such "protocols" (sets of conventions) a *model stack*. In our reference model, every data modeling language like UML, RDF etc. can be viewed as a specific model stack.

Obviously, implementing layered data interoperation has a number of challenges. Often a clean-cut distinction between layers or features is not possible. Furthermore, mutual dependencies between the layers may exist. For example, n-ary relationships may be implemented using ordered relationships, or the other way around. Further challenges include the choice of flexible and powerful APIs for every layer, building data gateways for existing data models etc.

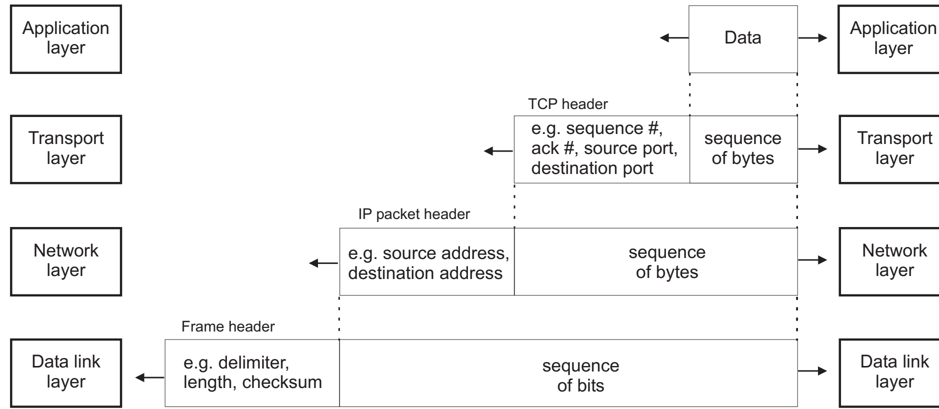

Fig. 1. An example of networking layers

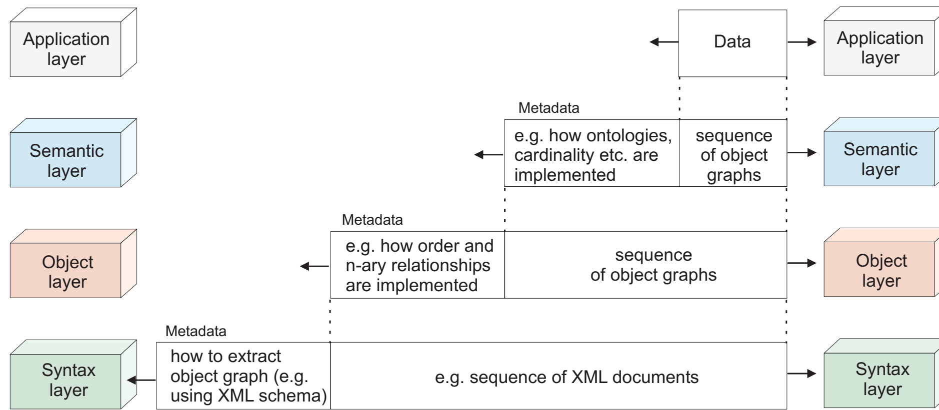

Fig. 2. An example of data modeling layers

As we will demonstrate in the following sections, every layer, or feature like "ordered relationships", can be logically implemented in a number of ways. Given n such features with m options each, this yields m^n possible incompatible model stacks, or data models. Clearly, direct interoperation between m^n data models is a tedious task that may require as many as $O(m^{2n})$ mappings between data models. This is exactly the obstacle that today's information integrators face. In fact, in current data modeling languages, the distinction between data modeling layers is blurred. Many standards like UML or XML Schema attempt to capture as many features as possible, reaching from the definition of syntactic elements to aggregation, class partitions, ontology languages etc. As a result, interoperation is exacerbated.

Using a layered approach sketched above, interoperation between data models can be simplified by an order of magnitude. Indeed, if every layer has m implementation options, only $O(m^2)$ mappings within a given layer are required in worst case. For n layers, this yields $O(nm^2)$ mappings, compared to $O(m^{2n})$. This simple quantitative analysis explains the tremendous success of layered internetworking. In computer networks, bridges and gateways are used to interoperate within a given layer like the data link layer. Analogously, data modeling gateways can be deployed to reduce the complexity of data interoperation. We call this approach *"interdataworking"*.

3 Data Modeling Languages for the Semantic Web

On the Semantic Web, a web of machine-processable information, a variety of data modeling languages are used. In this section we discuss several major languages. The goal of our discussion is to identify how specific data modeling primitives are used in those languages. In the next section we will show a way of organizing these modeling primitives into layers and sublayers. The list below contains the major types of approaches for data representation languages for the Semantic Web. Note that the selected list is by no means exhaustive:

- OEM (Object Exchange Model) [PGMW95] is a data model developed for Information Integration Projects at Stanford University.
- RDF and RDF Schema: RDF (Resource Description Framework) [LS98] and RDF Schema (the Schema Language for RDF) [Be00] are W3C Recommendations for describing metadata on the web.
- SHOE (Simple HTML Ontology Extensions) [HHL99] is an extension to HTML which allows

web page authors to annotate their web documents with machine-understandable content.
- UML (Unified Modeling Language) is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.
- OIL (Ontology Inference Layer) [FHH+00] is a language defined on top of RDF for enabling more expressive ontology definitions.

In the next sections we discuss these modeling languages in more detail.

3.1 OEM

OEM is one of the first and simplest information models that have been proposed for exchanging information on the Web by the database community. The main features it offers are object identity and nesting. The OEM model is a directed labeled graph, in which every object has a distinct identity and a type. Apart of atomic types like integers and strings, OEM supports sets and lists (sets and lists correspond to container types in RDF, see below). OEM object graphs can be represented in a graphical notation and serialized using a simple text-based syntax. This syntax is not based on XML; OEM was suggested before XML became available.

3.2 RDF and RDF Schema

RDF is a data model for representing data on the Web. RDF defines the following modeling primitives:

- *Object identity*: RDF distinguishes between resources, which have object identity (an OID), and literals, i.e. opaque strings. An OID is represented by a URI, a Uniform Resource Identifier (URIs are generalized URLs). A URI does not necessarily address a resource on the web. For example, the International Standard Book Number 0679405739, which identifies a 1992 edition of the novel "War and Peace", can be used as a URI "ISBN:0679405739". Thus, real world objects are represented in RDF using surrogates, or symbols that are associated with these objects.
- *Binary relationships*: Relationships in RDF are modeled via binary relations. Thus RDF models have the form of a directed graph.
- *Reification*: In RDF a specific binary relation between two objects is called a statement. To allow statements about statements, a statement can be reified, that means expressed by another objects with a certain set of properties. The object is a placeholder for the original statement, and hence can be used to make statements about the original statement.
- *Container*: RDF defines specific container types representing sequences, alternatives, and multisets.

RDF Schema is the Schema language for RDF. RDF Schema is similar to frame-based languages. The main modeling primitives defined in RDF Schema are:

- *Classes*: RDF Schema allows defining an explicit hierarchy of classes. A class is a resource and has a unique ID. The subClass relationship is defined by the property subClassOf.
- *Property and Property Constraints*: RDF Schema has modeling primitives for defining property constraints that restrict the range and domain of a property to certain classes.

Notice that RDF Schema is defined on top of RDF. RDF itself does not depend on RDF Schema.

3.3 SHOE

SHOE is an ontology-based knowledge representation language providing annotations for HTML pages. SHOE defines the following modeling primitives:

- *Categories*, which are similar to RDF-Schema classes, are defined with a <def-category> tag and may specify one or more subsuming categories (superclasses). Categories can be used to build term taxonomies, by defining a hierarchy of child-parent relationships.
- *Relations*: SHOE contains means to define n-ary relations, defined with a <def-relation> tag, which also contains type definitions for each argument.
- *Ontology Primitives*: Special modeling primitives are aiming at ontology administration: Ontology extensions are expressed in SHOE with the <use-ontology> tag, which contains the identifier and version number of the intended ontology. The <use-ontology> tag also allows to define a URL attribute (pointing to the ontology definition), and a prefix attribute used to define a local identifier for terms. The <def-rename> tag allows to define a renaming for a concept defined from another ontology.
- *Inference Rules* are defined using the <def-inference> tag to supply additional axioms. SHOE axioms are equivalent to definite Horn Clauses (a subset of First Order Predicate Logic, which resembles if-then rules).

3.4 UML

UML has been designed as a modeling language for software-intensive systems. UML possesses a comprehensive logical foundation. For this reason, it has been deployed for modeling tasks significantly broader than software engineering. In brief, UML has the following features relevant for our exposition:

- *Abstract notation* is a human-readable graphical notation of models. A UML model is comparable to a schema, or ontology. An instance of a UML model

comprises objects that participate in various relationships with each other.
- *XMI serialization*: XML Metadata Interchange (XMI) is an XML-based encoding standard for UML models. Object Constraint Language (OCL) is a formal language used to specify well-formedness rules for models. OCL can be compared in style with XML DTDs. DTDs constrain the number of possible valid instances of XML documents, whereas OCL constrains the number of possible valid instances of UML models.
- *UML CORBAfacility* is a programming language-neutral API for manipulating UML models. It defines classes like Classifier, Method, Generalization etc. with corresponding access methods and properties.
- *Four-layer metamodel structure*: the architecture of UML is based on a four-layer structure. The four layers are: user objects, model, metamodel, and meta-metamodel. In short, descriptions of models belong to higher levels of abstraction than the models themselves. That is, user objects, models, metamodels and meta-metamodels live in disjunct "worlds", and cannot directly relate to each other. UML is primarily concerned with the metamodel layer, which is an instance of the meta-metamodel layer. The meta-metamodel layer is defined in a separate standard called MOF (Meta-Object Facility). MOF is hard-wired, i.e. its semantics is considered to be well known. MOF's modeling primitives include Meta-Classes, MetaAttributes, MetaOperations etc.

UML defines a rich set of models for virtually all aspects of software engineering. These models are organized in packages and comprise interfaces and components, distribution and concurrency, activity diagrams, patters and collaborations etc. Features comparable to RDF's object identity and relationships are defined in the UML package Behavioral Elements/Instances and Links. As another example, the package Foundation/DataTypes defines the data types like Integer, Boolean, String, Time etc.

3.5 OIL

The OIL language is based on Description Logic (DL)-oriented ontologies and has a well-defined first-order semantics and automated reasoning support, e.g. for class consistency and subsumption checking. OIL supports the following modeling primitives:

- *Ontology Metadata*, based by Dublin Core Metadata Element Set. Ontology definitions consist of an optional import statement, an optional rule-base and class and slot definitions.
- A *slot definition* (slot-def) associates a slot (a binary relation) name with a slot definition. A slot definition specifies global constraints that apply to the slot relation. A slot-def can consists of a subslot-of statement, domain and range restrictions, and additional qualities of the slot, such as being inverse, transitive, and symmetric.
- A *class definition* (class-def) associates a class name with a class description. Sophisticated class definitions (e.g. equivalence between classes, i.e. renaming, or subclass-statements) are expressed as Boolean combination of class expressions using the operators AND, OR or NOT. Slot-constraints, which relate a class to a certain property (or slot), are also class expressions. Possible slot-constraints are:
  - *has-value*: Every instance of the class defined by the slot constraint must have a slot value that is an instance of each class-expression in the list.
  - *value-type*: If an instance of the class defined by the slot-constraint is related via the slot relation to some individual x, then x must be an instance of each class-expression in the list.
  - *max-cardinality*: An instance of the class defined by the slot-constraint can be related to at most n distinct instances of the class-expression via the slot relation (similar are min-cardinality and, as a shortcut, cardinality).

The syntax of OIL is oriented towards XML and RDF. [HFB$^+$00] defines a DTD and a XML schema definition for OIL and [BKD$^+$00] defines OIL as an extension of RDFS.

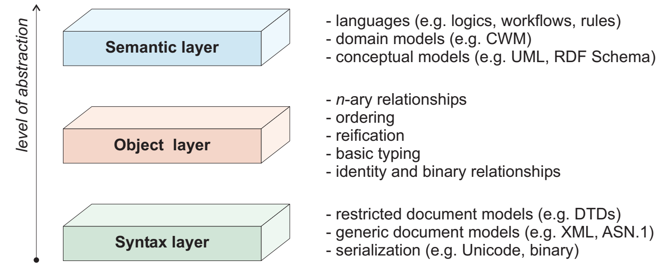

Fig. 3. The syntax, object, and semantic layers

4 Syntax, Objects, and Semantics

To reduce the complexity of building the Semantic Web, we suggest viewing Web-enabled information models as a series of layers: the syntax layer, the object layer, and the semantic layer (see Fig. 3). Every layer may have multiple sublayers. The semantic layer, or knowledge representation layer, deals with conceptual modeling and knowledge engineering tasks. The basic function of the object layer, or frame layer, is to provide applications with an object-oriented view of their domain. The syntax layer is responsible for "dumbing down" object-oriented information into document instances and byte streams.

Every of the three layers has a number of sublayers. Every sublayer corresponds to a specific data modeling feature, e.g. aggregation or reification, that can be

logically implemented in many different ways. Plausible criteria for designing the layers and sublayers include grouping two (sub)layers if they have mutual dependencies, or merging a sublayer that has a single possible implementation option with an adjacent sublayer. In the rest of this section, we briefly describe each of the layers in a bottom-up fashion.

4.1 Syntax Layer

The main task of the syntax layer is to provide a way of serializing information content into a sequence of characters or bits. Any application that exchanges information with other applications or stores it persistently needs to structure the information carefully so that the recipient or reader can retrieve the information in its original form. Data structures used by applications are typically not just flat lists of uniform data elements. Therefore, additional markup mechanisms are required to preserve nested structure of data.

In the past two years we have witnessed how an impressive global-scale agreement on a common syntax layer has been achieved. XML has become pervasive, its use ranging from electronic publishing to electronic business. XML tagging or ASN.1 encoding rules are examples of markup mechanisms for preserving the structure of data. The syntax layer could be divided into three sublayers (bottom-up):

- *serialization*: data instances are serialized into byte streams. For example, XML documents are serialized as Unicode character strings, whereas ASN.1 uses a binary encoding.
- *generic document models*: applications structure information as nested data structures. XML provides a generic document model. Instances of this model can be manipulated using APIs like XML DOM.
- *restricted document models*: sometimes applications want to enforce structural constraints on the nested data structures they use. XML Document Type Definitions (DTDs) are an example of grammars for describing such structural constraints.

To reconstruct the objects and relationships from the representation used in the syntax layer, metadata is required. Obviously, two peer syntax layers need to agree on the metadata standard used for this purpose. For example, the emerging XML Schema standard can be deployed to extract objects and their relationships from an XML document. In this case, a specific XML schema would capture the information on how to identify objects. Another alternative is to choose a more generic or implied encoding standard. For example, [LS98] describes a standard way of encoding object graphs in XML. In this case, no special schema needs to be appended to the XML data.

4.2 Object Layer

The purpose of the object layer is to offer applications an object-oriented view on the information that they operate upon. In contrast to data structures, objects have immutable object identity, i.e. change of an object's identity results in a different object. The very basic function of the object layer is to enable manipulation of objects and binary relationships between them. However, different applications may require more functionality of the object layer, depending on their complexity. We identified four additional sublayers that are often used. In Section [5] we describe the object layer in more detail. Here is a brief summary of the five sublayers:

- *identity and binary relationships*: every object-oriented model provides these features.
- *basic typing*: a simple abstraction mechanism. One object is used to "type" another object.
- *reification*: some information models (e.g. RDF, UML) require access to whole relations and individual links between objects.
- *ordering*: ordered relationships are integral part of some information models (e.g. UML).
- *n-ary relationships* are deployed in information models like SHOE.

4.3 Semantic Layer

Roughly speaking, the semantic layer provides interpretation of the object model used in the object layer. This objects, or surrogates, used in the object layer are mapped onto physical and abstract objects like books, airplane tickets, database tables, logical formulae and paragraphs of text. The ultimate goal of the Semantic Web is to make applications interoperate in the semantic layer. The semantic layer is comprised of a number of rich and complex sublayers like:

- *conceptual models*: vocabularies for representing conceptual models (e.g. RDF Schema, UML Foundation/Core). Features of conceptual models may include elements like generalization, aggregation, cardinality constraints etc.
- *domain models*: deal with ontologies of a particular application domain, e.g. transportation, manufacturing, e-business, digital libraries, Web resources, etc.
- *languages*: instead of using a natural language, the machines on the Semantic Web convey information using formal languages. These languages can be highly specialized or serve a general purpose. Examples include workflow definition languages, Datalog, first-order logic, UML statecharts, SQL etc. Terms and expressions in these languages are first-class objects that can be manipulated on the object layer. In this way, applications can dynamically learn the semantics of previously unknown languages.

5 Object Layer: Features and Design Issues

The object layer is the focus of our paper. In this section we discuss the five sublayers of the object layer in more detail. We illustrate the features of these sublayers using examples from RDF, UML, SHOE and OEM. The purpose of our discussion is to gain a better understanding of the design issues involved in the object layer. Such understanding is beneficial for the specifications of the mappings between object layers in different model stacks. Ultimately, we hope it can contribute to an agreement on the capabilities of object layers similar in scale to an agreement on the syntax layer (XML).

In the discussion of the object layer we are again following a bottom-up approach, i.e. from the ground-level features to more high-level ones. The design issues that we consider in this section have a logical character. They do not necessarily preclude the variety of implementation alternatives at the programming level. Nevertheless, a logical implementation can have major impact on the API design. In Sect. 6 we briefly examine some programming-level implementation issues.

5.1 Identity and Binary Relationships

Object identity and binary relationships can be seen as the least common denominator between any two object-oriented models. A model lacking object identity is simply not an object-oriented model any more [Cat91]. In every object-oriented model, objects do not exist on their own, but engage in multiple relationships with each other. Binary relationships is the simplest form of such relationships. Notice that at the object layer we deal with objects at the instance level, i.e. every object is treated as an individual identifiable entity.

As long as an application does not need to exchange information with other applications, is does not matter how the objects are identified. In fact, suitable object-oriented APIs may hide the object identity from the programmer completely. To take advantage of the Semantic Web, applications need to communicate, either directly or indirectly by publishing information in machine-readable form. Thus, explicit identifiers for the objects are required. In RDF, objects are identified using the Uniform Resource Identifiers (URIs), a generalized form of Uniform Resource Locators (URLs). A similar approach is taken by SHOE. In UML, objects are identified using Universally Unique Identifiers (UUIDs). OEM allows any unique variable length identifiers. URIs, UUIDs etc. support global identity for the objects, which is a prerequisite for building the Semantic Web.

Information models use different abstract notations for binary relationships between objects. In this paper, we adopt the RDF notation. Fig. 4 illustrates a binary relationship between a source and a destination object. As a rule, the position of the object in a relationship, i.e.

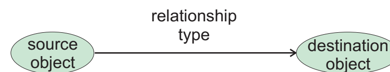

Fig. 4. Abstract notation for a binary link between two objects

source or destination, is significant. In RDF, every such relationship is viewed as a statement, or assertion. The source and destination objects are the subject and the object of the assertion, respectively.

In UML, relationships between object instances as shown in the figure are referred to as *links*. The relationship type, i.e. the relationship as a whole in the semantic layer, is called *association*. To avoid ambiguity, we follow this terminology.

5.2 Basic Typing

Information models sometimes deploy a primitive typing mechanism to differentiate the objects among each other. Another object is used to denote the type of the given object. We refer to such mechanism as basic typing. The semantics of basic typing is broader (less strictly defined) than that of instantiation; if two objects are of the same type, they can be used in a similar context within the application.

In OEM, basic typing is used to denote atomic types such as integer or string, and container types such as set or list. Since the "types" themselves are first-class objects, the application can request additional information about the types.

In RDF, the purpose of basic typing is to allow bootstrapping of more complex typing facilities in the semantic layer. In the notation used above, basic typing of an object A using object B is represented as an arc from A to B with a label type that denotes basic typing.

5.3 Reification

To refer to individual links between objects, or to associations (relationships as a whole), a reification mechanism is required. Reification is Latin for "making into a thing". Using reification, links and associations can be treated as first-class citizens in an information model. Associations are reified to enable applications to provide additional information about them. Reification of links can be used for multiple purposes. For example, in RDF every link corresponds to an assertion. Thus, reification of links provides a "quotation" mechanism, i.e. an application can refer to information stated by another application. Being able to discuss, dispute or support the relationships and properties of objects is a crucial prerequisite for machine communication. On the other hand, reification of links can be used to implement nesting of instances of information models.

Reification of links and associations is illustrated in Fig. 5. The big oval denotes the object that represents

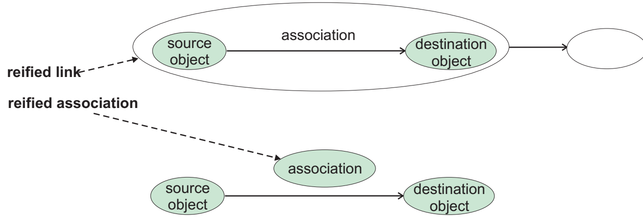

Fig. 5. Reification of links and relationships

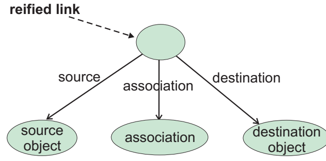

Fig. 6. Logical implementation of reified links in RDF

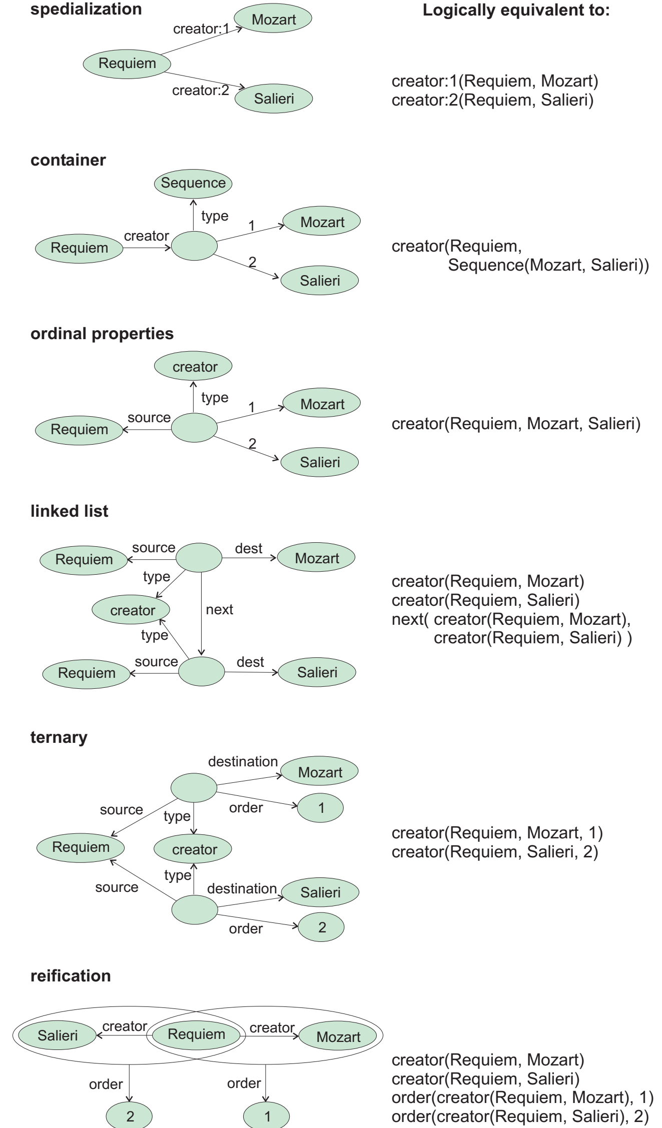

Fig. 7. Implementation alternatives for ordered relationships

the reified link. In the figure, this object is used as the source for another link. To emphasize reification of the association in the bottom part of the figure, the association is circumscribed as an object. This object can, too, participate in other links. These two kinds of reification provide the necessary prerequisites for computational reflection, i.e. the capability for a computational process to reason about itself [Smi96].

Both UML and RDF support reification of links and associations. In both standards, link reification is logically implemented by introducing a new object with properties that identify the parts of the link. The logical implementation of link reification in RDF is illustrated in Fig. 6.

5.4 Ordering

Some information models like UML make heavy use of ordered relationships. To illustrate ordered relationships, consider the DublinCore association "creator". In Pushkin's poem "Mozart and Salieri", which inspired the movie "Amadeus", both Mozart and Salieri were presented as the "creators" of "Requiem". No doubt, when modeling the authorship of "Requiem", we want Mozart to be the primary author.

Fig. 7 illustrates six logical implementation alternatives for the ordered binary relationship between "Requiem" and the two composers. The right-hand size of the figure presents a "logical" view of the object graphs. The six alternatives are named *specialization*, *container*, *ordinal properties*, *linked list*, *ternary*, and *reification*, according to their logical implementation. Notice that although any representation can be bijectively translated into every other one, they are more or less semantically faithful. For example, the second representation (container) is particularly semantically misleading for representing ordered relationships since it states that the creator of "Requiem" is an object typed as Sequence.

A qualitative comparison of the alternatives is presented in Tab. 1. Besides semantic faithfulness, we consider how difficult it is to use the same logical schema for representing the inverse order. Inverse order is required when the objects at the source end that are related to a single object at the destination end have an ordering that must be preserved. For example, if the "creator" association were to capture the chronological order of the pieces written by the composers, representation for the inverse order would be needed. We gave a minus (−) to the schemes that required creation of additional reified objects for links or associations to support inverse order.

Finally, the last metric that we consider here is the implementation effort. By implementation effort we mean not the effort needed to implement the API that allows manipulating ordered relationships, but the effort needed to use such an API. The typical operations we considered are

– find all creators of "Requiem" (ignore order)

Alternative	Semantic faithfulness	Inverse order	Implementation effort
specialization	++	−	−−
container	−−	−	−−
ordinal properties	+−	−	−
linked list	+−	+	+
ternary	+−	+	+−
reification	+	+	+−

Table 1. Logical implementation alternatives for ordering

– find all properties of "Requiem" (hide auxiliary objects like instances of Sequence)
– find the first creator of "Requiem"
– add/insert a second (third etc.) creator

The schemes specialization and container are especially implementation-intensive. Specialization requires tracing the ordered versions of "creator" like "creator:1" for every access, whereas container entails checks to determine whether a single object or a bag is the destination of the link. In our comparison, we only considered ordered binary relationships, since ordered n-ary relationships are used very seldom and their semantics is typically hard to comprehend. Although ordering by reification looks fairly verbose in the figure, we found that it has a number of preferable characteristics that make it a viable choice for a logical implementation of ordered relationships.

In some information and data models, ordering is built-in, i.e. it cannot be reduced to other modeling primitives like reification and binary relationships. Such models include UML, OEM, and XML. In other models like RDF and SHOE, ordering is not a built-in feature and can be implemented in various ways, similarly to the alternatives that we considered above. The choice of alternatives depends on the availability of modeling primitives. For example, since SHOE lacks reification, ordering by reification is out of the question.

5.5 *n-ary Relationships*

The last important feature that belongs to the object layer are n-ary relationships. Although n-ary relationships are used fairly seldom compared to binary relationships, they are supported by some Web-enabled information models like UML or SHOE. Hence, in general, Semantic Web applications should be prepared to deal with n-ary relationships.

Often, n-ary relationships are logically implemented on top of the four sublayers discussed above. Nevertheless, a clear definition of the semantics of n-ary relationships is crucial for interoperability between information models. n-ary relationships cannot be implemented as a combination of binary relationships without using additional objects. Thus, in the object layer, an n-ary link

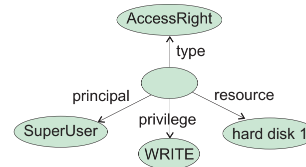

Fig. 8. Example of a ternary link

is typically represented as an object that is linked to n objects participating in the link (see Fig. 8).

Notice that the logical implementation depicted in the figure does not impose a specific implementation on the programming level. For example, the above ternary link can be implemented as a 3-tuple in a relational database. Using n-ary relationships, however, requires specifically designed API methods.

5.6 *Summary of Object Layer Features*

The five features of the object layer that we discussed above are summarized in Tab. 2. The features that are implementable, but are not standardized in a particular information model are not counted. For example, order can be implemented in SHOE in many different ways using n-ary relationships, but every application may do it in a different way.

Some features in the table are marked as implicit. These features, like ordering in UML or n-ary relationships in SHOE, are visible on the API level only. They are not directly represented in the object model.

In our discussion of the object layer we do not intend to pinpoint the "best" logical implementation of each sublayer. It is clear that the designers of different data models may choose one or another option depending on their needs. Thus, in two distinct data models ordering, for example, may be implemented either in a ternary fashion or using reification. Instead of choosing the best option, our goal is to emphasize the usefulness of each sublayer, and provide a roadmap for designing bridges and gateways between similar sublayers in different model stacks.

Feature	RDF	UML	SHOE	OEM	OIL
object identity and binary relationships	+	+	+	+	+
basic typing	+	+ (implicit)	+ (implicit)	+	+
reification	+	+	0	0	0
ordering	0*	+ (implicit)	0	0*	0
n-ary relationships	0	+	+ (implicit)	0	0

Table 2. Object layer features in RDF, UML, SHOE, OEM and OIL.

*: RDF containers and OEM lists do not carry the semantics of ordered relationships.

6 Implementation and APIs for Object Layer

In the previous section we discussed the logical implementation of the object layer. This section deals with the programming-level realization. The logical design of the object layer largely determines both the APIs and the exchange syntax (i.e. mapping to the syntax layer) used for implementing the object layer. In particular, the APIs need to consider both navigational and declarative access to the objects. The navigational access is of primary relevance for in-memory implementations, whereas declarative access is important for database support. Layer and sublayer APIs are an integral part of a layered data modeling architecture and require careful design. In this section, we illustrate some factors that need to be considered for such design.

6.1 Navigational Access

Traditionally, object-oriented models deploy APIs that are tailored for the object model of a given application. The objects are represented by instances of programming language objects. The properties or links between objects are accessed using member variables or get/set methods. Basic typing is provided by the typing system of the programming language. While perfect for a closed domain, such APIs are very inflexible. For example, it is usually not possible to add a property of a new kind to an object at runtime. Furthermore, inheritance schemes are fixed (e.g. single inheritance) and cannot be chosen by developers. Tailored APIs usually do not support reification of binary links. Generic APIs like [Mel99] provide the necessary flexibility. However, they are not as compact and intuitive, increasing development time and maintenance costs. If an API provides a class like "Link", reification comes usually for free, since instances of "Link" can be used without worrying about their identity. "For free" means that no additional objects and arcs as shown in Fig. 6 need to be created. When such lightweight reification is possible, a convenient strategy for implementing ordered relationships is order by reification. n-ary relationships can be realized in a straightforward fashion mirroring the logical implementation described in the Sect. 5.5.

6.2 Declarative Access

Relational or object-oriented databases usually offer a declarative query language. An important consideration for implementing the object layer on top of a database is how well the querying capabilities of the database can be exploited. A clever implementation would be able to translate many kinds of declarative access operations into a single database query. In such cases, the query optimizer of the database system can be used effectively. The tradeoff between tailored and generic representations applies for databases similarly as for APIs. For example, if associations are represented as separate relational tables, the capability of reification of associations is lost, and no queries with variable associations are possible.

To illustrate the importance of the design of the object layer, consider the following implementation of ordering using a relational DBMS. In this implementation, a single table tuples holds binary links between objects in a generic fashion. The table contains four fields that represent object identifiers, all fields are of the same type (Object identifiers in a database system are typically implemented as integers. In the examples below we are using stylized string values). The implementation uses order by reification. A sample content of the database is shown below.

```
ID  S         P        O
--------------------------------
id1 Requiem   creator  Salieri
id2 Requiem   creator  Mozart
id3 id1       order    2
id4 id2       order    1
id5 Pinocchio creator  Geppetto
```

The table contains two ordered links and one unordered link. The field ID contains identifiers of reified links. All "find"-queries listed in Sect. 5.4 as implementation criteria can be executed using a single SQL query. The most sophisticated query of these is retrieving the first creator. The complicating factor is that some creators are unordered. Still, retrieving the first creator for an object like Requiem can be done using the following single query:

```
SELECT t1.S, t1.O
FROM   tuples AS t1
       LEFT JOIN tuples AS t2 ON t1.ID=t2.S
WHERE  t1.S=Requiem AND
       t1.P=creator AND
      (t2.P IS NULL OR t2.P=order) AND
      (t2.O IS NULL OR t2.O=1)
GROUP BY t1.S
```

The GROUP BY clause is required to reduce the number of multiple unordered creators to one. The first creators of all objects can be retrieved by dropping the first conjunct in the WHERE clause. The result of the query would be:

```
(Requiem, Mozart)
(Pinocchio, Geppetto)
```

6.3 Mapping to the Syntax Layer

The mapping to the syntax layer can be optimized to support the features provided by the object layer. As an example, consider how serialization of reification and order can be implemented in a compact way. Order by reification can be expressed as

```
<tuple ID="id1" S="Requiem" P="creator"
       O="Mozart"/>
<tuple SID="id1" P="order" O="1"/>
```

The XML attribute SID in the second tuple is a reference to an ID attribute declared in the first tuple. For even more compact representation, a specialized ordering syntax can be used. Thus, the fact that Salieri is the second creator can be serialized as:

```
<tuple S="Requiem" P="creator" O="Salieri"
       order="2"/>
```

7 Related Work and Conclusion

In this paper we make the following three contributions. First, we analyze some information models and suggest a layered reference model for Information Model Interoperability. The reference model allows reducing the complexity of achieving interoperability between information models on the Semantic Web. We identify the object layer and examine its features in detail. Finally, we discuss issues involved in implementation of the object layer to illustrate requirements for object layer APIs.

In our approach to structuring the Information Model Interoperability (IMI) reference model we are building on the analogy with the Open Systems Interconnection (OSI) reference model used in computer networks (see [Tan97] for a good summary). One of the major contributions of OSI is to provide a clear distinction between services, interfaces and protocols used in internetworking, enabling a stack of services on top of the more basic levels. Using layering for data modeling includes the following advantages:

- The complexity of data model interoperation is reduced to interoperation within a specific layer or sublayer. Bridges and gateways can be built to support mappings between layers.
- Comprehensive APIs for individual layers facilitate reuse and reduce the costs of application development. For example, OIL reuses RDF parsing and querying tools.
- Implementations of the layers and agreements between peer layers can be exchanged without affecting the applications.

Layering techniques have also been tried in knowledge representation systems [Bra79]. Brachman's architecture is aiming at the implementation of a knowledge representation system, starting from data structures, which are used to implement logical inference engines, to adequate epistemological primitives, which facilitate knowledge modeling and engineering. In contrast, the IMI models deals primarily with interoperation between information providers and consumers. Our model is structured in a way that similar fundamental knowledge representation primitives are situated in the same or close by layers. The stratification of epistemological primitives is not given in Brachman's architecture: the epistemological primitives (one layer in Brachmans model) are situated in several IMI layers.

On a limited scale, a layered approach to data modeling has been successfully tried in practice. For instance, [Mel00] demonstrates how the semantic layer of UML can be built on top of RDF, and [BKD+00] defines OIL as an extension of RDF Schema on top of the object layer of RDF. Such reuse eliminates effort for defining yet another object model or syntax, and boosts interoperability. As another example, many information models adopted XML for their syntax layers and are able to reuse XML tools and parsers developed by third parties. We believe that a layered approach to data modeling can be an important step toward the realization of the Semantic Web.

References

[Be00] Dan Brickley and R.V. Guha (eds). Resource Description Framework (RDF) Schema Specification 1.0. W3C Candidate Recommendation, 2000.

[BKD+00] J. Broekstra, M. Klein, S. Decker, D. Fensel, and I. Horrocks. Adding formal semantics to the Web: building on top of RDF Schema. Technical Report: Free University of Amsterdam, 2000.

[Bor85] A. Borgida. Features Of Languages For The Development Of Information Systems At The Conceptual Level. *IEEE Software*, Jan 1985.

[Bra79] R. J. Brachman. On the Epistomological Status of Semantic Networks. In *Findler, Nicholas V. (Ed., 1979). Associative Networks. Representation and Use of Knowledge by Computers. New York: Academic Press*, volume 3-50, 1979.

[Cat91] R. G. G. Cattell. *Object Data Management.* AW, 1991.

[DMvH+00] S. Decker, S. Melnik, F. van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks. The Semantic Web: the Roles of XML and RDF. *IEEE Internet Computing*, Sep 2000.

[DSS93] R. Davis, H. Shrobe, and P. Szolovits. What is a Knowledge Representation? *AI Magazine*, 14(1):17–33, 1993.

[FHH+00] D. Fensel, I. Horrocks, F. Van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a Nutshell. In *Knowledge Acquisition, Modeling, and Management, Proceedings of the European Knowledge Acquisition Conference (EKAW-2000), R. Dieng et al. (eds.), Lecture Notes in Artificial Intelligence, LNAI, Springer-Verlag*, Oct 2000.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From Semistructured Data to XML: Migrating the Lore Data Model and Query Language. In *WebDB Workshop*, 1999.

[HFB+00] I. Horrocks, D. Fensel, J. Broekstra, S. Decker, M. Erdmann, C. Goble, F. van Harmelen, M. Klein, S. Staab, R. Studer, and E. Motta. The Ontology Inference Layer OIL. Technical Report, Free University of Asterdam, 2000.

[HH00] J. Heflin and J. Hendler. Semantic Interoperability on the Web. In *Proc. of Extreme Markup Languages*, 2000.

[HHL99] J. Heflin, J. Hendler, and S. Luke. SHOE: A Knowledge Representation Language for Internet Applications. Technical Report CS-TR-4078 (UMIACS TR-99-71), 1999.

[LS98] O. Lassila and R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. `http://www.w3.org/TR/REC-rdf-syntax/`, 1998.

[Mel99] S. Melnik. An API for RDF. `http://www-db.stanford.edu/~melnik/rdf/api.html`, 1999.

[Mel00] S. Melnik. Representing UML in RDF. `http://www-db.stanford.edu/~melnik/rdf/uml/`, 2000.

[PGMW95] Y. Papakonstantinou, H. García-Molina, and J. Widom. Object Exchange Across Heterogeneous Information Sources. In *Proc. of the 11th IEEE Int. Conf. on Data Engineering (ICDE)*, pages 251–260, Taipe, Taiwan, March 1995.

[Smi96] Brian C. Smith. *On the Origin of Objects.* MIT Press, 1996.

[Sow00] John F. Sowa. Ontology, Metadata, and Semiotics. In *Proc. Int. Conf. on Conceptual Structures (ICCS)*, Aug 2000.

[Tan97] Andrew. S. Tanenbaum. *Computer Networks.* Prentice-Hall, 3rd ed, 1997.