Frontiers in Artificial Intelligence and Applications

ONTOLOGY AND THE SEMANTIC WEB

Robert M. Colomb

IOS Press

ONTOLOGY AND THE SEMANTIC WEB

Frontiers in Artificial Intelligence and Applications

FAIA covers all aspects of theoretical and applied artificial intelligence research in the form of monographs, doctoral dissertations, textbooks, handbooks and proceedings volumes. The FAIA series contains several sub-series, including "Information Modelling and Knowledge Bases" and "Knowledge-Based Intelligent Engineering Systems". It also includes the biennial ECAI, the European Conference on Artificial Intelligence, proceedings volumes, and other ECCAI – the European Coordinating Committee on Artificial Intelligence – sponsored publications. An editorial panel of internationally well-known scholars is appointed to provide a high quality selection.

Volume 156

Recently published in this series

Vol. 155. O. Vasilecas et al. (Eds.), Databases and Information Systems IV – Selected Papers from the Seventh International Baltic Conference DB&IS'2006
Vol. 154. M. Duží et al. (Eds.), Information Modelling and Knowledge Bases XVIII
Vol. 153. Y. Vogiazou, Design for Emergence – Collaborative Social Play with Online and Location-Based Media
Vol. 152. T.M. van Engers (Ed.), Legal Knowledge and Information Systems – JURIX 2006: The Nineteenth Annual Conference
Vol. 151. R. Mizoguchi et al. (Eds.), Learning by Effective Utilization of Technologies: Facilitating Intercultural Understanding
Vol. 150. B. Bennett and C. Fellbaum (Eds.), Formal Ontology in Information Systems – Proceedings of the Fourth International Conference (FOIS 2006)
Vol. 149. X.F. Zha and R.J. Howlett (Eds.), Integrated Intelligent Systems for Engineering Design
Vol. 148. K. Kersting, An Inductive Logic Programming Approach to Statistical Relational Learning
Vol. 147. H. Fujita and M. Mejri (Eds.), New Trends in Software Methodologies, Tools and Techniques – Proceedings of the fifth SoMeT_06
Vol. 146. M. Polit et al. (Eds.), Artificial Intelligence Research and Development
Vol. 145. A.J. Knobbe, Multi-Relational Data Mining
Vol. 144. P.E. Dunne and T.J.M. Bench-Capon (Eds.), Computational Models of Argument – Proceedings of COMMA 2006

ISSN 0922-6389

Ontology and the Semantic Web

Robert M. Colomb
School of ITEE, University of Queensland

IOS Press
Amsterdam • Berlin • Oxford • Tokyo • Washington, DC

ISBN 978-1-58603-729-1
Library of Congress Control Number: 2007922436

Publisher
IOS Press
Nieuwe Hemweg 6B
1013 BG Amsterdam
Netherlands
fax: +31 20 687 0019
e-mail: order@iospress.nl

Distributor in the UK and Ireland
Gazelle Books Services Ltd.
White Cross Mills
Hightown
Lancaster LA1 4XS
United Kingdom
fax: +44 1524 63232
e-mail: sales@gazellebooks.co.uk

Distributor in the USA and Canada
IOS Press, Inc.
4502 Rachael Manor Drive
Fairfax, VA 22032
USA
fax: +1 703 323 3668
e-mail: iosbooks@iospress.com

PRINTED IN THE NETHERLANDS

To Max

This page intentionally left blank

Ontology and the Semantic Web
R.M. Colomb
IOS Press, 2007

Contents

List of Figures and Tables

Ontology and the Semantic Web
R.M. Colomb
IOS Press, 2007

1 Introduction: the Dream of Interoperability

What problem would you be trying to solve that you need an ontology for?

Nearly every business or organization is supported by an information system. Companies in the finance, insurance and real estate industry group are very little more than information systems. Mines and farms use information systems to keep track of production and assets. Manufacturers, wholesalers and retailers use information systems to manage their production, sales and employees. Construction firms use information systems to bid for and manage projects. Transportation firms use them to schedule services. The health sector is served by thousands of systems assisting in the operation of various departments in hospitals, doctors' surgeries, and the flow of payments through the system. Universities use information systems to keep track of students, courses, libraries and staff. Governments use them to record births, deaths and marriages, and in the provision of all sorts of services.

So there are millions of information systems. Since the mid-1990s it has been possible to make these information systems universally accessible via the world-wide web. A large and rapidly increasing fraction of systems have indeed been made available via the web.

Anyone who does anything in this world will typically interact with several businesses or organizations supported by information systems, sometimes very many. A shopper will visit several stores comparing products, availability and price. All but the simplest medical conditions involve several doctors, pharmacies, pathology laboratories and other services. A trip can involve airlines, hotels, car hire companies, travel insurance and tours, mediated by travel agents, facilitated by credit card companies and bank automatic teller machines. A manufacturer will interact with many suppliers, customers, transportation services, banks, stock exchanges and governments at many levels in order to carry on its business.

If doing something involves interaction with many businesses and organizations, each supported by information systems, it is reasonable to expect that the information systems themselves will interoperate to support the interaction. This in fact happens in many areas. Funds transfer among banks around the world is done using an electronic funds transfer (EFT) system called SWIFT that links the information systems together. It is possible to assemble complex travel itineraries with confirmed reservations because the airlines share a small number of common reservation services, the first of which, called SABRE, was developed in the 1960s by American Airlines. Many governments support electronic lodgment of income tax returns with refunds being paid directly to the taxpayer's bank account using EFT. And of course purchase by point-of-sale (POS) mediated EFT or credit card is so common as to be hardly noticed. On a larger scale, major manufacturers and retailers are integrating their systems with their suppliers' in a process called supply chain management mediated by electronic document exchange (EDI).

Having all these information systems publicly accessible on the web opens many additional possibilities. There are an enormous number of services. Accessing these services is often tedious and time-consuming. Furthermore, the services are changing

continuously. Opportunities often arise which must be noticed and acted upon within a short time window. The Internet must be constantly monitored, which is also tedious and time-consuming.

Computers were invented to perform tedious and time-consuming tasks, so that one would look to computer programs to at least help us with dealing with services available on the web. On an individual level, why not comparison shop on the web? Or build an entire travel itinerary, not just airline reservations, without using a travel agent? On a company level, why not make arrangements with suppliers on the fly? Or automatically manage outsourcing? Or even build a virtual enterprise which automatically outsources all of its operational functions on the fly?

On a more strategic dimension, if in a given jurisdiction all university programs, courses and enrolment numbers are published on the web, we should be able to find out how many students are studying in each field, so as to be able to predict the supply of graduates in the next few years. If a great variety of statistical resources are published on the web (the Bureau of Statistics in the small country Australia publishes tens of thousands of time series), why not automate routine decisions as to whether a particular proposed business is likely to be viable at a proposed location? Or we might want to track the topics reported in the news media over a period of time.

There actually are many such programs, for example:

- Alerters which monitor publications for articles which satisfy an interest profile, generally managed by indexing services for scientific publications. Medline, operated by the National Library of Medicine in the United States supports a service like this. There are also programs which monitor chat room conversation openers in a similar way.
- So-called meta-query facilities which broadcast a query to multiple information sources, producing a combined result. It is possible to search all the university libraries in Australia with a single query. Meta-search engines like dogpile.com or mamma.com accept a query then automatically pose it to several search engines.
- Shopping bots which search multiple sources of standard products like books, CDs or software for best price or fastest delivery or some other criterion. Bottomdollar.com is an example.
- Auction bots which participate in auctions for designated objects, making bids implementing a bidding strategy.
- Purchasing programs which execute complex purchasing protocols on electronic commerce exchanges.

These sorts of programs are often called *agents*, because they operate in an open environment, and are often hosted on a server so can run independently of the user.

Besides these fairly common examples, there are more complex situations where people have proposed designs for agents but which are not yet realized, at least in routine robust ways:

Travel planning beyond a simple trip ticket purchase or hotel booking, involving itinerary planning, multiple transport mode booking, multiple accommodation booking, booking at attractions, and notification of items of interest, including travel warnings or potentially attractive events.

Tendering for complex products or services, possibly assembling a whole purchase from partial tenders. One might want to equip an office with personal computers, telecommunications equipment and local networks, for example.

Configuring an intelligent house using computers and a network, but including appliances, home security, cameras and microphones, air conditioning, lighting, home entertainment and so on, permitting internet and wireless access but keeping out electronic intruders.

The purpose of this work is to explain how this dream can be realized. Its main thesis is that to for a group of systems to interoperate, the organizations responsible for the systems must first agree on what the words mean in the interoperation. This agreement is called an *ontology*, a description of the world shared by the participants.

We begin with analysis of a simple example.

1.1 A Book-Shopping Bot

One of the common applications is a shopping bot, for concreteness let us say an agent to find and order a book from one of the Web-based booksellers, choosing the optimum combination of price and delivery time according to the user's instructions. Let us assume that the agent is given the title and author of the desired book, together with a maximum price and a maximum delivery time. The agent's task is to find a supplier with the lowest price whose delivery time is not greater than the maximum delivery time.

1.1.1 Who to Talk to

The first problem the agent has is to know which of the hundreds of millions of web sites are actually booksellers. One possible way is to use a search engine. Figure 1 is the first page of results using the search engine Google, at the time of writing, with the keyword "bookseller"

The first three results are bookseller trade organization sites, not booksellers. The fourth site is a major international bookselling site, so is a potential target of a purchase request. But the fifth site is a bookseller in Brazil (not good for an English-language request), and the remaining four are all very specialized booksellers. Furthermore, notice that the major online bookseller Amazon.com does not appear. It does not appear even on subsequent pages. But it is the first site in the result if the search term is "bookstore" instead of "bookseller".

This sort of situation will be familiar to anyone who has used search engines or other information retrieval systems. In the discipline of information science, the fact that a search result contains unwanted items is called *limited precision*, and the fact that a search result may not contain wanted items is called *limited recall*. Limited precision and limited recall are characteristic of such systems, to the extent that sixty years of research has not had much success in improving either precision or recall.

What can our agent make of this? It would seem pretty clear that we could not trust a program with power to spend our money to identify appropriate booksellers without assistance of some kind. The assistance would have to be more than a better selection of keywords, since changing keywords and complexifying queries is known to change the result set, but not to eliminate the limited precision and limited recall of the service.

- Welcome to TheBookseller.com/The Bookseller is a central source of industry information for publishers and booksellers in the UK....
- thebookseller.com - careers
- www.bookweb.org/bookstores/Trade organization devoted to the support of booksellers.
- BarnesandNoble.com - The World's Largest Bookseller Online
- Bem-Vindo à Bookseller Editora-/Bem-Vindo à Bookseller Editora, uma das mais conceituadas editoras jurídicas do Brasil. ...
- Henry Hollander, Bookseller Home Page/FEATURED ITEMS: Western Jewry: An Account of the Achievements of the Jews and Judaism in California, Henry Hollander, Bookseller is pleased to announce the re / Antiquarian and scholarly bookstore specializing in Judaica.
- Michael Shamansky, Bookseller Inc./Bookseller Inc. PO Box 3904, Kingston, New York 12402 US Phone: 845-331-8519 ...
- Ken Lopez - Bookseller, ABAA. Modern First Editions/ Good as it is to inherit ... not be shared with anyone. email:
- Pat Ledlie - Bryology in Maine/Bookseller for books about conservation biology, environmental science, and natural history books.

Figure 1. First page of results from Google search for "bookseller"

One way to solve the problem is for the user to manually identify a number of appropriate booksellers, and to provide the agent with a list of sites. (This is in fact how shopping bots work.) A generalization of this method is if someone makes, publishes and maintains a list of bookseller web sites. The user now needs only to find an appropriate bookseller directory site and tell the agent about that.

A little thought will reveal that not just any bookseller directory site will do. A minimum criterion is that we need a site which is current (all links go to actually functioning bookseller sites). However, since our agent is going to spend our money, we need a directory of reputable and reliable booksellers. We don't want sites which will take our money and not deliver anything, or which frequently deliver the wrong item or frequently deliver more slowly than the advertised delivery time. So whoever compiles the directory must put in considerable effort, and the effort must be ongoing to keep not only the sites current but the certification of reputability and reliability also current.

1.1.2 What Kinds of Things Can We Say?

Now we have provided our agent with a list of sites to consult, we need to consider what we want our agent to say to them. Clearly, the first thing we want is for the agent to find out if the bookseller has the book we want, and at what price and delivery time. If the agent decides to buy, then it needs to be able to order the book. We can call the former pair of request and response a *price and availability request* and *price and availability response* respectively, and the latter pair an *order* and *order acknowledgement* respectively. Any functioning bookseller site will have facilities for these things, but they will differ greatly among themselves in how these functions are accessed. The problem for the agent is to know how to say what it needs to say to the target sites, and how to interpret the responses it gets.

Again, there are several ways to do this. One is to write a program which interacts with the HTML forms on a given site. The program scans the marked-up text searching for keywords, entry fields and tables, providing the information requested by that site and giving the site's response to the agent in a uniform way. This sort of program is called a *wrapper*, and is the way many shopping bots are built.

There are problems with the wrapper approach. One is that the wrapper is brittle. If the wrapped site changes its web page, the wrapper ceases to function properly until it is updated correspondingly, and popular active sites often change their interfaces. A second problem is that some sites view their interface as intellectual property, and actively discourage wrapping, even to the point of pursuing legal remedies. These problems are essentially due to the directory site not having the cooperation of the individual target sites.

A third problem is the cost of maintaining the wrappers. The cost tends to put wrapping out of reach of individual users, making it more feasible as an additional feature of a directory site. The directory site provides either an application program interface (API) or a set of standard messages (structured as XML, say), for communication between the users' agents and the target bookseller sites.

If the difficulties of the directory site are largely due to not having the cooperation of the target sites, then a natural solution is to obtain cooperation. If a target bookseller site agrees to participate in the directory, then it makes sense for the target site to itself provide the API or standard XML messages implementing the directory's *request*, *response*, *order* and *acknowledgement* functions. The target would also agree to keep these constant for defined periods or to give adequate notice for changes. This eliminates the brittleness problem, and greatly reduces the cost of maintenance. A directory site working in cooperation with the target sites in this way is called an *e-commerce exchange*.

1.1.3 What Is in a Message?

Now we can send a *request* message and get a *response* from the booksellers, but what do we put in the request and how do we interpret the response? If we look at the web forms of various booksellers, they have different items of information on them. Often the same item named differently and in different formats on different sites. The exchange's standard messages have a given set of fields, which are used by the user agents. Sorting out the relationships between the fields on the bookstore web sites and the exchange's standard message is a major part of the complication of the wrappers as described above. If the bookstores participate in the exchange, they can agree on a common set of fields, described according to say a common XML schema, and can commit to providing at least a minimum subset of the agreed fields.

Using an exchange with standardized message types and schemas, it becomes a relatively simple matter to implement a purchasing agent.

1.2 What Resources We Need to Support an Agent

We have seen from the book shopping bot example that an agent needs to know three kinds of things:

- A trusted list of sources with which to communicate

- A standard set of types of messages with agreed semantics
- Each message having a standard schema of data items, with agreed interpretations

And that a good way to provide these things is by an exchange which maintains agreements with the sources and manages the standard message types and schemas.

1.2.1 Complexity in Exchanges

Our example was very simple – buying a book of an unspecified sort. If we look into the book trade, we will see that there is an enormous variety of types of books and an enormous range of specialties of booksellers. At one end, we have the universal booksellers like Amazon, Barnes and Noble, and Blackwells, which use the *Books in Print* as their catalogues and have agreements with all the major and most of the minor publishers. At the other end we have people who specialize in chess books or used comics or the occult. Several of the booksellers in Figure 1 are pretty specialized. There is also an enormous range of people buying books – after all, each of the specialized bookshops has its own clientele.

So if we think of the problem of billions of people operating bookbuying agents and buying all sorts of books new and used in dozens of languages all around the world, we see that we either need a large variety of exchanges with different specialties or for the exchanges to have a deep taxonomy of classes of publication, very likely both. If we have lots of specialized exchanges, then we probably need exchanges of exchanges organized into a deep hierarchy of classes.

Besides being more specialized, there is need to be more general. Some of the large sellers carry, besides books: videos, DVDs, CDs, magazine subscriptions and all sorts of other things. Each of these other sorts of product have their own range of more specialized outlets, and would have their own range of more or less specialized exchanges. The exchanges need to be organized into some sort of Yellow Pages directory in the same sort of way that physical shops are, with boutiques occupying the most specific classes in the Yellow Pages, and the department stores and supermarkets the more general classes. The general suppliers of course all have their own taxonomy of departments and subdepartments.

Our book-buying example was what is called business-to-consumer (B2C) e-commerce. At the time of writing, there is much more activity in what is called business-to-business (B2B) e-commerce, where both parties to a transaction are possibly large organizations, and where the products are far more specialized and the trades much more complex. B2B exchanges tend to be in specific industrial sectors and to include a complex structure of specialized products and services. An exchange supporting say home renovators in a particular city will have sources of building materials, hardware, machinery, spare parts, repair services, subcontractors and specialized sources of engineering, architectural, legal and financial services.

In summary, the trusted sources need to be organized into complex classification systems within exchanges, and the exchanges themselves need to be classified, with the whole organized into systems of Yellow Pages style directories.

1.2.2 Complexity in Message Types

Our example had four types of messages: *request*, *response*, *order*, and *acknowledgement*. Under the surface, things are much more complex here, too.

Consider first the details of the implementation of the order requested by an *order* message. Payment will be by credit card, so there needs to be a B2B exchange between the bookseller and the credit card company validating the credit card. Fulfillment of the order will require an exchange of messages with the warehouse, then a billing message to the credit card company with an acknowledgement. There may be a whole series of messages from the bookseller to the customer agent marking various stages in the order fulfillment and shipping process or explaining delays. The bookseller may maintain order state and support a set of message types allowing queries and responses on the state of orders placed by an agent.

B2B transactions tend to be even more complex. A relatively simple purchase transaction may involve a sequence of messages from an Electronic Data Interchange (EDI) system

1. *Request for quotation* (*RFQ*) issued by the purchaser, asking for price and availability of the desired product
2. *Quote* by the supplier, saying that the product is available at a given price within a certain time period
3. *Purchase Order* by the purchaser, accepting a quotation and promising to take delivery and ultimately pay for the product
4. *Delivery Advice* by the supplier, saying that the product ordered has been delivered
5. *Delivery Acknowledgement* by the purchaser, agreeing that the product has been received in good order
6. *Invoice* by the supplier requesting payment for the delivered order within a specified time
7. *Payment* by the purchaser of the amount agreed in the purchase order and due in the invoice.

In practice, of course, the interaction can be much more complex, involving many more exchanges of different types, and the sequence need not be linear.

1.2.3 Complexity in the Content of Messages

All of the messages involved in the interoperation between the sites contain many fields. The same information is often repeated in many messages. For example the product details would generally appear in the *RFQ*, *Quote*, *Purchase Order*, *Delivery Advice*, *Delivery Acknowledgement* and *Invoice*. Not only that, but the information in the messages is often copied from the tables in the information systems supporting the activities of both organizations. Besides appearing in the messages, the product details would generally appear in the same or closely related form in the *Product* tables of both organizations. Note that establishing the correlation between the product tables of the two organizations requires its own series of exchanges. The identity and many details of both partners similarly appear in the messages and in the tables, as do quantities, dates and prices.

Having the same information appear repeatedly is redundant. However, this redundancy is important when two different organizations are involved, in order to make sure that the subject of the communications is what both organizations think it is, to prevent misunderstandings, and to enable audit. *Product* information appears repeatedly because the different messages are saying different things about the same product, and there are often many exchanges of messages going on at the same time so we need to be able to keep track of what is about what. We need to make sure that the

amount paid is related to the price in the invoice, which is related to the price in the purchase order, which is in turn related to the price in the quotation. Messages often refer to other messages, too. The payment refers to the invoice, which refers to the delivery acknowledgement, which refers to the delivery advice, which refers to the purchase order, which refers to the quotation.

In other words, the messages exchanged say things about a number of more or less complex objects that exist independently of the messages. Further, the messages themselves are objects which other messages can say things about.

1.2.4 Complexity in Who Can Send and Receive Messages

Messages do not exist in a vacuum. Messages must be sent and received by agents. Further, it is often required that the sender and receiver of a message take specific roles in the exchange. For example, a request for proposal message must be sent by an agent taking the role of a purchaser to an agent taking the role of a supplier. Further, the same agent can often take different roles in different messages. Companies A and B may purchase from each other. Say A supplies stationery and B supplies transport services. So in some RFP messages the purchaser is A and the supplier is B, while in others the purchaser is B and the supplier A. So besides the concept *role* we need a concept *player*, which designates the agents authorized to take roles in the exchanges. Agents A and B are both players in our example. Players often must be registered by the authority running an exchange.

The concepts of player and role will be elaborated in the next Chapter when we introduce the concepts of speech acts and institutional facts.

1.3 Plan of the Book

The book is divided into four parts. The first, Chapters 1 to 4, concerns the problem ontology is meant to solve. Chapter 1 has been a discussion of some of the problems encountered in developing an exchange of interoperating agents. Chapter 2 brings out some of the characteristics of the objects and actions of interoperating systems using John Searle's theory of speech acts and institutional facts. Chapter 3 describes the key barrier to interoperating among autonomous systems, namely semantic heterogeneity, and shows the role of standards. Chapter 4 contains a detailed description of six specific ontologies which are used as examples in the remainder of the text.

The second part is an examination of the functional requirements for an ontology. One requirement is for representation of complex objects, the subject of Chapter 5. A second is a rich subclass structure, discussed in Chapter 6. Chapter 7 describes two formal upper ontologies which provide a rich set of concepts which enable more robust and complete ontologies. An ontology is a designed artifact, so some useful design principles are presented in Chapter 8. A sketch of the enormous variety of applications of ontology is given in Chapter 9.

Part three looks at some ontology representation languages: RDF in Chapter 10, OWL in Chapter 11, use of OWL to represent more advanced concepts in Chapter 12, Common Logic in Chapter 13 and Topic Maps in Chapter 14.

Finally, in Chapter 15 we consider some of the requirements for an ontology server, the software environment used to develop and deploy ontology.

1.4 Major Exercise

This major exercise is intended to help the reader gain an understanding in a practical situation of the main principles of ontology from parts one and two of the text. It is to be done in groups of up to three.

1. Find a published ontology of some kind. No two groups should have the same ontology.
2. Identify three distinct players who interoperate using this ontology, and three distinct roles that can be taken in the interoperation. Briefly describe what each role does and how the players interact using the ontology.

   Show three concrete actions taken by players as they interoperate. Include details of players and roles involved, and the contents of the messages.
3. Briefly describe the ontology, giving enough detail to understand the elements of the ontology involved in the actions of 2. You should include enough concepts to get an idea of the scope and content of the whole ontology. Include some instances of all classes, including all instances used in the actions of 2. Your description should be supported by a diagram in the representation language of Chapter 4, with between 25 and 50 classes.
4. Describe three institutional facts created in this network, including brute fact and context. What roles are responsible for creation of each fact? Who is responsible for keeping the definitive record of the fact? You may use the actions of 2.
5. Describe two shared complex objects. What are their parts? What are their identifying and unifying relations?
6. Describe a bulk class in the ontology, showing how it satisfies the criteria to be a bulk class. If there is none, propose a plausible addition.
7. Describe the three classes in the ontology from 5 and 6. Give their rigid properties, indicating whether the rigid properties are lexical or logical. Show a system of subclasses for each class, inventing a plausible system if necessary. Each subclass is either defined or declared. If it is defined, give the defining predicates. If it is declared, tell how objects are classified into the subclass and by which role.
8. Describe a property (relationship, association) involving at least one of the classes from 7. This property should have a subproperty structure. Invent a plausible structure if necessary. Show a population of property instances, including at least one instance of each subproperty.
9. Describe in detail two endurants and two perdurants in the ontology. (If there are no perdurants, invent two plausible ones). What endurants participate in the perdurants? How are the histories of the endurants represented in the ontology?
10. Criticise the ontology in terms of the five principles of Gruber.
    1. Clarity: suggest a plausible unintended interpretation of one of the concepts. How does (or could) the ontology prevent that unintended interpretation?
    2. Coherence: suggest a plausible inference that an agent could be expected to draw from the ontology. How does (or could) the ontology support the reasoning necessary to make the inference?
    3. Extendibility: suggest a plausible extension to the ontology. Show what changes would need to be made. Are any of the changes redundant? If so, show how. If not, show how the ontology design anticipated the extension.
    4. Encoding bias: Show how one of the actions of 2 is implemented. Does it make sense for it to be implemented in a different way? If not, why not? If so,

does the ontology make the different implementation difficult? Consider each element of the implementation of the action.

5. Ontological commitment: Describe the particular system of interoperations the ontology is intended to be used for (this could be a class of systems, with multiple separate instances). Is there a different system of interoperations in which the ontology could be reused? If so briefly describe the different system and how the ontology could be adapted to the re-use. If not, why are there no similar systems?
6. In each quality dimension, indicate whether in your judgment the quality is high or low.

11. Propose an improvement to the ontology. Argue why this improvement is a good idea in terms of at least one of Gruber's principles. Show a cost of the change. On balance, is the improvement a good idea? Take a position and justify it.

Note: The object of the exercise is to demonstrate your understanding of the concepts and principles of ontology as contained in the course material. Since engagement with the detail is essential for understanding, descriptions should be detailed enough to see what is going on. For example a simple *create(object)* is not sufficient. The description should include under what conditions the *create* is possible, who sends the message to whom, and what information is contained in the message.

1.5 Further Reading

The Wikipedia entry for the semantic web is worth a look: http://en.wikipedia.org/wiki/Semantic_Web.

The semantic web started with [1].

A good look at the organization of an e-commerce exchange is in [2].

The introduction to a new journal, *Applied Ontology* [3], gives a good perspective on what ontology is about and how it relates to computing more generally. The article is short, and the most relevant material is in the first two pages.

2 Institutional Facts: Making Sense of What is Going On

Interacting with Internet services and interoperation among Internet services involve more or less complex communication about things which exist in some sort of reality more-or-less independent of the communication. In fact, as we shall see, the information systems often create aspects of the reality they share.

2.1 What We Talk About in Information Systems

Most of our intuitions about communication involve talking to other people about fairly concrete physical objects, say buying bananas in a fruit shop. We get lots of help from the physical world in such communication. Even if we are in a country where we don't speak the language, we can point to the bananas. The fact that we are in the fruit shop and pointing to the bananas tells the shopkeeper that we want to buy some. We can hold up fingers to say how many we want.

Physical objects have many properties we can use to fine-tune the communication. We can say that the bananas in a proposed bunch are too big, or too ripe, or that the end one has a bruise.

Communication with an information system, or between information systems, is much more limited. We can communicate only using pre-defined message types. We can say only what can be indicated by choosing pre-defined possible contents of pre-defined fields in the messages. The objects themselves have a funny existence, because we can only know about them what is represented in the tables in our information systems. There is no pointing, no seeing, no touching. Furthermore, much of what we communicate about is the result of previous communications.

We need to develop some understanding of the special features of this kind of communication about these special kinds of objects. We will make use of the philosopher John Searle's concepts of speech acts and of institutional facts.

2.1.1 Brute Facts, Speech Acts and Institutional Facts

Searle distinguishes two types of facts, *brute facts* and *institutional facts*. A *brute fact* is about something in the physical world that is independent of human society, while an *institutional fact* is dependent on human society. Suppose a truck turns up one morning and dumps 10 tonnes of mushroom compost in your driveway. This is a brute fact. The driveway would be there and so would the pile of mushroom compost, even if suddenly all human beings disappeared. Suppose further that on the previous day, you had sent a message to the landscape supply company ordering 10 tonnes of mushroom compost. In this circumstance, the pile of compost constitutes the delivery of your order. The delivery of an order is an institutional fact. Without human society, there would be no landscape supply company, nor you for that matter, and the truck would never have arrived and deposited the compost. Further, part of the meaning of the delivery of your

order is that you are now obligated to pay the landscape supply company an amount of money in exchange for the compost and its delivery. Without human society, an obligation to pay has no meaning.

Searle uses a formula "(brute fact) X counts as (institutional fact) Y in context C" to organize the relationship. In our example, the brute fact X is the truck dumping the compost in the driveway. The institutional fact Y is the delivery of an order. The context C in this case is your previously having placed an order for that amount of compost.

Your placing the order is called a *speech act*. A *speech act* is something that is said which changes how the world is. The term "said" is used in a very general sense – you could have called by the landscape supply company and placed the order by speaking to a clerk, or telephoned, or sent a message to their web site as in the previous chapter. In the same way that an institutional fact is a brute fact in context, a speech act is a physical action in context. We can say that a physical action X counts as speech act Y in context C. An institutional fact is a record of a speech act having been made.

You can see how your placing an order changes the world by considering what would happen if you had not placed an order, but still a truck arrived and left 10 tonnes of compost in your driveway. In this case, instead of you having an obligation to pay for the compost, the landscape supply company has an obligation to return, clean up the pile, and possibly compensate you for any damage caused. When you make the speech act of placing an order, the institutional fact of your having placed the order becomes true, which constitutes a change in how the world is.

Note that the "X counts as Y in context C" formula applies here, too. The brute fact is your sending a message. The institutional fact is your having placed an order. The context includes that you are an account customer in good standing of the landscape supply company, that the company is in fact in business of selling mushroom compost and that your location is within their delivery zone. Finally, note that the delivery is also a speech act, in that it places you under an obligation to pay. In practice the delivery is generally accompanied by an invoice, but not necessarily. The driver may simply say "here is your compost" and expect you to hand over some cash. It may help your intuition to imagine that a second truck arrives an hour later and leaves another 10 tonnes. This second truck is not a delivery but a mistake, leaving a mess that the company is obligated to clean up.

A simple way to think about the distinction between brute fact and physical action on one hand and institutional fact and speech act on the other is to imagine a biologist observing the behaviour of the people involved (or the operation of the programs). Assume the biologist is an extra-terrestrial, and does not necessarily even know that the humans have any language, or that the programs are exchanging messages in a language. The biologist sees people making noises with their mouths, making marks on paper, carrying things around, exchanging them, making movements with their hands, being in particular places, wearing particular costumes, and so on. They see the programs sending bit streams with various patterns and lengths to other programs, movements of machinery, and so on.

The brute facts and physical actions are what the biologist sees. The institutional facts are how the world of humans is different afterwards, and the speech acts are what makes the change.

The context of a speech act includes that the agents participating in the making of it by properly constituted players taking roles in the speech act which they are authorized to take. In some cases, the players and roles are highly regulated, and in

other cases less so. For example, in some places street vendors are unregulated, so any person on the plaza can take the role of purchaser and anyone else can take the role of supplier. In contrast, marriage in many cultures is highly regulated. The players taking the roles of bride and groom must have a license which establishes their legal age, lack of impediments to marriage, compatible blood types, and of course gender. The celebrant is authorized by the state to play that role, and has a certificate indicating that they have passed the requisite conditions. The marriage is valid only if all the players are authorized, and each is authorized to take the role they take. The appropriate context for a speech act is often partly codified as *framing rules*.

Note that a player is any entity capable of taking action, an agent. A player can be a person, as in the marriage example above. A player can be an organization. Think of a University granting a degree. The speech act is made as a composite of several actions taken by people acting in organizational roles, so the act is thought of as being made by the University, rather than any particular person. A player can be a machine. An obvious example is a vending machine, which makes the speech act of selling something resulting in the institutional fact of your owning the object. On-line e-commerce sites like Amazon.com make speech acts in the same sort of way.

Information systems are almost exclusively concerned with storing institutional facts. Most messages between information systems are speech acts. The information systems' business rules enforce the context rules determining the validity of the speech acts, and the systems themselves keep track of how the world changes as a result.

Information systems rarely have anything to do with brute facts in themselves. In the delivery of the mushroom compost, the information system (operated by the garden supply company player acting in the role of supplier) records the order (issued by the gardener player acting in the role of purchaser), issues a dispatch notice to the shipping department, and records the receipt of the payment. The truck and pile of compost are (partly) controlled by the information system, but are generally thought of in themselves as outside the information system.

Returning to the examples in the previous chapter, the series *RFQ*, *Quotation*, *Purchase Order*, *Delivery Advice*, *Delivery Acknowledgement*, *Invoice* and *Payment* are all speech acts, and the business of the information systems is to record these acts as having been made and keep track of the consequent changes in the way the world is. The book shopping bot actually makes the speech acts *Query* and *Order*. The selected bookseller site makes the speech act *Response* (a type of quotation), and the speech acts involving the credit card company and the shipping department. The record of the state of the order is a record of the consequent institutional facts.

Almost everything in an information system is a record of an institutional fact. The fact that someone is a customer (stored in the *Customer* table) is an institutional fact. The customer's name is an institutional fact (created in a speech act by the person's parents). The customer's credit rating is an institutional fact created in a speech act by the company's accounting department. There may be a bin with mushroom compost in the yard, but the fact that the bin contains "Grade B spent mushroom compost" is an institutional fact created in a speech act by the marketing department, as is the fact that the price is so much per tonne.

2.1.2 Interoperation Requires Ontology and Background

We have established in the previous chapter that the interoperation of two information systems requires information structures that are outside either system, namely a

taxonomy of sites, a taxonomy of message types (which we now recognize as types of speech acts), and a taxonomy of the contents of the messages (which are institutional facts) including the contents of the two information systems (also institutional facts) which form the necessary context for the institutional facts reflected in the messages.

What we are talking about is a description of a collection of things which exist in the environment of the interoperating systems. Such a collection is called an *ontology*. Associated with the description of each thing is an agreement about the semantics of that thing, how it is to be interpreted when it is used in a message. These agreements are called by Searle *background*. They are not necessarily articulated in a formal way, and the participants may not even be consciously aware of them.

For example, what do we have to know to know how a fruit shop operates? We know that the bananas on display are for sale, not as biological specimens or as displays of what ripening fruit looks like. We know that buying bananas involves handing over bits of currency in exchange for the right to carry away some bananas, and that we can't carry away any bananas, or eat any, without handing over bits of currency. We know whether the shopkeeper insists on picking out the fruit we buy, or whether we are allowed to select our fruit ourselves. (If we are in a strange place, we may need to observe others buying to find this out.)

In the bookseller's site, we know that when we enter the author and title in the fields so labeled and press the click button when the mouse pointer is over the picture of the magnifying glass on the screen, that the screen will soon refresh with a price and availability message. We know that this message tells us that if the book is in stock we can buy it for the price indicated. We know what obligations a seller incurs if they send a *quote* message in response to an *RFQ* message from a potential customer. We know what the no-frills airline means when it says each passenger has a 15 kilogram baggage limit, that is whether it enforces the limit closely, or whether it well accept a 19 kilogram bag without levying the extra charge stated in the documentation.

All of these things are outside the systems in question. Some of them are more or less written down, but many are not. Even if written down, as in the case of the airline, how the regulation is interpreted in practice may not be articulated.

So besides an ontology of the things existing in the environment of the two systems, the interoperating agents need to share an understanding of the background behaviors and practices underpinning the operation of the systems involved.

2.2 How an Institutional World Operates

Now that we have an idea of what institutional facts are, we need to understand how an institutional world operates.

Everyone has had experience in playing games. A game is an example of an institutional world. Nothing that happens in any game has any meaning outside of human society. So an analysis of how a game works in terms of institutional facts and speech acts should help to clarify the operation of institutional reality.

First of all, games have rules. Everything that is possible, permissible or forbidden in a game is specified in its rules. The rules are institutional facts, created either by tradition as in chess or by a governing body for the game, as in soccer. The rules are said to be *constitutive* of the game: they define the world in which individual games are played.

Games of the sort we are considering have *players*. A chess game has two, let us say Spassky and Deep Blue. A soccer game has also two players, say Manchester United and Real Madrid, but the soccer players are teams consisting of 11 individual players, which we can think of as parts of the team. The parts have names: goalie, striker and so on. So players can be either simple or complex. The number and structure of the players is specified in the rules.

Players generally take *roles* in the game. In chess, one player takes the role of white and the other of black. In soccer, the teams play roles like attack and defense. Moreover, the parts of the teams take roles in the individual actions that make up a game. For example the goalie can play the role "catch" in an action where an opposing striker plays the role "kick".

An individual game or game instance is a world structured by the rules which accumulates institutional facts as the game progresses. Each new institutional fact is an instance of a type of institutional fact permitted by the rules, established by a speech act. In chess, each speech act is a movement of a piece, sometimes accompanied by the removal of an opposing piece from the board. One exception is the speech act "resigning" in which one player declines to move allowing the other to be named the winner.

Some kinds of institutional facts are not specified in the rules, but are used in commentary on a game. They are descriptions of states of the game. In chess, there is a standard set of openings "Ruy Lopez", "Sicilian Defence" etc. There are concepts like "control of the centre", "extent of development", "end game". There are names of tactics like "fork", "pin", "discovered check". These kinds of institutional facts are derived from the rules. This commentary is used to describe the state of a game instance as it progresses.

The brute facts are the physical actions and objects which when performed in the proper context constitute moves in the game. A piece of wood carved into a shape of a horse counts as a "knight" in the context of a chess game. Picking up a piece and putting it down again on the board counts as a move in the context of the person picking up the piece and putting it down being a player whose turn it is and the displacement is legal under the rules.

Background in a game consists in understanding turn taking, winning and losing, fair play and a host of other things.

Further, many games have additional players whose role is to either enforce the rules or to adjudicate whether a particular action counts as a significant speech act. Think of the referee in a soccer game, or the touch judges in tennis. We will refer to these sorts of players generally as referees below.

2.2.1 Performatives and Informatives

So in general, speech acts made by certain players create institutional facts. Some players are authorized to create certain kinds of institutional facts, while other players are authorized to create other kinds. Since we are operating in an information systems world, any player who needs to know about a change in the institutional reality created by another player must use a message to do so. This situation gives rise to two different kinds of speech acts. A *performative* speech act creates an institutional fact, which is a change in the shared reality of the community. An *informative* speech act is used by players to find out the current state of the shared reality.

For example, the community in a university involved in courses, enrolment and grades is a loosely coupled institutional reality. The community includes the Faculties who are responsible for course offerings and student admissions, the students who are responsible for enrolling in courses, and the lecturers who are responsible for assessment. Some of the information is held in the central student records system, but much is held in idiosyncratic spreadsheets in the workstations of lecturers and students.

The ontology is that there are students, courses and enrolments (association between student and course). An attribute of enrolment is grade. In the lecturer's spreadsheet is recorded the various components of the grade in terms of the results of various assessments.

Let us assume that students maintain spreadsheets containing their record of study (enrolments and grades). Assume that the Faculties, lecturers and students share the schema involving students, courses, enrolments and grades; and that the students and lecturers share the schema involving grade components.

Instances of students and courses are created by the Faculties. The speech act of enrolling a person in a degree plan, thereby creating a student, involves the student making an application, submitting in addition supporting documents and making payments; the Faculty assessing the application against its criteria and intake quotas then sending a letter of offer; and finally the person accepting the letter of offer and enrolling in courses. All of these individual steps in the speech act creating a student are also speech acts, and the documents submitted are generally records of institutional facts. The Faculty's accepting the person as a student is among other things a judgment that the context is correct. The background for this complex speech act includes literacy, knowing how to operate in a semi-bureaucratic environment, understanding deadlines, and many other things.

Creation of a course offering is a similarly complex process. A subcommittee within a Department prepares a proposal for the course, which is submitted to and approved by first the Department's curriculum committee then the Board of Studies for the degree program in which the course will be offered. The proposal is then submitted to the Faculty, where it is subject to further decisions until it is inserted in the course catalog for the appropriate year. A new course now exists. Similarly to the creation of a student, the process is made up of numerous intermediate speech acts and the documents submitted are records of institutional facts. The background necessary is similar, also.

Instances of enrolments are created by the individual students, by entering forms on the University's web site. The context includes the person being a valid student and the course being a valid course. The evidence of validity in this case is the ability of the student to access the web site and to see the course among the list of offerings.

Values of the grade attribute are created by the Faculties on advice of the lecturer also via a process with several steps. The lecturer creates the values of the individual assessments making up the grade.

All of these speech acts are performative, and create the institutional facts making up the formal educational world.

The letter of offer of a student place to a person is an informative speech act. Lecturers must find out the students enrolled in their courses (class lists) by another informative. The Faculties must find out the grades advised by the lecturers in informatives. The students must find the grades awarded by the Faculties and the individual assessment reports awarded by the Lecturers by still other informatives.

Note that some of the informatives are initiated by the agency making the speech act (letter of offer of a place, grade recommendation by the lecturer), while some are initiated by the agency wanting the information (class list enquiry, grade enquiry). The former kind of delivery of an informative is called *push* (the information is pushed out by those who create it) while the latter in contrast is called *pull* (the information is pulled in by those who want it).

All of these speech acts are made possible by the overall structure of institutional facts which is created by some even more remote body, such as by an Act of Parliament followed by decisions of the University governing bodies at the highest levels (which create Faculties and assign their powers and hire Lecturers and assign their powers).

So we have an ontology of institutional facts which pre-exists at the type level, acting as a sort of playing field. At the instance level institutional facts are created by various parties, with the information being propagated to the other parties in various ways. Aspects of the world are seen as fixed by those parties who cannot create them. To the student, their program enrolment and possible course selections are solid reality. To the lecturer, the course they are assigned to teach and the class list are solid reality. But to the Faculty, students, courses and lecturers are variable parts of reality which it is the Faculty's business to maintain.

Refereeing in this university world is done by various bodies. There is a right of appeal for marks and grades which begins with asking the lecturer through to appeal the University Senate. Plagiarism is dealt with by the Department with referral to the more central agencies in the University. For certain situations, there may be appeal to bodies outside the university – an ombudsman, say, or to the police or to the civil courts.

In a similar way, the world of book shopping is an institutional environment. The various players are authorized to make particular classes of speech acts thereby creating instances of institutional facts. The booksellers create their catalogs and prices, which are fixed reality to the customers. The customers place orders, which are fixed reality to the booksellers. Listing in the trusted list of sellers is created by the agency maintaining the list, as are the types and formats of the messages by which the system operates. All of these aspects of reality are created by performatives. A customer's search for an appropriate bookseller and subsequent query of price and availability are informatives, as is the message from the bookseller to the customer telling the price and availability. This whole world operates in a background of knowing about buying and selling, supported by the institutional facts of laws and regulations governing advertising and contracts enforced by the police and courts, who play the roles of referees in this game.

2.3 Key Concepts

An **Institutional fact** is a record of a **speech act**. A **Brute fact** X counts as an **institutional fact** Y in **context** C. Context includes **framing rules** and **background**. Most information systems manage records of speech acts. Interoperating agents perform speech acts, both **performative** and **informative**.

2.4 Exercise

The exercises in this book are based on the same systems domain, that of IT support for the Beijing Olympics in 2008. Since there will be many exercises based on different aspects of the domain, it may be worthwhile to invest some time looking at the problem. Useful and easily accessible resources include a Wikipedia entry <http://en.wikipedia.org/wiki/2004_Summer_Olympics> which has links to other summer and winter Olympics entries; and the official International Olympic Committee's site <http://www.olympic.org/uk/index_uk.asp>. For a look at the vast scope of the Olympics as an enterprise apart from sport, see the official report of the Sydney Olympic Committee <http://www.gamesinfo.com.au/postgames/en/volume_en.htm>.

1. Consider the following examples of institutional facts. What are the corresponding brute facts? In what context does the brute fact count as the institutional fact (consider both framing rules and some of the background)? What is the speech act of which the institutional fact is a record, and who makes it?
   - An Australian 1 dollar coin
   - Your drivers licence
   - Your citizenship
   - How you are legally able to reside here to study in this University
   - A goal in soccer.
2. (relevant to points 2 and 4 of the major exercise) Consider a hypothetical Olympics, sketched below. You can add details from your general knowledge or the result of research into the Olympics.

   There are sports, including swimming and athletics. Each sport has an international sporting federation. Swimming has FINA, athletics IAAF. At the Olympics, each sport organizes a number of events drawn from the events recognized by the corresponding federation. Events are open to either men or women competitors. Some events in Swimming are 100 metres butterfly, 400 metres freestyle, 1500 metres freestyle, and 400 metres medley relay. Some events in Athletics are 100 metres, 400 metres, 1500 metres and 400 metres relay. In some cases competitors are individual persons, while in others (the relays in this case) competitors are teams of individual persons. Competitors are organized into national teams, each organized by the country's national Olympic committee (NOC). A national team consists of a number of competitors and a number of officials. A particular Olympic Games is organized by a Local Organizing Committee (LOC), under the auspices of the International Olympic Committee (IOC).

   An event is organized into a number of sub-events (heats, semi-finals) intended to reduce the number of competitors in the final sub-event. For the two sports considered, we can call these sub-events races. Each race includes a number of competitors and a number of officials. At the end of the race, the competitors complete in a time and achieve a finishing position. In the final race, the first three finishing position competitors are awarded medals: gold, silver and bronze. There are Olympic records for times of races in each event, and also world records. Olympic records are recorded by the IOC and world records by the respective sporting federations.

It is usual to aggregate results in events by national team, so that during the Olympics a record is kept of the number of medals won by competitors of each national team, with subtotals for gold, silver and bronze. Each NOC maintains its own totals on advice from the LOC of results in events.

a. Identify three distinct players who interoperate using this ontology, and three distinct roles that can be taken in the interoperation. Briefly describe what each role does and how the players interact using the ontology. Show three concrete actions taken by players as they interoperate. Include details of players and roles involved, and the contents of the messages.
b. Describe three institutional facts created in this network, including brute fact and context. What roles are responsible for creation of each fact? Who is responsible for keeping the definitive record of the fact?
c. Describe two performative and two informative speech acts performed in this system.

2.5 Further Reading

Institutional facts were first described [4], especially chapters 2 and 6.

3 Semantic Heterogeneity

If all these information systems are accessible via the web, then we should be able to more-or-less automatically integrate them. This was the original dream of the semantic web. If everyone annotates their web site using the standardized metadata language RDF, then everyone can interoperate with everyone else. There have been attempts to integrate autonomous systems by having each system autonomously providing something like metadata. None of these attempts have been successful, and the purpose of this chapter is to see why, and to see how exchanges can feasibly be created.

3.1 Federated Databases

The most significant effort to figure out how to do a bottom-up integration of autonomous information systems was the federated database movement of the late 1980s and early 1990s.

Federated databases start from the extreme success of database technology, especially relational database technology, in automating the information systems supporting coherent organizational subunits. There are millions of systems doing order-entry, inventory management, accounting, student records, payroll, project management, credit cards, airline reservations and many other functions.

Although a few systems were integrated, for example order entry/inventory/accounting, most organizations had (and have) many autonomous systems. A major hospital will have dozens of systems, a large company hundreds, and the government of a large country thousands. With the increasing accessibility of these systems via communication networks, people began to want them to talk to each other.

Relational database technology gets its power from the ability to separate the data schema from its implementation, so that the same schema can be implemented on one or many servers, for example, without the user needing to know anything about the implementation. This facility of allowing the user to focus on the schema without having to know about the implementation is called *transparency*.

So the proposal was to create a single organization-wide schema, called a *global schema*, allowing the data to be stored in the original autonomous *local* systems, with transparent access via a layer of software called *middleware*, which provides access to the component systems' schemas and transports subqueries and responses.

And since all the communication networks can be interconnected, the same approach was taken to bringing together information systems across organizational boundaries, so there is no particular reason why the local systems should be organizationally connected with each other. Some organization needs to maintain the federation, including the global schema and the middleware platforms, but the individual organizations do not need to be organizationally connected with it. Imagine a system that gives uniform access to the public aspects of the student record systems of all the universities in a country, for example.

The global schema must be represented in a single data model (e.g. relational, object-oriented), which is called the *common data model* for that system. The local

systems each have their *local schema* expressed in their *local data model*. To construct the global schema, it was proposed to first convert the local schemas to the common data model, yielding a collection of *component schemas*. Since not all of the functionality of a particular local system might be relevant to a particular federated system, an export *schema* would be created for each component containing the views showing the data participating in the federation. The global schema is the union of the export schemas. Finally, each application would see a collection of views on the global schema, called an *external schema*. This system of local, component, export, global and external schemas is called the *five-schema architecture*.

It is important to remember that each of the local systems was expected to retain its autonomy, so that the expectation was that the global schema could be constructed without modifying the local schemas. The component schemas were expected to be constructed as views on the local schemas, and the union of the component schemas into the global schemas might need to be augmented by additional views.

So the main technical problem was thought to be constructing views. Some views are simple: the same thing is called by different names - a company may be a customer in an order entry system and a supplier in a purchasing system; different things may be called by the same name – the table *dept* in one system might be departments, but departures in another. More complex are where the same thing is represented by different units in two systems, say an amount by US dollars in one and by Euros in another, or by feet in one and metres in another.

Then there are various structurally different ways of representing things: a subclass in one system may be a relationship in another; through to structural differences that cannot be expressed in SQL, such as instances in one system being table names in another. The classic example of the latter has one system representing stock prices in a table with schema (stock, date, closing price), while another system represents the price history of each stock in a separate table.

Differences in representation among systems is called *semantic heterogeneity*. The sort of heterogeneity that can be resolved by views in the ways sketched is called *structural semantic heterogeneity*.

Not all heterogeneity can be resolved. For example, system A may have an attribute *shipping weight* with two values, *heavy* and *light*, where *heavy* is greater than one kilogram. System B has an attribute with the same name and values, but in system B, *heavy* is defined as greater than 2 pounds. Some conversions are possible: *light* in B is *light* in A; *heavy* in A is *heavy* in B. Others are not. B's *heavy* could be either *light* or *heavy* in A, and similar for A's *light*. Heterogeneity which cannot be resolved is called *fundamental semantic heterogeneity*.

3.2 Semantic Heterogeneity

Structural semantic heterogeneity can be resolved using views, so preserves the autonomy of the local systems. Fundamental semantic heterogeneity requires that at least one of two conflicting systems must be changed, so violates autonomy. The bottom-up approach to federating databases outlined above depends on autonomy. If fundamental semantic heterogeneity is exceptional, then the methods can be adapted, but if fundamental semantic heterogeneity is usual, then the bottom-up method fails.

In the bottom-up approach to federating databases, the principal technical issue is the construction of the views in the component and global schemas, and the primary

responsibility for constructing the views is borne by the system integrators. If the bottom-up approach fails because of fundamental semantic heterogeneity, the alternative is to start with a global schema, then require each local system to modify itself to conform to the global schema. The responsibility for making the necessary changes now falls on the local systems. Since they must change in any case, it is often simpler for them to change their local ways of doing things so as to make it easier to integrate with the federation. Complex views become less important.

However, there are situations where the cost of making a local change is prohibitive. Consider the shipping weight example above. A simple way to integrate the two systems is for the global schema to adopt the definitions of system A, so that *heavy* is defined as weighing more than one kilogram, so system B must change. But the definitions for system A are derived from the European Union postal rate schedule, while those from system B are derived from the United States postal rate schedule. In order for systems A and B to federate in the way proposed, system B would lose integration with its postal system, which is a major disadvantage.

An alternative would be for both systems to publish the weight in their local units, kilograms in system A, pounds in system B, together with the thresholds used to calculate shipping costs. The federated system would use kilograms, so system B would have to do the appropriate conversions. This might be feasible.

But consider the following statistical application, where a proposed federation will monitor the total amount spent on computer equipment year by year from two different countries, A and B. Both countries maintain databases showing amount spent in various industrial categories. The semantics of domain *B* are consistent with the proposed global schema, namely computer equipment, but the semantics of domain *A* is expenditure on computer together with communications equipment. The attribute in *A* is more inclusive than in *B*. On the other hand, its domain might be less inclusive: it might be recorded only for organizations of at least a certain size. It should be clear that no view could resolve this semantic heterogeneity.

We look at the possibility of changes to make the two agree. Domain B might achieve its coverage as a by-product of its customs and import duty policies, while domain A must rely on surveys. To change either so that its data could be exactly related to the other would be prohibitively expensive, and almost none of the costs would be attributed to technical changes in the respective information systems. So a federation is just not feasible.

So fundamental semantic heterogeneity can not only make the bottom-up approach to federating databases unworkable, it can make federation by any means infeasible. How prevalent is it?

3.3 Semantic Heterogeneity Is the Norm

Because of the nearly universal experience of the difficulty of building integrated systems, we want an understanding of semantic heterogeneity that shows it to be the normal state of affairs, something to be expected. Its opposite, semantic homogeneity, is something strange, needing explanation. A system is heterogeneous until proven homogeneous.

Perhaps if we step away from computer systems into the physical world we can tie the computer concepts of the previous section to our everyday lives, thereby getting a better feeling for them.

Consider greeting cards. There is a large number of greeting card suppliers. It is plausible to try to build a federation of greeting cards. If we had such a federation, its global schema might look something like the UML diagram in Figure 2.

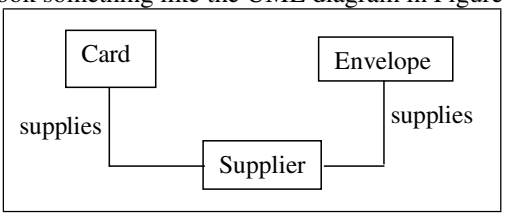


Figure 2. Global schema for greeting card federation

The class *Card* would be divided into subclasses along several dimensions:

- *Occasion*: birthday, holiday, anniversary, get well, friendship, etc.
- *Style*: traditional, art, humorous, risqué
- *Size*: various categories

The class *Envelope* would be divided into subclasses also, but along a single dimension *Size*.

From the last section, we recall that the whole point of a federated system is that it can be examined transparently, without taking its sources into account. So we should be able to look at the *Card* and *Envelope* classes without regard to the *Supplier* class, until we need to go to the supplier to make a purchase. And in fact, the Card class works fairly well this way. Many greeting card retailers do organize their displays along these dimensions, and will often mix cards from different suppliers in the same section of the display.

Size is an exception, though. Each supplier makes cards of a variety of sizes. Each supplier also makes envelopes of a variety of sizes. If the federation were working, it would be possible for one to choose a card, note its size, then go to the envelope display and select an envelope of the appropriate size without regard to supplier.

But different suppliers use different sizes, so that the aggregate number of size classes in the federation for cards is very large. Similarly, each supplier produces envelopes corresponding to the sizes of its cards, so the number of size classes for envelopes is also very large. Furthermore, we generally find that the envelope suitable for a card of a given supplier is also an envelope of the same supplier. So much so that greeting card retailers generally will place corresponding envelopes with each card, usually of the same supplier. So size is semantically heterogeneous, because size is not transparent. We need to know the supplier in order to see the relationship between card and envelope size.

Similar semantic heterogeneity is present in all sorts of other areas.

- There are many low-voltage electronic appliances: mobile phones, cameras, scanners, and so on. Each of these needs a power supply transforming the mains power. If this domain were semantically homogeneous in the way batteries are, there would be a number of power supplies corresponding to different appliance power supply requirements and you could use the same power supply for different appliances with the same requirements. But this is manifestly not the case. Each appliance comes with its own power supply. Even if two power supplies have the same specifications, they generally have different connectors.

- Television remote controls all have about the same functionality, but each model has its own remote. So-called universal remotes are not federations of the individual models, but a union of them. The same with remotes for VCRs, DVDs, CD players and air conditioners.
- Mobile phone subscription and payment methods vary enormously in features among providers, so much so that it is very difficult to make price comparisons for a given service.
- Cars have similar parts, but they are semantically heterogeneous. A fuel pump for a Ford will not fit a Toyota.

But there is homogeneity, too.

- Remote control units may not interoperate, but they use the same symbols on their buttons which have the same functions. This is not an accident. There is a body, the International Electrotechnical Commission (IEC), which has developed a standard IEC 60017 *Graphical Symbols for Use on Equipment*, which has a catalog of about 800 symbols indicating various functions. Of course, in order for the symbols to be standard so also must be the functions. Manufacturers agree to implement the standard functions and to use the standard symbols for them.
- Electrical devices all plug into the same power outlets with the same plugs. There are national standards for electrical power supply, outlets and plugs.
- Telephone equipment all plug into the same telephone sockets, and all have a common core of telephone functionality. The telephone functionality is a standard maintained by the International Telegraph and Telephone Consultative Committee (CCITT), while the sockets and plugs are standardized by national standards bodies.

In all sorts of industries there are standards bodies which develop standard ways of doing things which are published and adopted generally voluntarily by individual organizations. In many cases the standard is set not by a standards body but by a single dominant supplier. The PC became a de facto standard when IBM entered the microcomputer market in the early 1980s. IBM had such market power that other manufacturers began to produce functionally equivalent machines, which could run the same software as the IBM computers. Eventually IBM withdrew from the market, but by that time there was a separation of computer manufacturers and software manufacturers, and no computer manufacturer could afford to change because potential customers would not have any software.

But the semantic homogeneity enabled by standards is limited to the scope of the standard.

- Electrical power is homogeneous within a single country, but among countries very heterogeneous. The US uses a system of 110 volt 60 cycle alternating current, while most of the rest of the world uses a system of 240 volt 50 cycle. A device designed for one system will generally fail to work on the other. Even within the 240 volt 50 cycle world, there are a variety of outlet and plug geometries. Travelers need to carry a collection of plug adapters.
- Telephone functionality is common throughout the world, since the CCITT is a world-scale standards body. Every telephone system signals readiness to accept a call by sounding a dial tone, for example. But what tone is used is not a CCITT standard. Dial tones sound very different in different parts of the world, so that for example computer fax software can fail because it doesn't

recognize a dial tone in a particular system. Similarly, the sockets and plugs differ among systems in the same sort of way as electrical outlets and plugs do.

In the physical world we expect heterogeneity. When we see homogeneity, we look for a reason, which is generally a standards body or a dominant supplier.

An organization acts as a sort of island of homogeneity. Remember an organization exists to make speech acts and therefore produce institutional facts. Within the organization the speech acts and framing rules are organized into a coherent system, so the institutional facts fit together. The system of institutional facts is therefore homogeneous.

We can see this in statistical publications, which are generally made by organizations, sometimes quite large, a Department of the Census for example. The Department will periodically collect a large amount of data, then publish perhaps thousands of cross-tabulations. These cross-tabulations are all homogeneous with each other. We can safely compare the average age per district with the average level of education per district.

The same is not true when statistics from different organizations are compared. It is difficult for example to get a clear comparative picture of unemployment among a group of countries. Within each country unemployment is counted in a particular way, consistent with many other statistical tables compiled by the relevant organization, but different countries have different framing rules to count a person as unemployed.

Homogeneity breaks down even within a single statistics-collecting organization, over time. In sporting statistics for example the performance records cannot strictly be compared before and after significant rules changes or changes in the game practice.

In the case of games, semantic heterogeneity is obvious. Within a given game, any player can interact with other players according to the rules. The rules, possibly enforced by a governing body, provide an island of semantic homogeneity. There are many games which are similar to other games, for example baseball and cricket, soccer and water polo, rugby league and rugby union, ice hockey and field hockey. We could try to federate these pairs of games using the techniques from the federated database movement described in the first section. If we did this, we would find pretty comprehensive correspondences. Baseball and cricket both have balls, bats, hits, runs, innings, outs, pitchers correspond to bowlers, and so on. Similarly for the other pairs. But it would be absurd for a baseball team playing baseball to interoperate with (play a game with) a cricket team playing cricket. Similarly it would be absurd for a team of one of the other pairs to interoperate with a team of the other. This is true even for very similar games like American and Canadian football (gridiron).

So if we keep in mind that institutional worlds are like games, we would expect semantic heterogeneity among autonomous worlds.

3.4 How to Make a Federation

Every organization is an island of semantic homogeneity, and in general we can expect that similar concepts in different organizations are more or less subtly different in meaning, so that different organizations are in general semantically heterogeneous. This means that the bottom-up method of making a federated database system will in general fail.

But organizations are not generally completely isolated. They interoperate with other on a business level. Organizations that interoperate will cooperate in making speech acts, so their institutional facts will integrate with other, at least to the extent they are cooperatively made. For example, if one organization buys something from another, the purchase is a cooperative speech act, recorded by one organization (the seller) as an instance of a *sale* institutional fact type and by the other (the buyer) as a *purchase*.

In order to cooperate in making a speech act, the two organizations must expose at least some of their data models as export schema. In the purchase case at least their product catalog (the seller) and requirements (the buyer) must be exposed. As we have seen in Chapter 1, the speech acts can be more or less complicated, possibly starting with a request for quotation and ending with a payment.

If they interoperate electronically, they need to make use of a number of standards. At least they need the communication protocols enabling the messages to be sent back and forth. Since their interoperation is a sort of game whose rules are at least enforced if not created by the commercial legal systems of the relevant jurisdictions, the individual speech acts are often standardized in one of the EDI standards. Each system needs to be able to create views on its local database to construct or interpret the relevant EDI message types.

So the global schema governing the interoperation comes partly from export schemas of the participants and partly from the EDI standards. The export schemas may be seen as bottom-up integration, but the EDI standards are pre-existing. Chances are each system will need to modify its practices and consequently its export schemas to make the exchange work. The global schema forms an ontology for the interoperation. We see that to at least some extent the global schema or ontology comes first and the participating organizations must adapt their systems to it.

Our dream in Chapter 1 was much more than the ability of two organizations to interoperate electronically. We wanted a many-to-many interoperation within a sort of exchange which provides a controlled but partly open environment. Suppose we have a network of pairs of organizations which interoperate via EDI. Does this lead to our goal?

The answer is not really. A single pairwise interoperation creates an ontology which establishes semantic homogeneity between the two organizations to the limits of the interoperation, but these ontologies are heterogeneous among themselves. They form a network, but not a single wider scale ontology.

For example, there are many variants of purchasing.

- Final sale: the purchaser pays the price and takes ownership of the goods.
- Sale or return: the purchaser pays the price, but retains the right to return unsold goods to the supplier for credit. Common in the publishing industry.
- Retail price maintenance: the sale could be either final or sale-or-return, but the supplier sets the retail price and the purchaser agrees to sell it at the nominated retail price. Common in industries with interesting products, including publishing and costume jewelry.
- Prices quoted at retail: even though the goods are paid for at wholesale prices, the prices are quoted at the retail price. Common in the department store business, where the buyers in individual stores get their authorization to buy for a period from retail sales totals of earlier periods, and the ultimate wholesale cost is handled by a separate central accounting department.

- Aggregate purchase rebates: sale is final, but if the purchaser over a period of time buys an aggregate total exceeding a supplier-set threshold, the supplier gives a rebate. Common in the grocery business.

These variants are generally not reflected in the EDI standards, since the messages all look the same. They are usefully seen as aspects of the background in the contexts of the various speech acts.

A single organization may use different variants with different customers. So the meaning of the EDI messages will be different depending on the partner. In fact, one of the limitations of EDI as traditionally practiced is that each pair of organizations must negotiate an agreement governing the exchange. The individual pairwise ontologies can't be integrated bottom-up into a single global ontology any more than the individual organizational data models can be integrated into a single global schema, and for the same reason: the prevalence of semantic heterogeneity.

To build an environment where agents can interoperate is a software development project in its own right. An aspect of it is a global schema or ontology, but this ontology is generally developed as part of the project. If an individual organization wishes to participate in the environment, it is their responsibility to make their internal systems conform to the global schema (called committing to the ontology).

A participating organization may need to change its business practices or its data model to commit to the ontology. Even though they will create an export schema to represent this commitment, the need for complex view definitions is lessened. If a particular aspect of the ontology is difficult to represent in their local data model, it will often be easier to change that aspect of their local data model along with the other changes they need to make. The responsibility for making the local system commit to the ontology is taken by the local system, not by the exchange as in the federated database model. The local systems must generally give up some of their autonomy in order to participate in the exchange.

The federated database approach has the global schema being constructed as a union of export schemas created from each of the autonomous local systems, with the responsibility of the construction taken by the exchange and the functionality of the global schema emerging from the integration. This approach is not generally feasible in practice. An approach which is feasible is for the exchange to develop a global schema as an ontology, with the local systems making the necessary changes to commit to it on their responsibility.

In the federated database approach the global schema is tailored to specific applications by the exchange creating a number of external schemas. Even this aspect of the federated database model does not suit an exchange. Consider that an exchange may have thousands of participants. Many of the participants will be interested only in particular aspects of the ontology. For example, a provider of cleaning services will be interested in only some aspects of the service portion of an e-commerce exchange, while a stationery supplier will be interested only in some aspects of the product portion.

Rather than have the exchange create an external schema for each participant, it is much simpler for the exchange provide publish-and-subscribe facilities for its ontology, and thereby allow each participant to subscribe to whichever aspects of the ontology are relevant to their participation.

3.5 Key Concepts

The main point of this Chapter is that the attempt to construct complexes of interoperating **autonomous** information systems from the **bottom-up** generally fails due to **semantic heterogeneity**. Semantic heterogeneity is explained as differences among the context and background in the systems of **institutional facts** implemented by the individual systems. It is generally necessary for the organizations involved to alter their systems of institutional facts to **commit** to a common **ontology**. This is generally easiest to do when the organizations do business with each other.

3.6 Further Reading

The classic reference to the bottom-up approach to federated databases is [5]. Semantic heterogeneity has been studied by [6] and by [7].

4 Some Examples

Now that we have a better idea of what an ontology is and how it originates, we can gain a better insight by looking at a few examples in some detail.

4.1 Need an Ontology Representation Language

If we are going to talk about the details of ontologies, we need a vocabulary to do this with. Such a vocabulary is called an ontology representation language. An ontology is a sort of data model, so we would like to adopt an existing representation language for the purpose. The problem is that there are a number of competing representation languages derived from different modeling traditions.

Some systems come from information systems and software engineering

- The *Entity-Relationship* (ER) model uses the terminology entity, instance, relationship, attribute and subtype
- The *Unified Modeling Language UML* uses the terminology class, instance, association, attribute and subclass
- *Object-Role* modeling uses the terminology object type, object, fact type, role and subtype.

These three systems are sketched in Appendix D.

Besides these modeling languages, there is a tradition of knowledge representation systems which has contributed to the concept of ontology.

- *Description logics* use the terminology concept, individual, role and inclusion
- The W3C *Web Ontology Language* (OWL) uses the terminology class, individual, property and subclass. (OWL is discussed in Chapter 11.)
- *Common Logic* (CL) is a syntax for the first order predicate calculus, and uses the terminology term, atom, predicate and function. (Common Logic is discussed in Chapter 13.)

And there are other sources

- *Topic Maps* is a system designed to represent the "aboutness" of topics, a sort of very extended book index, which uses the terminology topic, subject, role and association. (Topic Maps is discussed in Chapter 14.)

In the present work, we assume that the reader is familiar with one of the software engineering modeling languages. However, for reasons we will explain later, we will use a terminology derived from OWL.

The underlying semantics of all these languages is derived from set theory and the predicate calculus. The main concepts we will use are

- *Class*: a set of individuals. UML Class, ER Entity type.
- *Individual*: a single, but possibly complex thing. An individual belonging to a class is called an *instance* of that class. UML, ER instance.
- *Property*: a binary relation among classes. UML association or attribute, ER relationship type or attribute. The two ends of the property are called *domain* and *range*, with the property being read from the domain to the range.

- *Subclass*: a class whose individual members are also members of another class (the superclass). UML subclass, ER subtype.
- *Participation*: a class participates in a property if the class is an end of the property.
- A *derived property* can be defined by *composing* two properties. The range of the first is the domain of the second.

Unfortunately, OWL does not have a well-established graphical representation. So we will use UML notation when we need to visualize an ontology fragment. We illustrate the notation in Figure 3 with a simple familiar example, that of students enrolling in courses.

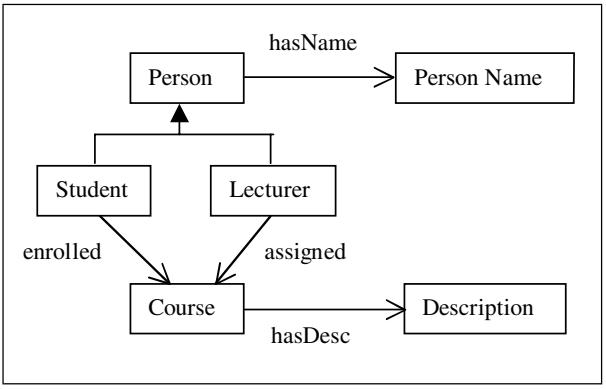


Figure 3. A simple data model

The classes are *Person*, *Student*, *Lecturer*, *Course*, *Person Name* and *Description*. The properties are *enrolled*, *assigned*, *hasName*, *hasDesc*. The classes *Student* and *Lecturer* are subclasses of *Person*. Individuals are instances of any of the classes. The properties are shown as having direction. The class away from the arrow is called the *domain*; the class at the arrow is called the *range*. (UML has directional or navigable binary associations where the corresponding ends are called *source* and *target* respectively.)

4.2 Z39.50 Information Retrieval Ontology

The reader will have used Z39.50, probably without knowing it, if they have ever accessed a major library catalog. Z39.50 is an ontology designed to enable access to library catalogs and other related kinds of resources. It is used by museums, statistical information resources, and by governments for access to information.

Players are people or organizations which either store or wish to access information. The key roles are information provider (called *service* in Z39.50) and information seeker (called *user*). A typical interaction between a user and a service would be:

- User wants to find books on a particular subject, so sends a query to the service
- Server generates a result set, which is the collection of catalog entries corresponding to books on the subject in the query. It responds with the number of books in the result set, together with a page of book catalog

summaries, at least title, author and date. Remember there could be dozens of books on that subject, so the page returned is only the first ten or so results.

- User ticks a tick box associated with each of the books on the page they think might be relevant, sends a message to the service asking for those ticks to be remembered, and then asks for the server to send the next page. There might be no interesting books on a particular page. When the last page has been viewed, there might be 7 of 85 of the books ticked.
- User then asks for full details of the first of the ticked books. The system responds with the details requested. This exchange is repeated until either the user stops making requests or the details of the last ticked book are viewed.

The service often will have a facility by which the user can ask that the result set be refined by date, or by language, or by type of material. Some services have a facility by which result sets can be named and boolean operations performed on them. Systems whose database is a collection of journal article abstracts often have this facility. Queries in these systems are like search engine queries, and will often generate a result set containing thousands of documents.

These interchanges require that the service be aware of state. The effect of a request by a user will depend on the history of previous requests. For example, the result of a *fetch the next page* command will depend on which page was last fetched. A command to refine a search within a result set will have a different result for different original queries, each of which will generate a different result set.

Speech acts supported by the Z39.50 ontology are mostly informative. The information providers update their information repositories outside of the interoperation supported by Z39.50. Principal speech acts supported are:

- Initialization Facility: establish a connection between user and service, initiate a Z39.50 session (called a Z-association), and negotiate expectations and limitations on the activities that will occur (e.g., maximum size of the records that will be transferred from the service to the user, the version of the protocol supported, etc.). This is a performative speech act.
- Search Facility: enables the user to submit a query using one of a number of standard query languages, including in particular the boolean query language typical of library catalogues and CD-ROM bibliographic databases. The server executes the search against a database(s), and a result set is created.
- Retrieval Facility: enables the client to ask for records from the result set or request from the server additional processing of the result set.

Each of these informative speech acts (facilities) is composed of two parts, a request message and a response message. Both the request and response are performative speech acts, which together constitute the informative (both players have to *do* something for the informative to take place). This type of information retrieval is often called *pull*, because the user must ask for the information, as distinguished from *push*, where the source sends information unsolicited. An advertising message is generally a push informative.

Besides these, there are facilities enabling an alerter to be established on the service as a sort of persistent query against the stream of new information being added to the database.

The world of Z39.50 includes

- Database: Contains the objects to be queried
- Query: A description of a subset of the database

- Result set: The subset of a database corresponding to a query
- Response Record: an object from the result set

The search facility allows a user to present a query on some databases to some servers, who will compute the corresponding result sets, held on the servers.

The retrieval facility allows the user to request subsets of records from result sets, which will be sent to the user.

Note that a result set is the same type of object as a database, and can be used as a database.

The objects in the Z39.50 world have properties.

- Database has a name
- Query has a type
- Result set has a name, size, and is ordered.
- Response Record has a format and a sequence number in its result set (result-set position), as well as a local identifier which points to the object represented by the response record.

We can think of the queries as having subclasses depending on their type, and response records as having subclasses depending on their format.

A visualization of the Z39.50 world is shown in Figure 4.

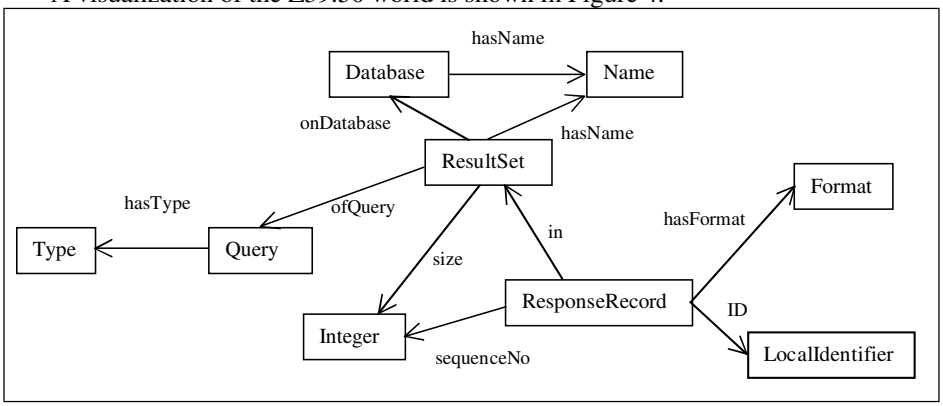


Figure 4. The Z39.50 world

The Z39.50 world has databases, queries and response records, but it doesn't include what these databases might be about, so doesn't include what might go into a query. If the database is a library catalog then it is about authors, titles, publishers, book identifiers and so on. If the database is a collection of museum object descriptions, then it is about collections, events, material, context, nationality and so on.

Various groups of people responsible for certain kinds of databases maintain what are called *attribute sets* which specify the kinds of things that are in their databases. The first and most important attribute set was developed by the library community. It is called *bib-1*. Some of bib-1 is shown in Figure 5. So the contents of library catalog-type databases has various kinds of names, some of which are authors of library items, titles, publishers, various kinds of publication identifiers, various kinds of library call numbers, various kinds of subject, and other kinds of things that could be found in a library catalog.

Personal name	Dewey classification	MESH subject
Corporate name	UDC classification	PA subject
Conference name	Bliss classification	LC subject heading
Title	LC call number	RVM subject heading
Title series	NLM call number	Local subject index
Title uniform	NAL call number	Date
ISBN	MOS call number	Date of publication
ISSN	Local classification	Date of acquisition
LC card number	Subject heading	Title-key
BNB card number	Subject Rameau	Author
BGF(sic) number	BDI index subject	Author-name personal
Local number	INSPEC subject	Author-name corporate

Figure 5. Bib-1 use attributes[1]

The names in Figure 5 are class names (called Use Attributes by Z39.50). Each of these classes contains a population (has an extent) which is mostly standardized by the library or publishing communities.

Several information repository communities have defined extensions to Bib-1 for their particular purposes. These include

- GILS: Government Information Locator Service
- CIMI: *Computer Interchange of Museum Information*. Supports searching museum collections

The Z39.50 world includes databases, result records and pointers to objects. The worlds specified by the various collections of use attributes include classes which can be the ranges of properties of the objects in the Z39.50 world. An object in the Z39.50 world can be associated with an instance of say *AuthorName*, *Title* and *LCCallNumber* in the bib-1 world. The properties connecting the two worlds are typically implicit, so not formally named.

A query will typically consist of a number of values of use attributes combined in some way, often in a Boolean syntax structure, for example "AuthorName = 'Colomb' or Title = 'Information Spaces'". Note, though, that the speech acts do not depend essentially on the use attributes. The use attributes are only employed to construct the query and interpret the response.

It is common for groups of libraries or other institutions to permit simultaneous search of all of their catalogs through Z39.50. This is possible because each member of the group has committed to the Z39.50 ontology and also to a particular set of use attributes.

4.3 Tic-Tac-Toe

We saw in Chapter 2 that a game is a good example of an institutional world. A game therefore requires an ontology, a set of terms which everyone agrees upon which describe its institutional world. A simple example is the children's game tic-tac-toe or noughts and crosses. We will first describe the game, placing in *italics* the terms in the ontology, then we will look at the ontology.

[1] ftp://ftp.loc.gov/pub/z3950/defs/bib1.txt

Tic-tac-toe involves two *players*, *X* and *O,* and a *game space*, an example of which is shown in Figure 6.

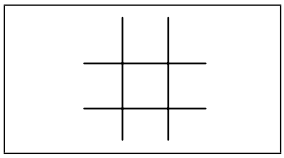


Figure 6. Game space for tic-tac-toe

Each *player* in *turn places a mark* in one of the *vacant cells*. *Player X* gets the first *turn*. If after a *turn* there is a *row* of *cells* with the *mark* of the *player* whose *turn* it is, then that *player wins*. If before a *turn* there is no possible *row* of *cells* with the *mark* of one of the *players*, then the *game* is *blocked*. Figure 7 shows examples of a *blocked game* and a *won game* for *player X*.

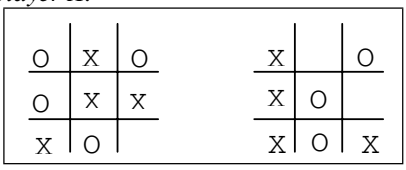


Figure 7. Blocked game (left) and winning game for X (right)

The game has two distinguished *strategies*. *Player X* has a *strategy* called "*not lose*" which guarantees that *player O* cannot *win*, which begins by *placing a mark* in either the *center cell* or one of the *corner cells*. *Player O* has a counter *strategy* called "*block*" which guarantees that the *game* will be *blocked*. If *player X* has *placed a mark* in the *center*, *player O places a mark* in one of the *corners*. If *player X* has *placed a mark* in one *corner*, *player O places a mark* in the *center*.

The two *strategies* can be illustrated by *game records* ending in each of the two *games* of Figure 7, as shown in Figure 8.

Player X	Player O
center	top left corner
bottom left corner	top right corner
top side	bottom side
right side	left side

Player X	Player O
top left corner	top right corner
bottom right corner	center
bottom left corner	bottom side
left side	

Figure 8. Game record of blocked game (left) and winning game for player X (right)

The *game records* have required some *place terms*: *centre*, *corner*, *side* and *spatial terms* to describe the individual *cells*: *top*, *bottom*, *left* and *right*.

All of these terms have been collected into an ontology in Figure 9, using the concepts class, instance and property from our representation language. Note that the visualization has been augmented to show instances on the diagram, where the instances are part of the ontology as in the classes *Strategy*, *PlayerRole*, *Places*, *Horizontal* and *Vertical*.

The class *Game* at the center of the top represents the game space of Figure 6. The *Game* consists of (property *has*) a number of *Cells*. The *Cells* have properties *atPl* with *Places*, and *atH* and *atV* with the two subclasses *Horizontal* and *Vertical* of *Directions*, respectively. *Cell* has another property *value* whose range is the class *Token* which contains exactly the tokens "X" and "O". *Cell* has two subclasses, *Vacant* and *Filled*, and *Filled* has two further subclasses, *X* and *O*, respectively those containing the token

"X" and "O". Figure 7 contains two instances of *Game*, while Figure 8 illustrates the use of the properties *atPl*, *atH* and *atV* of *Cell*.

Games have a number of distinguished states, modeled by the property *hasState* whose range is the class *State*. A *State* is either *Blocked* or *Winnable* (the left game in Figure 7 is blocked, the right winnable). A *Winnable State* has two distinguished subclasses, *Initial* and *Won*, with *Won* having two further subclasses *WonX* and *WonO*. The right game in Figure 7 is of state won *wonX*. Notice that the subclasses *Blocked* and *Winnable* exhaust the class *State* (every instance of *State* is an instance of one or the other), as do the subclasses *WonX* and *WonO* of *Won*. But there are many instances of *Winnable* which are neither *Initial* nor *Won*, for example the game with all cells vacant except for *X* in the center, so these subclasses do not exhaust their superclass. It is very common in an ontology to have a few distinguished subclasses which do not exhaust their superclass. For economy, we generally only name the subclasses we want to talk about, leaving the rest in the superclass.

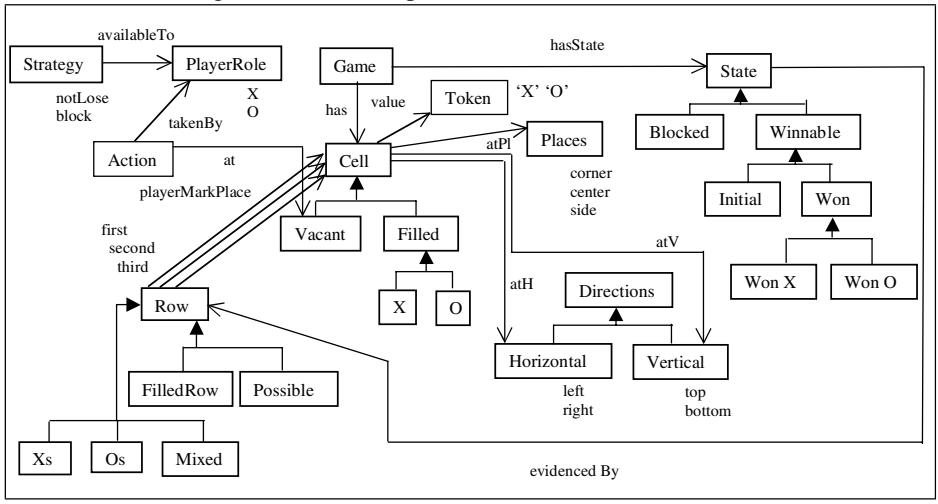


Figure 9. Tic-Tac-Toe ontology

A game is won if there are three cells in a row with the same mark. A game is blocked if there is no row which could possibly have three cells with the same mark. These situations are modeled by the class *Row* in the bottom left of the Figure. A *Row* has three properties, *first*, *second* and *third*, the range of all of which is *Cell*. The *Row* which tells us that player X has won the right game in Figure 7 has the property values "top left corner" for *first*, "left side" for *second* and "bottom left corner" for *third*.

Row has five subclasses, in two groups, *FilledRow* versus *Possible* and *Xs*, *Os* versus *Mixed*. A *FilledRow* has no cells vacant, while a *Possible Row* may have vacant cells but all the filled cells have the same mark. In the right game in Figure 7, the row "top side", "center", "bottom side" is *Possible*; the row on the bottom is a *FilledRow*, but the row on the right is neither. So *FilledRow* and *Possible* do not exhaust *Row*. All the rows in the left game of Figure 7 including the cell "bottom right corner" are in neither subclass, for example. The other group consists respectively of rows in which no *Filled Cells* are *O*, no *Filled Cells* are *X*, and some *Filled Cells* one and some the other. The subclass *Xs* includes all rows which could possibly be filled with *Xs*. So a row all of whose cells are *Vacant* is an instance of *Xs*, but a row with any cell *O* is not.

Similarly for *Os*. So unlike all the other sets of subclasses we have seen in this ontology, these subclasses are not disjoint. A row all of whose cells are vacant is an instance of both *Xs* and *Os*. But these three subclasses do *exhaust* their superclass (every instance of the superclass is also an instance of one of the subclasses).

Having *orthogonal* (independent) groups of subclasses is fairly common. This means that an instance of the superclass can be an instance of at least one subclass of each group. For example, the row "top left corner", "left side", "bottom left corner" of the right game in Figure 7 is an instance both of the subclass *FilledRow* in the first group and *Xs* in the second.

The taxonomy of rows is used to provide a vocabulary to specify the subclasses of *State* via the property *evidencedBy*. An instance of *State* is associated with possibly several instances of *Row* because an instance of *State* is associated with instances of *Game* via the property *hasState*. For example, an instance of *State* is an instance of the subclass *Blocked* if all of the rows in the instance of *Game* are instances of *Mixed*, of the subclass *Initial* if all of the rows in the instance of *Game* are instances both of *Xs* and *Os*, and of *WonX* if there is an instance of *Row* which is both an instance of *Xs* and of *Filled*. Note that the ontology as specified in Figure 9 does not contain these rules, but only provides the terminology in which the rules can be stated.

In the upper right corner of Figure 9 are the classes through which the instances of *Game* are created by changing the value of instances of *Vacant Cell*. A value is changed by an implementation of an instance of the class *Action*. The action is *at* an instance of *Cell* and *takenBy* one of the instances of *PlayerRole*. Finally, the ontology includes the two distinguished instances of *Strategy* "notLose" and "block", each of which is *availableTo* one of the instances of *PlayerRole* ("notLose" by "X" and "block" by "O"). Note that the ontology does not include a definition of the strategies.

The Tic-Tac-Toe ontology includes some instances (of classes *Strategy*, *PlayerRole*, *Action*, *Places*, *Horizontal* and *Vertical*). In this way it differs from the Z39.50 ontology of Figure 4, which consists entirely of classes and properties. Even the add-ins of the particular sets of use attributes are also entirely classes. Instances of many of the classes in the Z39.50 world are standardized (e.g. ISBN), but the instances are not considered part of the ontology.

We now look at what this ontology lets us say. The central class is *Cell*. Instances of *Cell* can be differentiated by values of the properties *atPl*, *atH* and *atV*. There are nine cells. Not all cells have values for all properties. In particular, if a cell has the value "center" for *atPl*, it has no value for either *atH* or *atV*. Allowing for the possibility of a cell not having a value for a property, there are 36 possible combinations of property value. In the ontology as shown there is no constraint to disallow combinations outside the nine valid cells, nor is there a constraint that shows that the properties *atPl*, *atH* and *atV* identify instances of *Cell*. A cell is either vacant or contains a token, as shown by the subclass structure.

The property *has* associates an instance of *Game* with a number of instances of *Cell*. In practice each instance of *Game* is associated with exactly nine instances of *Cell*, but again the ontology does not enforce that. Each cell has three flavours (*Vacant*, *X* or *O*), so there are $3^9 = 21{,}183$ possible instances of *Game*. Some of these instances are of course redundant because the game is either won or blocked while there are vacant cells. In Figure 7, for example, there are three possible instances of *Game* equivalent to the left instance and nine equivalent to the right. The ontology does not contain any mechanism to eliminate the redundant instances. More generally, the ontology does not enforce a constraint that an instance of *Game* be attainable in play.

For example, the situation where all nine cells contain "X" is one of the 21,183 possible instances.

So we see that the ontology allows us to represent games and states of games, and to describe them. It allows us to talk sense but doesn't prevent us from talking nonsense. It would be possible to enhance the ontology with constraints disallowing non-sense, for example by requiring that a *Game* instance participate in the *has* property with exactly nine cells or by requiring that a *Game* instance contain either equal numbers of cells of flavour *X* and *O* or at most one more *X* than *O*. The ontology provides the terminology with which to describe the constraints in some constraint language.

Further, the ontology includes the messages by which the state of an instance of *Cell* is changed (instances of *Action*) but has no mechanism for recording an actual game. To do this would require concepts of a sequence of action alternating between *PlayerRoles*, terminating at either a *Blocked* or *Won* state of *Game*. It would also need a concept of a player who can take a *PlayerRole* in a given actual game, and a way of representing the outcome of an actual game. It would need a mechanism for associating the sequence and outcome of an actual game with the players and the roles they have taken. The ontology also allows us to name instances of *Strategy* but would need some sort of programming language to permit us to completely describe one.

But this ontology could be quite useful. Consider a network of agents which interact by playing games of Tic-Tac-Toe. One agent could be designated as keeper of games. This keeper of games agent would accept messages from the player agents establishing contests and assigning *PlayerRoles*. The data structures to do so would be outside the ontology. This agent would implement constraints for valid game states and update the game states according to the *Action* messages sent by the players. So long as the keeper of games agent were trusted, it would not be necessary for the other agents to maintain the constraints. In fact, since the game space is very small, the individual player agents could allow the keeper of games agent to send the current game space and state properties as an acknowledgement of a move message being accepted as valid.

4.4 Standard Industrial Classification

Many ontologies are published as taxonomies, or hierarchical collections of broader and narrower terms. A good example is the Standard Industrial Classification (SIC) maintained by the United States Department of Labor, some of which is shown in Appendix A. This is a classification of industries at four levels of specificity. At the top are ten very general divisions (called Divisions), one of which is *Division I: Services*. Each Division is subdivided into at most ten Major Groups, for example Division I contains *Major Group 73: Business Services*. Major Groups are further divided into at most ten Industry Groups. Major Group 73 contains *Industry Group 736: Personnel Supply Services*. Finally, Industry Groups are divided into at most ten Industries. Industry Group 736 contains *Industry 7361 Employment Agencies*. In total, there are something less than ten thousand Industries in the system.

First, the SIC is a good example of a system of institutional facts. The brute facts are a myriad of individual enterprises ranging in scale from sole traders to giant multinational corporations like General Electric or large governments like the Government of the United States (these are of course in their own right institutional

facts, and strictly their contexts are included in the context of the SIC). Large organizations are generally divided into smaller administrative units. One of the subunits of General Electric is the e-commerce service provider General Electric Global Exchange Services. One of the subunits of the Government of the Australian state of Queensland is the Department of Education, which runs primary, secondary and technical schools. These subdivisions are in their own right also institutional facts. But for the purposes of the SIC, the organizational subunits are brute facts.

There is a group within the US Department of Labor which decides what general kinds of economic activity take place, and which more specific kinds are more similar to each other than to others so should be grouped together into more general categories. So they decide that the Queensland Department of Education is appropriately described as a provider of educational services along with (the private) Stanford University, even though it is a Government Department and there is a Division *Government Services* which has a Major Group *Administration of Human Resource Programs*. These decisions are all made in the context of a system of rules for deciding the appropriate scale of organizational unit to consider and what to do about individual functions within the smallest units. For example, even if an automobile manufacturing plant has a group which trains new staff, the plant will be considered to be in Major Group 37 *Transportation Equipment* of the *Manufacturing* Division in the context of the rules.

There are a number of ways of representing the SIC in our ontology representation language depending on what the SIC is to be used for.

The original purpose of the SIC was to provide a way of organizing and aggregating data obtained on individual workplaces or organizational units, data such as employment, gross output and job vacancies. A simple data model for this purpose is shown in Figure 10. It shows *organizational subunits* with two properties, *workforce* in *full-time employment units* and the *SIC classification*.

This formulation treats the SIC classifications simply as a collection of objects with no visible internal structure. It has the advantage that it allows a given organizational subunit to be associated with SIC at any level of specificity, and also with multiple instances of SIC. In this formulation, the SIC could be employed as a use attribute in a Z39.50 use attribute set supporting economic development, statistics, or the like. The disadvantage is that it doesn't make it obvious that the workforce can be aggregated at different levels of specificity, and that the hierarchical structure of the SIC can be navigated using on-line application processing functions like drill down and roll up. (*Drill down* takes a total at a more general level and breaks it down into more specific subtotals. If we had employment at the division level for *Services* for example, we could drill down to the major groups making up that division so we could see the relative importance of *Business Services* versus *Motion Pictures* for that particular economy. *Roll up* is the reverse. We start with a more specific subtotal and pass to a more general.)

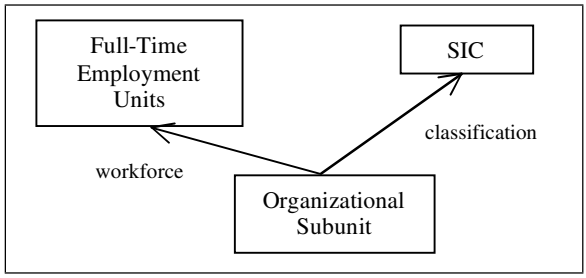


Figure 10. Using SIC as a classifier

Figure 11 shows one way to make the hierarchical structure visible. Each level of specificity is connected to the next more general level by a property. The cost is that each organizational subunit must be directly associated with the most specific classifiers, the Industry level, so that it is no longer possible to have some units directly classified at more general levels and some at more specific. Data warehouses typically have schemas that look like this.

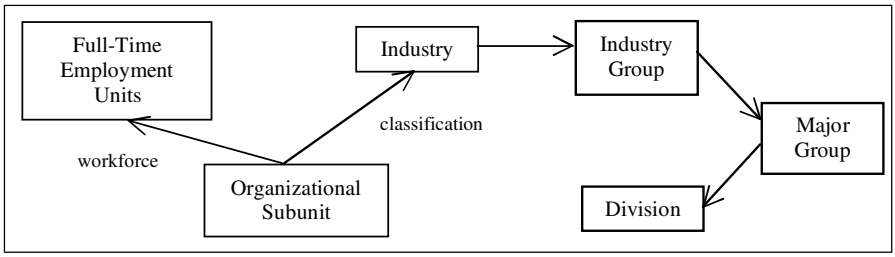


Figure 11. SIC ontology showing hierarchical structure

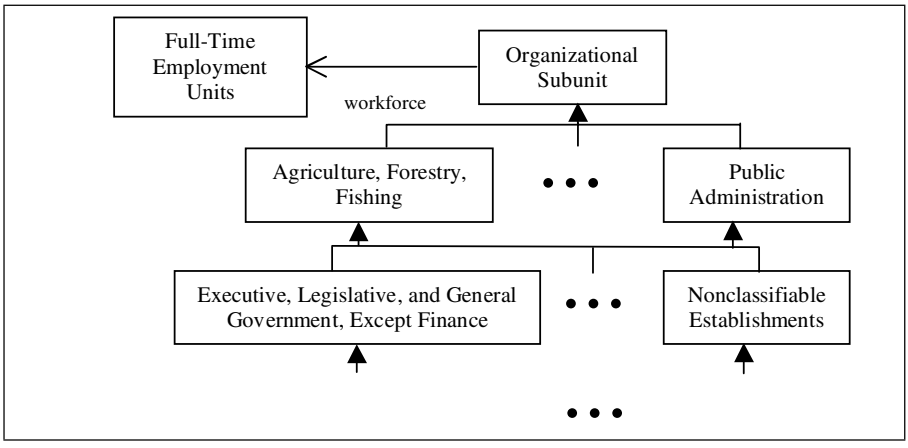


Figure 12. SIC as subclass structure for Organizational Subunit

The disadvantage of this model as an ontology is that the four properties implementing the structure are not the same. A way to overcome this disadvantage is to make the original purpose of the SIC visible by representing it as a subclass structure for *Organizational Subunit*, as in Figure 12.

A problem with this approach is that conventional visualization techniques are not capable of showing systems with thousands of subclasses. Of course, this is a problem only if the user of the ontology is a human. A computer program would be capable of

managing a structure of this scale. A second problem is that the layered structure of the hierarchy (*Division*, *Major Group*, *Industry Group*, *Industry*) is not apparent.

One way to overcome these two problems is to re-interpret the classes *Industry*, *IndustryGroup*, *MajorGroup* and *Division* of Figure 11 as having instances the subclasses of Figure 12 rather than simply names. We could then invent some new notation, and represent the system of Figure 12 as in Figure 13.

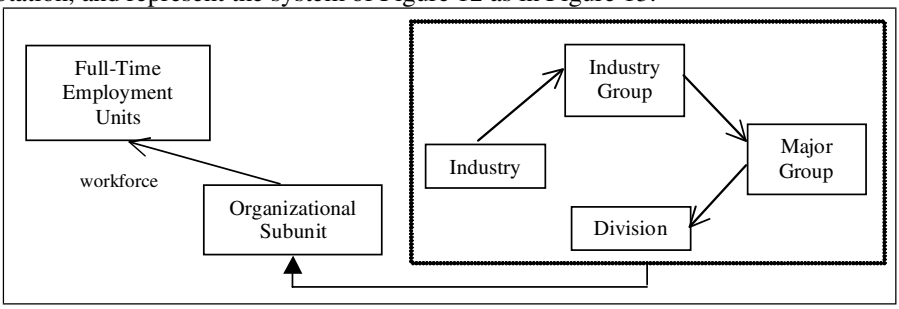


Figure 13. SIC shown as subclasses grouped as instances of a system of classes

This discussion of the Standard Industrial Classification System has shown a number of ways of representing a very common type of structure as an ontology. It has also shown that there is more than one way of representing an ontology, as with any other data model. Representing an ontology is a design task, like any other software engineering task.

4.5 SNOMED

SNOMED is an ontology used for classifying medical treatment incidents for purposes of medical records and for health insurance reimbursement. It is part of the infrastructure for integrated health care systems where information about patients can flow among hospitals, medical practitioners, and health funding agencies. It is similar in many ways to the SIC, but is very much larger.

As shown in Table 1, the SNOMED system has more than 150,000 class names organized into 11 groups called facets or axes. A particular incident, say a fever caused by an infection in the throat by a particular bacterium treated by a particular antibiotic, would be classified using names from *Anatomy*, *Living organisms*, *Symptoms*, and *Chemicals and drugs*, at least. There are therefore a vast number of possible classifications.

We can observe several things about the SNOMED system. First, all of the facets are large. The smallest, *Social context* has more than 1000 class names, while the largest, *Diagnoses*, has more than 40,000. These facets are therefore all hierarchically organized systems in themselves comparable in size to the SIC. A medical incident report is intended to be classified by generally several of the facets. This is like the two systems of subclasses for *Row* in the Tic-Tac-Toe ontology of Figure 8, except on a very much larger scale.

Table 1. The facets of the SNOMED system

Topography (Anatomy)	13,165
Morphology	5898
Diagnoses	41,494
Procedures	30,796

Functions, symptoms, etc.	19,355
Living organisms	24,821
Chemicals and drugs	14,859
Physical agents, forces	1,601
Social context	1,070
Occupations	1,949
General modifiers	1,594
Total	156,602

Secondly, the facets are generally not exhaustive. An *exhaustive* system of subclasses is one for which each object to be classified must fall into at least one of the subclasses. The SIC is exhaustive. With SNOMED, most medical incident reports would be described by only some of the facets. If no drug is prescribed, then no classification from *Chemicals and drugs* would be used. If a person has a systemic disease like chicken pox then no classification from *Topography (Anatomy)* would be used. Of course it is possible to expand each of the facets with a class name *Not applicable*, which would then make each of them exhaustive. But the existence of non-exhaustive systems of subclasses means that the instances are no longer subordinate to the class system, as in our metamodel. An *individual* object may or may not belong to a class in a particular system.

Thirdly, the entire classification system is never used in practical analysis of the data collected. The total number of classifications possible, if all eleven of the facets are used in every incident report, is more than 10^{40}. If every person on earth had 10 incident reports per year for their entire lives, there would be a total of fewer than 10^{13} incident reports after 100 years. This is to say that even if each incident report had a different classification, only 1 in 10^{21} classifications would have an entry. This is equivalent to 1/10 of a microsecond during the entire history of the earth. (The point here is partly that due to combinatorial explosion, faceted systems of subclasses can generate enormous numbers of combinations of class membership.) In this example, the class systems are subordinate to the individual. An individual is classified in particular ways depending on the purposes of the user.

The SNOMED system is used for transaction purposes - insurance reimbursement funding categories are often based on high levels of the code system, and the system provides a controlled vocabulary for keeping medical records. The SNOMED system also supports data warehousing type queries (such as described in the section on the SIC above) on large populations of incident reports. Each query involves only a few of the facets.

Finally, we observe that the degree of specificity of classification differs greatly between incident reports. For example, a routine antibiotic treatment for a sore throat would be coded in the *Topography/Anatomy* facet with a fairly general category indicating the throat, while a treatment for a small tumor would be coded with a much more specific class denoting the specific structure in the throat on which the tumor was found. In the routine sore throat incident, the report would use a general class from the *Living organism* facet indicating unspecified bacterium, while an unusual case would require a laboratory analysis of the organism and a much more specific class from that facet. In contrast, in the SIC system of Figure 13 an organizational subunit is always an instance of a most specific subclass, one of the *Industries*.

An adequate treatment of systems as large and complex as SNOMED is a major challenge for a general ontology system. One issue is how to describe sensible or required combinations of facets.

4.6 Periodic Table

The Periodic Table of the Elements is a key tool in chemistry and physics, and would be a key component of applications in these fields. It is a system which organizes the properties of the 109 known chemical elements. Its standard representation is shown in Figure 14.

Period	Group 1 IA 1A	2 IIA 2A	3 IIIB 3B	4 IVB 4B	5 VB 5B	6 VIB 6B	7 VIIB 7B	8	9 VIII 8	10	11 IB 1B	12 IIB 2B	13 IIIA 3A	14 IVA 4A	15 VA 5A	16 VIA 6A	17 VIIA 7A	18 VIIIA 8A
1	1 H 1.008																	2 He 4.003
2	3 Li 6.941	4 Be 9.012											5 B 10.81	6 C 12.01	7 N 14.01	8 O 16.00	9 F 19.00	10 Ne 20.18
3	11 Na 22.99	12 Mg 24.31											13 Al 26.98	14 Si 28.09	15 P 30.97	16 S 32.07	17 Cl 35.45	18 Ar 39.95
4	19 K 39.10	20 Ca 40.08	21 Sc 44.96	22 Ti 47.88	23 V 50.94	24 Cr 52.00	25 Mn 54.94	26 Fe 55.85	27 Co 58.47	28 Ni 58.69	29 Cu 63.55	30 Zn 65.39	31 Ga 69.72	32 Ge 72.59	33 As 74.92	34 Se 78.96	35 Br 79.90	36 Kr 83.80
5	37 Rb 85.47	38 Sr 87.62	39 Y 88.91	40 Zr 91.22	41 Nb 92.91	42 Mo 95.94	43 Tc (98)	44 Ru 101.1	45 Rh 102.9	46 Pd 106.4	47 Ag 107.9	48 Cd 112.4	49 In 114.8	50 Sn 118.7	51 Sb 121.8	52 Te 127.6	53 I 126.9	54 Xe 131.3
6	55 Cs 132.9	56 Ba 137.3	57 La* 138.9	72 Hf 178.5	73 Ta 180.9	74 W 183.9	75 Re 186.2	76 Os 190.2	77 Ir 190.2	78 Pt 195.1	79 Au 197.0	80 Hg 200.5	81 Tl 204.4	82 Pb 207.2	83 Bi 209.0	84 Po (210)	85 At (210)	86 Rn (222)
7	87 Fr (223)	88 Ra (226)	89 Ac~ (227)	104 Rf (257)	105 Db (260)	106 Sg (263)	107 Bh (262)	108 Hs (265)	109 Mt (266)									
Lanthanide Series			58 Ce 140.1	59 Pr 140.9	60 Nd 144.2	61 Pm (147)	62 Sm 150.4	63 Eu 152.0	64 Gd 157.3	65 Tb 158.9	66 Dy 162.5	67 Ho 164.9	68 Er 167.3	69 Tm 168.9	70 Yb 173.0	71 Lu 175.0		
Actinide Series~			90 Th 232.0	91 Pa (231)	92 U (238)	93 Np (237)	94 Pu (242)	95 Am (243)	96 Cm (247)	97 Bk (247)	98 Cf (249)	99 Es (254)	100 Fm (253)	101 Md (256)	102 No (254)	103 Lr (257)		

Figure 14. Periodic Table of the Elements[2]

Element is the class being subdivided. The table shows two systems of subclasses (facets), each of which is exhaustive, *Group* and *Period*. Properties shown on the diagram include *atomic number* and *average atomic weight*. There are two additional subclasses shown, the two series of rare earths, *Lanthanide Series* and *Actinide Series*. Each of these is a subclass both of particular instances of *Group* and *Period*.

Figure 13 is simplified. In the source there is an additional system of subclasses, the eight *Element Groups*, from *Alkali Metals* to *Noble Gasses*, which interact with the *Groups* and *Periods* in a complicated way. Further, the Table often includes many other properties of the elements, including *half-life*, *colour*, *melting point*, *combustion temperature* and so on.

This example is interesting because it is an ontology with several facets of subclasses, and because it includes a system of instances as well as classes.

[2] Source: Los Alamos National Laboratory's Chemistry Division
http://pearl1.lanl.gov/periodic/default.htm

4.7 Dimensions

Our final example is that of dimensions, which is complex in a different way from the Periodic Table. There are a large number of dimensions and units of measure. One reason one might want an explicit representation of the dimension of an attribute or class is that different dimensions permit or forbid certain arithmetical operations. For example, it is semantically valid to add two volumes of the same unit, or two distances of the same unit. But it is not semantically valid to multiply two volumes, while it is semantically valid to multiply two distances (giving a different dimension, 'area'). It is questionable to add two temperatures. One can compute the average of a set of temperatures, but it does not make sense to add the temperature of the cup to the temperature of the coffee poured into it.

Another reason is that a system of dimensions is an aspect of reality that can be factored out of specific applications. A system of dimensions includes

- the names of the dimensions: distance, area, volume, time, charge, voltage, acceleration, capacitance, etc
- rules for determining the dimension of the result of a calculation on dimensioned objects: area is length x length, acceleration is length / (time x time), etc.
- a system of interdefined units (metres, kilometres, millimetres; grams, kilograms, milligrams; square yard, perch, acre).
- The rules can incorporate some physics or other science. For example, the temperature of the cup can't be added to the temperature of the coffee, but the temperature of the filled cup can be calculated using a formula involving mass and specific heat of the constituents as well as their temperature.

Dimensions can be represented in units in generally more than one way. Dimensions and their units often come in systems (e.g. the American foot-pound-second, the SI metre, kilogram, second, the SI centimetre, gram, second). Money is a dimension with units the various currencies, which is a sort of system. But there are many dimensions that are idiosyncratic, such as "burn-rate" with units million US dollars per month used in the venture capital industry, widely reported during the dot-com boom of the late 1990s. The same unit name often occurs in more than one system. Sometimes the units are the same, as between the British and American 'mile', and sometimes different as between the British and American 'gallon' or the Troy and Avoirdupois 'ounce'. Sometimes the same unit name refers to two different dimensions, such as the (fluid) ounce versus the (unit of force) ounce.

The most general structure is the dimension system. Individual dimensions are not themselves dimension systems, so it is not appropriate to consider them subclasses. We rather treat the dimensions as parts of a dimension system.

A dimension exists independently of any units. It can be referred to ostensively "that distance" or concretely "the distance from Ghent to Aix". Any property relating to the same dimension is comparable. Further, each dimension in a system is represented by any of a collection of units, for example distance in the metric system is measured by metres, kilometres, centimetres, millimetres, and so on.

To adequately represent dimensions we need to have a large number of calculation rules and consistency constraints. Besides the relationships among the units of a single

dimension, familiar examples of cross-dimensional relationships are that distance = rate x time, or Ohm's law. The whole field is called Dimensional Analysis.

4.8 Discussion

In this Chapter we have described in some detail three example ontologies in terms of a metamodel based on OWL, and three other ontologies in less detail. Of the first three, Z39.50 is a widely used system supporting interoperability, Tic-tac-toe is a moderately complex example showing many features of the metamodel, while the SIC is a large taxonomy, similar in structure to a wide range of other ontologies, which shows some scale issues.

The second group includes SNOMED, which is a faceted system like many SICs, showing different issues of scale, and also the importance of the distinction between individual and instance of a class. The Periodic Table is a small complex system showing the importance of instances as well as classes in an ontology. Finally, dimension systems are an example showing the need for the ability to include complex consistency checking and calculation rules in an ontology.

We will return to these examples in subsequent Chapters.

4.9 Key Concepts

To represent an ontology we need an **ontology representation language**. Ontologies can be packaged in modules. A given ontology can be represented in many ways. Some ontologies are huge and complex. Some ontologies include individuals. Some ontologies need complex rules in their representation

4.10 Exercise

1. (relevant to point 3 of the major exercise) Consider the Olympic fragment from the Chapter 2 exercise.

Make a description of the ontology as a diagram in the representation language of Chapter 4. Include enough detail to understand the elements of the ontology involved in the interoperations. Include some instances of all classes, including all classes used in the actions of the exercise from Chapter 2.

5 Complex Objects

We have seen some examples of ontologies, expressed in a simple metamodel derived from the Web Ontology Language OWL. We now look at representing complex objects, objects which are wholes consisting of configurations of parts.

5.1 A World of Objects

Like humans, systems of interoperating agents on the Semantic Web live in a world of objects. One absolute requirement for successful living is to be able to identify objects. We need to know that the things we see when we wake up in the morning are the same things that we saw before we went to sleep the night before. Many objects are complexes of parts making up a whole. A second very strong requirement is to be able to tell which parts make up which wholes.

As we have seen in Chapter 2, in the human world there are physical objects and objects which are institutional facts. Institutional facts behave very differently from physical objects. Information systems deal almost exclusively with institutional facts. We want to use our intuitions about humans dealing with objects to help us understand how agent programs deal with objects. Our most accessible intuitions are of physical objects. But because physical objects are so different from institutional facts, our intuitions about physical objects are misleading. We need to think about how we as humans deal with institutional facts in order to transfer this experience to the design of information systems.

Imagine that I have lost a bag containing an Australian $1 coin and my driver's license. All three are physical objects, but the coin and driver's license are also institutional facts. Suppose I replace them. Clearly, the new bag is different from the old bag. The new coin has the same value as the old, but is a different coin. The new license is a different piece of paper, but is the institutional fact different? We need to look a little more closely.

An institutional fact is a record of a speech act. The relevant speech act is a declaration by an agent of the Queensland Department of Motor Transport that I am a competent driver and authorized to drive a motor vehicle on public roads in Queensland. The license document is a record of this act. But when I ask the Department for a new license document, they don't have to make another speech act. They don't require that I pass another series of driving tests. What they do is consult their records (these days in their information system), discover that they have a record of my being authorized to drive in Queensland, and issue another license document. The institutional fact was not lost, only a copy of the record of the speech act. So my new bag contains a new coin but the same license to drive.

Nearly all institutional facts are like my driver's license. When a speech act is made, there are generally multiple copies made, and if one copy is lost it is generally possible to consult other copies and make new ones. Think of citizenship, as recorded in a birth certificate or a naturalization certificate. Think of the institutional fact of being a student enrolled at a particular university, or of being a graduate with a

particular degree, or the record of having purchased something, or of having made a payment on your credit card account. The only way these institutional facts can be destroyed is by all copies of the record of the speech act being destroyed. This does indeed sometimes happen, but the practice of keeping multiple records insures against loss of institutional facts. Database management systems have elaborate procedures for backup and restore partly for this reason.

Notice that unlike the license to drive, the $1 coin, an institutional fact, was lost along with the bag. This tells us that there is more to institutional facts than that discussed in the previous paragraph. To tell a more complete story, however, would take us too far from the point. The objects in an ontology behave much like the license to drive, so we will develop the example further.

Our two key problems are how we can identify an object and how we can tell which parts belong to which whole. So how do we identify my license to drive?

The Queensland Department of Motor Transport, who made the speech act creating the institutional fact, identifies the license to drive by its license number. However, when I go to get a new copy of the license I may not remember the license number. The Department recognizes this possibility, so makes use of their practice of issuing a license to only one person and only one license to a person, by recording also my identity, using my name and date of birth. It gets these of course from documents I present at the time of my original license issue. The documents are further institutional facts, records of the speech act in which I was named. So if I can present documents verifying my identity, the Department can search its records and find its record of my license to drive and from this issue me a new copy. Their database system will represent the license number as a primary key and my name and date of birth as a secondary key.

A license to drive is not a simple object, but composed of several parts. To see this, consider the process in which the institutional fact was created. First, a license number is created by some process. Second, details of the type of license are entered. Third, my name and date of birth are extracted from my identity documents. Fourth, my address is extracted from documents verifying my address. Fifth, some identifying physical characteristics are recorded. Finally, because there are many circumstances in which officials must confirm that I am the person to whom the license was issued and no other identity documents may be available, the Department takes a photograph and a sample of my signature and adds these to the record. A driver's license is a complex object.

There are millions of driver's licences issued by the Queensland Department of Motor Transport. Each license has several parts, so there are tens of millions of parts. How do we know which parts make up which wholes? To see this, we need to look at the physical objects constituting records of the license. One such object is the driver's license document issued by the Department for me to carry. Here, all the parts are assembled into a single card sealed in plastic. So the parts making up a license are held together by all being in the same container.

But the Department also maintains the record, in a very different form. They keep the record in a database system. One record type in the system will contain all the textual information in separate fields. So these parts are held together in a similar way to the paper document, by all being fields in the same database record. But the photograph and signature are images. These may be stored in different databases, as records one of whose parts is a field containing the license number. These parts are connected to the textual record by all three records containing the same license number.

In database terminology the three records are held together by foreign key dependencies.

5.2 Ontologies Versus Models

Of course individual information systems as well as ontologies represent objects. The examples in Chapter 4 show that an ontology looks very much like a conceptual model as has been used for many years in designing information systems. In fact, the two are closely related, but not identical. A conceptual model relates to a particular information system, while as we have seen an ontology is a representation of a world external to any particular system, usually shared by a community of interoperating information systems.

Further, a conceptual model is a specification, with the detailed content and representation of individuals left as an implementation decision. Therefore two different implementations of the same conceptual model may be quite different. Think of a relational database implementation versus an object-oriented database implementation. In particular, think of difference in the way a driver's license is represented as a license document and in the Department of Motor Transport's database.

Since an ontology must facilitate interoperation, it must include specification of individuals sufficiently concrete that two different systems will be able to represent them in the same way, at least externally. In particular, this includes naming conventions and identification schemes. This is why a driver's license is identified by both license number and by the identity of the person to whom it is issued. It is further why the person to whom the license is issued is identified in three different ways. In our examples of Chapter 4, specification of individuals was central to the Z39.50 system, representations of individuals were central to the Periodic Table, and the SNOMED system with several non-exhaustive facets requires the representation of individuals independent of the classification systems.

Internal representation is of course still an internal design decision. Different systems will implement the identification in different ways, so long as externally they are compatible. So if the ontology uses a relational data model, an object-oriented implementation must provide views to represent the internal structure of objects as tuples in tables linked by foreign keys. If a player has its internal system structured differently from the ontology, it must store the tables giving the correspondences between internal and external identifiers and naming schemes. For example, in a bibliographic application, the internal representation of citations and bibliographic references may be in IEEE format, different from the ontology which might be in ACM format.

So an ontology is generally concerned only with the shared objects and not with the internal workings of any of the interoperating systems, so has different content. As we have repeatedly seen, ontologies are largely about social reality, representations of institutional facts. Different players are empowered to make the speech acts creating different classes of institutional facts. Much of what goes on in the participating systems is *epistemological*, asking for and receiving information about a changing social reality (recall the distinction between performative and informative speech acts in Chapter 2). Since the participating systems communicate generally solely by exchange of messages, the classes and *choreography* (which messages can and can not

be sent in particular states of complex systems of exchanges) of the messages is also part of the ontology. Although an ontology and a model are managed by similar technologies and have similar representations, they deal with different issues.

Further, it is common for ontologies to be assembled at least partially from standard components. The Z39.50 ontology of Chapter 4 is a generic query system which can be applied in many applications, each of which has its own set of use attributes. The EDI system sketched in Chapter 1 is a component of very many electronic commerce exchanges. The SIC classification of Chapter 4 is very widely used. Assembling an ontology partly from standard components introduces a number of problems not normally encountered when building data models. We will see other standard components when we consider RDF and OWL in Chapters 10 and ll. Chapter 15 describes some of problems in serving an ontology with imported components.

5.3 Complex Objects

Our information systems are often about things which have a complex structure. To take a more extensive example than a driver's license, for many purposes an aircraft (referred to below as aircraft A) is a complex thing. To its manufacturer's parts management system, aircraft A appears as a bill of materials, represented as a whole/part structure. To the airline using it, aircraft A appears as a collection of seats, represented as a set/instance structure. To the manufacturer's production scheduling department, aircraft A appears as a process and a set of completions of process steps (the parts are the various stages of production). To a safety inspection system, aircraft A appears as an ordinary physical object together with a set of properties.

Aircraft A is the same thing in all cases. Further, these various views of aircraft A interact as the respective information systems interoperate. Person P may wish to book a seat on an aircraft which has been recently inspected, which was manufactured in a given time period and has a particular type of avionics system. Perhaps person P was the engineer who designed the system and supervised its manufacture during the given time period, and wants to feel how it works in routine use when recently tuned according to specifications.

Each of the information systems dealing with aircraft A represents it as a collection of fragments. In the same way that we need to know how the parts of a license to drive are held together, our agents need to know how these fragments come together to represent the entire aircraft. This is called the problem of *unity*. Our agents also need to be able to identify aircraft A in its various representations. This is called the problem of *identity*.

Complex objects are of course often represented in conceptual models of information systems. For example, Figure 15 shows two classes of complex objects modeled in the ER notation, *Event* and *Team*. An instance of *Event* has parts which are instances of *Race*, and an instance of *Team* has parts which are instances of *Competitor*. Relationships among the complex objects are between instances of their parts.

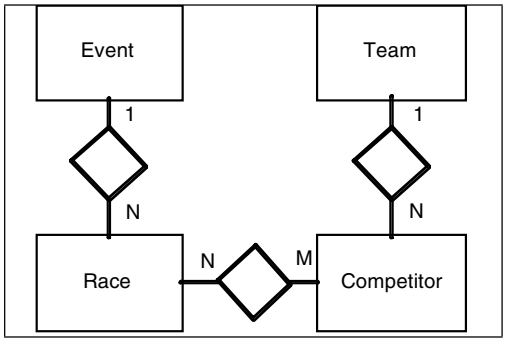


Figure 15. Model of two complex object classes

Notice that the ER modeling system does not distinguish between the parts of complex objects and the links between the parts of one and the parts of the other. Further, since it doesn't make sense to have a race without an event nor a competitor without a team, a representation of the model in a relational database schema would often be something like

R(EventID, RaceID, TeamID, CompID) (1)

In which the complex objects are not explicitly represented as distinguished objects at all. It is left to the user to formulate queries which bring the parts together.

If the conceptual model of Figure 15 represents an ontology supporting many interoperating systems, the representation of the objects and their structure becomes more important. If the events are to be held at a particular sports stadium, the stadium management's information system is interested only in the events, their duration (the stadium is booked for an agreed period of time), and their type (the stadium's configuration may depend on what types of events are to be held). They don't need details of the races, nor any information at all about the teams or competitors. Similarly, the accommodation service needs to know about teams (because the team management makes the booking and pays the bills), but perhaps only the number of competitors as a property of the team, rather than the individual athletes. On the other hand, the event organizers need to know the details of the races and of the competitors. All these views need to be integrated for the systems interoperation to work.

We conclude that since an ontology models classes of objects external to any implementation, it is much more valuable to explicitly model the structure of complex objects than it typically is in a single system.

5.4 Representation of Identity and Unity in a Single Information System

5.4.1 Single System Example

The main purpose of this Chapter is to see what we need to do to represent unity and identity in an ontology supporting interoperating information systems. Before we look at these issues in ontology, we will look at how these are represented in the more familiar conceptual models underlying single information systems.

Figure 16 shows a fragment of an information system supporting an order-entry/ order fulfillment style application. The model is ER, with the notation showing *many-to-one* relationships as having an arrowhead at the *one* end (*One-to-one* relationships

have arrowheads at both ends.) The lower part of the figure is an abstract model of the process from a more general ontology. It shows a distinction between the interaction with the customer (*Transaction*) and the processes that the organization must undertake to fulfill the order (Activity). The notational convention is that the entity type at the arrowhead end of a relationship is mandatory, while the entity type at the undecorated end is optional. For example, in order for there to be a valid instance of *Purchase Transaction*, there must already be a valid instance of *Customer* for it to be related to. However, there can be instances of *Customer* associated with no instance of *Purchase Transaction*. We will say that *Purchase Transaction depends on Customer*.

From the Figure, we see that *Purchase Transaction* depends on *Customer* and *Product*, while *Activity* depends on *Purchase Transaction* and *Product*. In terms of Searle's framework of institutional facts (see Chapter 2), the purchase transaction between an instance of *Customer* and the supplier in respect of a instance of *Product* counts as the customer buying the product in the context including: the identifier of the customer is an instance of *Customer* in the supplier's database, as is the identifier of the instance of *Product*; and that the customer instance is not barred from participating because of poor credit; and that the instance of *Product* refers to products which are in stock and to which the supplier holds title. The context for making a speech act recorded in *Purchase Transaction* includes the existence of valid instances of *Customer* and *Product*.

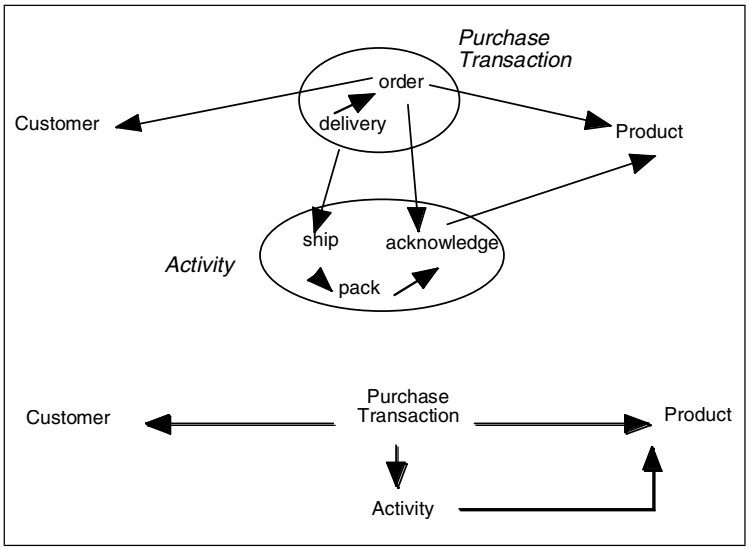


Figure 16. A fragment of a refinement of an order-processing system. Upper part shows decomposition into parts of *Purchase Transaction* and *Activity* from lower part

The upper part of the Figure shows a refinement of the lower model, showing both *Purchase Transaction* and *Activity* as processes with parts. This is a fragment of a more specific ontology. The more general *Purchase Transaction* is implemented in the refinement as an *Order* followed by a *Delivery*, while the more general *Activity* process is implemented by first acknowledging the order (*Acknowledgement*), then packing it (*Pack*), then shipping it to the customer (*Ship*). One of the constraints on the refinement of the more general ontology is that each object in the more specific ontology must behave according to the pattern of the more general object it refines. So the behaviour

of the parts of *Purchase Transaction* must refine the behaviour of the whole, as must the behaviour of the parts of *Activity*.

Relationships can also be refined into parts. The relationships between *Order* and *Acknowledge*, and between *Delivery* and *Ship,* are both refinements of the relationship between *Purchase Transaction* and *Activity*.

This refinement of classes and properties illustrates that institutional facts can be viewed at coarser or finer granularities, depending on the purposes of the viewer, underscoring the importance of the whole-part relationship.

5.4.2 Axioms for Identity and Unity

We begin our investigation with formal criteria for unity and identity developed in the OntoClean project, and by seeing how these formal criteria apply.

As we will see below, there are some kinds of object to which one or the other of identity and unity do not apply. However, we will begin with objects to which both apply. The OntoClean axioms for unity are developed using theory of wholes and parts, called mereology[3]. A whole is a collection of parts, and we say

P(x, y) is true if x is a part of y (2)

An object x is an integrated whole if it has a division into parts that is a closed system under a suitable equivalence relation[4] *sameWhole* called the *unifying relation*. The axiom given is

∀y(P(y, x) -> ∀z(P(z, x) <-> sameWhole(z, y) (3)

That is to say, every part of x in a certain division is related by the unifying relation to every other part in that division, and to nothing else. In our driver's license example, the unifying relation for the paper license record is "being present on the same card", while in the database record, the unifying relation for the parts in different tables is "dependent on a particular license number". All and only the parts for a given license satisfy these relations. If the whole is an html form *f.html* and the parts are the individual fields, then the unifying relation for field *x* is “in file *f.html*”. All and only the fields in *f.html* are parts of that form. If the whole is a student’s academic record, then all of its parts have the student identifier as a key or foreign key. The unifying relation for the academic record for student *y* is “record having key or foreign key value *y*”.

OntoClean also has an identity axiom which requires another equivalence relation called an *identifying relation*. A property R carries an identity criterion iff

R(x) and R(y) <-> x = y (4)

That is to say if x and y have the same property R, they are the same object. For example, an object has the property *time interval* with three attributes, a starting time t_s, and ending time t_e and a duration $d = t_e - t_s$. One identity relation is *same-starting-time-and-same-ending-time*, which would be used for say timetabling classes. Another identity relation is *same-duration*, which would be used for say cost estimation.

Our intent in this example is to see how these ideas apply to the problem of interoperating information systems. The variables in the preceding formulas therefore are intended to be instantiated by various business objects handled by the information systems. The business objects are all either institutional facts or brute facts associated

[3] See Mereology in Explanations, Appendix C
[4] See Equivalence Relation in Explanations, Appendix C

with institutional facts, therefore the classes we are thinking about are generally representations (brute facts which are records of institutional facts).

A key characteristic of the representations we are going to deal with is that they can be reproduced, so that a paper form is equivalent to a filled-in screen form is equivalent to a pattern of magnetic domains on a computer disk. This raises some subtle issues. In the non-computerized world, the representations of some institutional facts cannot be copied. For example, a copy of a cheque is not a cheque. A copy of a banknote is a counterfeit, so not money. A copy of a passport is not a passport, but can be evidence that a passport exists.

In information systems, the problem is often to restrict copying. Almost any representation in an information system can be copied from keyboard to screen to memory traces in a computer to magnetic domains on disk to printer and so on, within a restricted domain. Some representations, for example product catalogues, can be copied very widely. This is one of the reasons why explicit representations of identity are so important in information systems – it is essential to prevent the proliferation of copies from confusing the institutional facts being created and stored in the systems.

More deeply, the institutional facts exist only in their representations, so a complex fact will exist only as a complex of related representations. This is why we need to pay attention to unity and identity.

5.4.3 Application to the Example – Individual Objects

We first look at a representation of a single class (entity type in ER terminology) which is the source of no many-to-one relationships, in the example of Figure 16 *Customer* and *Product*. (We will call these universal targets *independent classes*.) The semantics of the ER model are based on set theory, so the principal interpretation of the representation of the class is as a set of instances. An instance is represented as a tuple of atoms, called *attribute values*.

The tuple of values can be constructed in a number of ways. One way is to store the tuple directly, in which case the unifying relationship is said to be lexical rather than logical. That is to say, unifying relationship for the tuple of attribute values is “being present in the same tuple”. We have already encountered this kind of unifying relation, in the Department of Motor Transport's tables from which the license is reconstructed. The paper copy of the driver's license also has a lexical unifying relation, namely "being present on the same piece of paper".

Another way to represent a tuple of values is to construct the tuple as a view (a query, which is equivalent to a logical formula). The assembly of records from different tables to reconstruct our driver's license is an example. This is a logical construction, but still based on the most primitive tuples which associate the identifier of the object with the institutional fact represented by a particular attribute value.

So the unifying relationship for a representation of an institutional fact is nearly always at least partly lexical.

In database theory, an instance of a class is identified by a subset of attributes, whose pattern of values is unique for each tuple (called a *candidate key*). It is very common for the system to be designed with an attribute intended as a candidate key, such as *driver's license number* or *product number* or *customer identifier*. This attribute is generally called a *primary key*. So in the case of *Product*, an identifying relation can often be represented logically as

$$\text{same-object}(x, y) =_{df} \text{product number}(x) = \text{product number}(y) \quad (5)$$

Remember that we are referring to the business object, which is characterized by its representation.

However, the identifying relation (5) is not sufficient. It works well for the internal working of the organization, but is not generally a satisfactory way of representing identity outside the particular organizational context; in the same way that the driver's license number is not a sufficient identifier for replacing the license document. Just as we were looking to replace the driver's license issued to me and presented my personal identification, a customer is looking for citric acid 99.9% purity in 200 litre drums, not product CA999-200. A customer knows their own name, address, and other details, not generally their customer number. Even though the product identifier is in practice used as the primary key, there must always be a candidate key composed of attributes whose values are known by all parties in potential interactions with the information system.

There could be a number of such candidate keys. For example, in the United States an organization has built an exchange for purchase of electronic parts. In this system, the electronic parts have besides their distributor-specific product identifier a global identifier called *Federal Supply Clauses* which can be used by agencies of the United States Government, but not by other purchasers. So the identifying relationship depends on context.

OntoClean uses the terminology that a class has a property that *carries* identity or unity if all instances of that class have values for the property which can be used as the basis for an identity or unity relation, respectively. As we have seen generally in previous chapters and will see more formally in the next Chapter, classes generally occur in subsumption hierarchies, where each more specific level can introduce new properties. The class at which a property is first introduced is said to *supply* the property. If the property is the basis of an identity or unity relation, then the class is said to *supply* identity or unity, respectively.

5.4.4 Objects That Are Classes

Our discussion of products raises a significant point. What the information systems designer thinks of as an instance can be from the perspective of the ontology a class, not an instance. The information system designer's concept of an instance of *Customer* is also an instance in the ontology, but the same is not true of *Product*. In particular, product number CA999-200 or "citric acid 99.9% purity in 200 litre drums" both identify product classes rather than instances. There can be an amount of the product in various places in the warehouse, in various stages of manufacture, and in various shipments to customers, not to mention waste or spoilage. Our identity criteria as so far presented do not go deep enough.

In some cases individual instances of products are routinely identified. A car or a computer or a piece of consumer electronics normally has a serial number plate on it, which identifies it as an instance of that class. This identifying plate is often the only difference between otherwise indiscernible objects of the same product number.

The citric acid example isn't like that, though. One need for identifying an instance of this product class is to identify a lot of the product subject to a particular transaction or activity. In this business context, the particular lot of the product which the transaction is about is determined quite late in the process, at the time of shipment. Often it is not determined until a particular collection of drums is loaded onto a truck. The drums may not be distinguishable one from another - their only labeling may be with the product class. So the particular instance of the product in this case would seem

to be "the drums loaded onto truck VVV1234 for delivery on 20 March, 2002 under delivery manifest number 77883322 to Acme Ltd".

However, another need is to identify the product in inventory at a particular time. Some products are produced in identified batches (e.g. pharmaceuticals) or have use-by dates, which can be identified as wholes of an amount of matter. However, many products are simply flows into and out of inventory. The product may be packaged in one form or another like our drums of citric acid, or may be stored in bulk (oil, say).

Note that unity can appear at some levels of granularity and disappear at others. Following an example of apples in the spirit of the purchasing application, the product class may be “Apple – Granny Smith in cases of 80”. The cases carry no identification. The cases carry unity, because we can tell what is in which case, but not identity. So at the level of maintenance of inventory, shipping and receiving, the product is also considered to be bulk. However, a particular customer (a restaurant, say) may have ordered a small number of cases in a particular transaction. The cases may be able to be identified now (by location in the truck, say) during the delivery, but then go into the restaurant’s cool store and merge with other cases in bulk again. When a case is opened, the apples in it are not identified, but an apple served to a particular customer of the restaurant may be identified, again by what amounts to packaging. To this customer, the apple carries identity not only of class, but also individuality, since it is the (only) apple served them at that time. It also has a *topological unity* (its parts are all contiguous), so is an individual in this context.

With the apple example, we have identity that depends very much on context. The last version had the apple served to a particular customer at a particular time. When we are engaged in an activity we often identify objects with reference to the engagement: my apple, the apple I just served, that apple. This sort of identification is called *indexical*. When we are interacting with an information system, the objects are often identified only by class. For example, in an e-commerce application, we might have a command “move product to shopping cart”. This identifies indexically the product we have on the screen and the shopping cart associated with our account. In these situations, the system will have an alternate identification in a broader context. The product is identified by product number, say, and the shopping cart by our account identifier, so that internally the command is "move product P to shopping cart S", but we see only the command with indexically identified objects. Drag and drop commands are indexical.

This discussion of objects missing one of identity or unity gives us a taxonomy. In terms of OntoClean, the class *Customer* is a collection of individuals. (An *individual* has both unity and identity). The class *Product* is a collection of classes, which may or may not consist of individuals. The individuals may be distinguished only by name. But we have seen that the classes in *Product* often carry identity (as an instance of the identified class), but not unity; or unity but not identity (objects are anonymous) so that the lots of product are not able to be identified other than by the container they may be in. We can call this identification pseudo-identity. We can call these kinds of object (which carry class identity but not unity or unity but not identity) *bulk* objects as distinguished from *countable* objects (which carry both identity and unity so yield individuals).

We return to bulk objects in Chapter 12.

5.4.5 Dependent Classes

Dependent classes like *Purchase Transaction* pose a somewhat more complex identity and unity problem. An instance of a dependent class requires the prior existence of instances of the classes depended on (in this case *Customer* and *Product*). The representation of objects belonging to dependent classes will often include identification of the objects depended on. Semantically, the independent classes define the most stable aspects of the operation since they must have instances in order for the other classes to have instances. The dependent classes represent records of the more dynamic aspects of the operation. The institutional facts referred to by the populations of the dependent classes have as some of their properties the identifiers of the records of other institutional facts. These other institutional facts must exist for the speech act creating the present institutional fact to be validly framed, so these records are important properties. A driver's license requires identity documents. An invoice may require a purchase order, a delivery advice and a delivery acknowledgement.

So, although it is common for invoices to be identified internally by say an invoice number, a convenient way to obtain an identifying relation for wider contexts is to build on the identifying relations for the objects represented by the relevant independent classes. (This is essentially the formal mechanism of *weak entities* in ER modeling.)

Note, by the way, that in our example, *Purchase Transaction* is dependent on both *Product* and *Customer*. There is a third dependency, on *Activity*, but this dependency is subject to an additional integrity constraint whereby *Purchase Transaction* is transitively also dependent on *Product*, but the product instance reached in either path must be the same. To complete a purchase transaction an activity must have already been completed. The purchase transaction and activity both depend on instances of Product, but the instances must be the same. The products shipped must be the same as the products packed. This third dependency does not add anything to the first two.

If each of the relationships between *Purchase Transaction* and *Product* and respectively *Customer* were one-to-one, then the concatenation of the identifiers of *Product* and *Customer* appropriate to the context would be sufficient to identify an instance of *Transaction*. In general, however, the relationships are many-to-one, so that for each pair of *Customer* and *Purchase Transaction* instances there can be many instances of *Transaction*. An additional local candidate key is also needed. Internally one might use a sequence number and externally for example *Date*.

5.4.6 Identification of Parts

Figure 16 shows in its upper half a refinement of the model in the lower half. The class *Purchase Transaction* has been refined into two parts, *order* and *delivery*; while the class *Activity* has been refined into three, *acknowledge*, *pack* and *ship*. How do we identify instances of these new classes?

We have looked at identification of instances of *Purchase Transaction* and *Activity*. One general way to identify parts of a whole is to use the identifier of the whole with the addition of a local part identifier, as in weak entities in the ER system. That method won't work in this case, since the whole is not represented. If we think of the lower part as being a representation of an earlier stage of the design and the upper part as being a representation of a later stage, then in developing the conceptual model of the information system, the lower half is replaced by the upper half when the system

design reaches that state of refinement. When we are thinking about *order* and *delivery* there is no longer any entity *Purchase Transaction*.

In operation, the system will generate linked instances of each of the part classes. If we happen to want to look at combinations of instances of *order* and *delivery* as instances of the more general *Purchase Transaction*, we will create them by a query. So the whole only comes into existence when all of its parts do. Further, we might be interested in partially completed wholes. In these cases, only some of the parts exist, so the whole does not exist at all. So we can't identify the parts with reference to the whole.

What we can do is take advantage of the fact that the instances of the part classes are linked by foreign keys. Every instance of *delivery* is linked to an instance of *order*, every instance of *pack* is linked to an instance of *acknowledge*, and every instance of *ship* is linked to an instance of *pack*, which is in turn linked to an instance of *acknowledge*. If we identify the instances of the parts in some way, we can derive an identification of the whole. Every instance of *Purchase Transaction* includes an instance of *order*, as does every instance of partially completed *Purchase Transaction*. Similarly, every instance of *Activity* includes an instance of *acknowledgement*, as does every instance of partially completed *Activity*. So one way to identify the whole is by one of its parts which always exists: *Purchase Transaction* by *order* and *Activity* by *acknowledgement*. Identifying a whole by one of its parts is called *metonymy*.

Metonymy is quite common. News reports of activities of national governments often refer to the country by its capitol, as "Washington said …", "The response from Tokyo was …". Governments are often identified by the leader, as "the Bush administration", "the Blair government".

5.4.7 Application to the Example – Classes

Individual objects have now been identified as instances of their respective classes. We have seen that some individuals can also be seen as classes. This leads us to ask how the classes are identified in general. Classes represent sets of business objects, and a set of business objects is often the subject of discourse.

Institutional facts are completely characterized by representations of their records, so have a limited set of properties. In particular, there is no way to distinguish an invoice from a credit note without the record of the class of institutional fact. So it is hard to see that the class of invoices is anything other than the subclass of institutional fact whose representations contain an attribute called "type" which has the value "invoice".

So a class of institutional fact is represented by a class, which formally is a set whose members are instances, but the normal practice is to define the class *lexically* (by form, in this case as members of a named set) rather than *logically* (as subsets of a superset satisfying a predicate expressed in logic or SQL). The same class can have many different populations – indeed much of the code in an information system is devoted to updating the populations of classes. That is to say, much of the code is devoted to updating the representations of members of sets of business objects. So we can think of a class as a sort of container.

In practice, systems designers identify classes in several different ways. If the primary design vehicle is the ER model, the class (entity) is represented simply as a name on a graphical element, which has graphical connections to representations of names of attribute value sets and graphical representations of relationships with other

classes. Often, the representation is a translation of the conceptual representation to a relational database table scheme, where the class name becomes the table name, and the relationships and attributes column names, the former labeled foreign keys.

Other ways also are used. Sometimes several classes are combined, possibly redundantly, in a single table (universal relation) – often as a view. In this case, to identify an instance of the representation of a class, the programmer needs to know which columns contain the preferred candidate key and which columns contain the attributes and foreign keys associated with that class. It is also possible to represent a conceptual model in a single table with four columns: *class*, *tuple*, *attribute*, *value*. Each row of this table contains a single value of a single attribute of a single tuple instance of a single class. (The tuple in this case is often identified by an arbitrary number which is not itself the value of an attribute.) Here, the programmer needs to know the name of the class in order to retrieve its instances.

So in practice the unifying relation for a class is something like an SQL statement (the statement, not the result), and the identifying relation is either the name of the class or the names of columns which are known by the programmer to constitute the representation of the class. Classes are therefore typically identified lexically.

5.4.8 Application to the Example – the System as a Whole

Finally, we take another step upwards in scale and look at the system as a whole. This system is a single system, which could be one of many operated by the same organization, so would have a name, say *Acme Corp Ordering System* (henceforth ACOS).

Either system in Figure 16 has a unifying relation, namely "is a part of ACOS', but that unifying relation is not represented in the structure. The unity of the system is maintained by its operational environment: it is located on a particular cluster of servers operated by a particular company, compiled from modules held under a certain version in a certain repository, and so on. As an isolated system, it has no practical need for its unity to be represented internally. Both its unity and identity are represented indexically.

However, it is not difficult to represent a unifying relation. One way is to supply an additional class *ID* called the identifying class, with exactly one instance and to require a (unique) many-to-one relationship *id* from each class in the system to the identifying class. We can parameterize the identifying class by the name of the system, represented as ID_x. A class used as an identifying class for an entire system in this way is called a *terminal object*. The unifying relation of the refined model is a refinement of the unifying relation of the abstract model. In this case, we have

$$\text{isAComponentOfSystem}_x(z, y) = \exists z,y\ (\text{id}(z, \text{ID}_x) \text{ and } \text{id}(y, \text{ID}_x)) \quad (6)$$

where the unifying relation is also parameterized by the name of the system.

Given the unifying relation (6), we can represent an identity criterion for "is system ACOS" as

$$\exists x\ \text{ID}_x \text{ is a terminal object for system ACOS} \quad (7)$$

Because they are both parameterized by the name of the system, the unifying and identifying relations are both lexical rather than logical.

This is all a bit academic, of course, since we have already established that in the isolated system we have taken ACOS to be, there is no practical need to represent either unity or identity. However, as we see below, unity and identity become very

important when we start seeing ACOS as a member of a community of interoperating systems.

5.5 Interoperating Systems

Now that we have come to an understanding of the issues of identity and unity in a familiar single-system environment, we now turn to the less familiar environment of interoperating systems where there are additional things to consider. Remember we are thinking about the agent programs that execute the interoperations. This is why in the previous section we looked at how to identify an entire system. Each agent must identify itself to the other, so must know its own identity as well as the identity of the other.

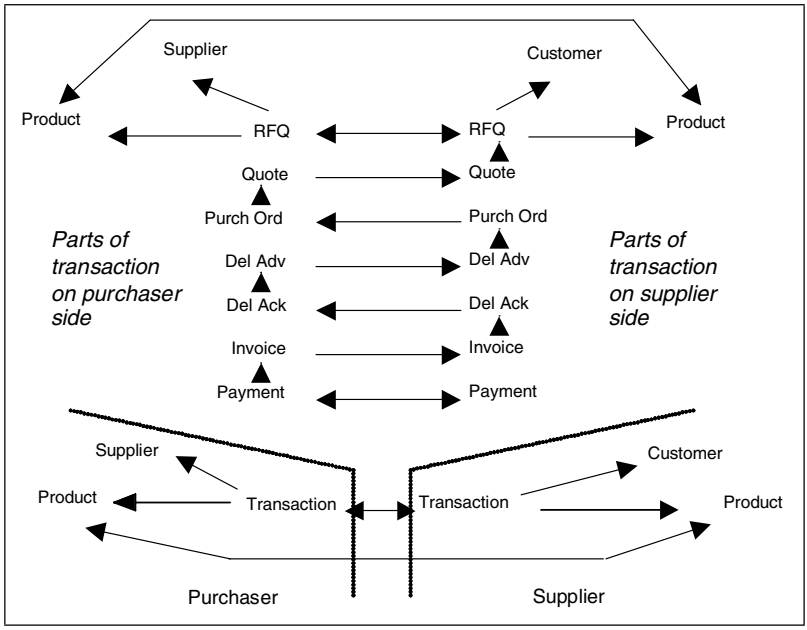


Figure 17. Purchasing and Supplier system interoperating: lower by *Transaction*; upper by *Transaction* refined into parts.

An order processing system would naturally interoperate in an electronic commerce environment with a purchasing system operated by another organization. Further, neither system would generally expose itself in its entirety to the other. Each system in the interoperation would see something like in Figure 17. As with Figure 16, the upper system is a refinement of the lower. The lower system shows a purchasing system (left) and a supplier's order processing system (right). Both systems interoperate via *Transaction*. If the underlying system has an identifying terminal object, then the view can have one, too, so that the view is unified and identified in the same way as the underlying system. (If necessary, the two can be differentiated by differentiating the terminal objects.)

Concentrating on the lower (abstract) system, note that an instance of *Purchase Transaction* is dependent on both *Supplier* and *Product* on the Purchaser side, and also on *Customer* and *Product* on the Supplier side. (Note that *Purchase Transaction* here is

not the same as in Figure 16, but is closely related.) We focus first on the relationships involving *Product* on each side.

Each of the Purchaser and Supplier system has a class called *Product*. They are not the same, as the Purchaser can buy from many suppliers, and the Supplier can sell to many purchasers. But they will have a common subclass, consisting of the products that purchaser buys from that supplier.

Product in each system is an independent class. In our (simplified) system, a transaction involves an instance of *Product* on the Supplier side and an instance of *Product* on the Purchaser side. In fact, it is physically the same product on each side, moving say from one warehouse to another. There is therefore an issue of identity – the identifying relations for *Product* on each side need to be correlated, generally by a one-to-one mapping. The mapping need be neither injective nor surjective – one company may purchase only some its products from a given supplier, and purchase only some of the products offered by any given supplier. This mapping implicitly provides an extensional unifying relation for the subclass of products shared between a given customer and supplier. The correspondence between product identifiers is an identifying relation for that subclass.

This method of federating product catalogs by pair-wise correspondences works well for one-on-one interoperations. It can in principle be extended to an exchange. Each player makes a correspondence between their own catalog and the catalogs of each of the other players with whom they interoperate, and the exchange's product catalog can be constructed from the transitive closure of each of the pairwise correspondences. When two players interoperate for the first time, a particular correspondence may be able to be derived if each of the players interoperates transitively with others, so there is some economy gained from the fact that there are many players in the exchange. The transitive closure of the correspondences functions as an identifying relation for the product on the exchange.

But the accumulation of pairwise correspondences is not very satisfactory at the exchange level. There may be thousands of players. An individual player may have to establish pairwise correspondences many times before gaining benefits from transitive closure. Worse, there is no reliable way to be sure that all products are in fact uniquely identified. There may not have been enough pairwise correspondences established that the transitive closure includes every player who deals in that product. A new product which has never been traded is just about invisible.

For this reason, exchanges generally will establish a priori an exchange-level system of identifying objects traded. In this way a player needs only establish a correspondence between their own catalog identifier and the exchange's identifier. This is what ISBN numbers and bar codes on products are all about. An exchange-wide or industry-wide system of identifiers takes work to establish and continual coordination to maintain. The system is a system of institutional facts and needs bodies authorized to make the corresponding speech acts.

The relationships involving *Supplier* and *Customer* raise additional issues. It is normal for the Purchaser system to have a class *Supplier* (as in Figure 16), and for the Supplier system to have an analogous class *Customer*. However, what is an instance of *Supplier* for the Purchaser is the entire system of the Supplier, and what is an instance of *Customer* for the Supplier is the entire system of the Purchaser. As we have seen in the previous section, in isolated systems neither the unifying nor identifying relations for the whole system are generally represented. For interoperation, however, they must be. Each system must have a data structure which allows them to tie together their

identities with various partners. Anthropomorphically, we might think of this as a primitive notion of "self". In the previous section, we used a terminal object for this purpose. So we have that the terminal object for each system must be mapped into an instance of a class of the other, in order for the two systems to interoperate. My system must know how the other system identifies me, and the other system must know how my system identifies it.

We move now to the refinement of the *Purchase Transaction* classes in the upper part of Figure 17. A business transaction is an institutional fact normally created in a series of speech acts organized into a process, using a semantic protocol like one of the EDI standards. This means that to carry out the interoperation the abstract system needs to be refined, so that *Purchase Transaction* has several parts. In the example of Figure 17, we have the interaction as proceeding from a request for quotation (*RFQ*) issued by the purchaser, through *Quote* by the supplier, *Purchase Order* by the purchaser, *Delivery Advice* by the supplier, *Delivery Acknowledgement* by the purchaser, *Invoice* by the supplier to *Payment* by the purchaser. In practice, of course, the interaction can be much more complex, involving many more exchanges of different classes, and the sequence need not be linear.

In the example, each side keeps copies of all messages, very likely linked to other aspects of their respective systems outside the view. Each message participates in a many-to-one relationship with an earlier message. The first message (*RFQ*) is shown with a one-to-one relationship between the parties – the relationship from Purchaser to Supplier represents acknowledgement by the supplier that a communication sequence has been established.

The refinement is conceived of as an articulation of the whole of *Purchase Transaction* into parts. We therefore need to consider the unifying and identifying relations required.

An instantiation of *Purchase Transaction* is as a process, so its parts come into existence one by one over possibly considerable time. (We do not need to take clock time into account, simply sequence.) Further, in the example the whole is not represented in the refinement, only the parts. A whole transaction instance is represented by the assembly of all of its parts. So besides unity and identity, we need to consider existence. We focus first on identity and unity.

Atomic parts (an *atomic part* has no parts itself) of an instance of *Purchase Transaction* are instances of *RFQ*, *Quote*, and so on. The first part to come into existence is *RFQ*, and it can be identified in the same way as we have discussed for the unrefined *Transaction*, as an instance logically by the relationships with either *Customer* and *Product* or *Supplier* and *Product* for Supplier and Purchaser respectively, together with an agreed disambiguating attribute like *Date*. Since there needs to be agreement between the parties on the identification of *RFQ*, there needs to be an intersystem identity relation, which can be supplied by including all three of *Product*, *Supplier* and *Customer* relationships, taking advantage of the identification of the Purchasing system as an instance of *Customer* and the Supplier system as an instance of *Supplier*. Of course the identity relation for the *RFQ* instance also includes the lexical identification of it as an instance of the *RFQ* class. So we would say that an instance of *RFQ* is identified by the combination of purchaser, supplier, product and date.

The other parts as they come into existence can be identified by their possibly indirect relationship with *RFQ*, if necessary including a further local identifier based on

Date or *Message-ID* or some other attribute whose scope includes the contexts of both parties.

Dependence on *RFQ* provides a convenient unifying relation for the whole transaction.

We now consider how to identify the whole, assuming we have a complete collection of parts. One obvious way is to employ *metonymy*, using the identifier of *RFQ* as the identifying relation of the whole. However, the various parts of the transaction come into existence over time, and in practice may never come into existence. At any given time, the populations of the classes refining *Purchase Transaction* will contain all sorts of incomplete transactions. Some of these incomplete transactions may be stopped. For example it often occurs that a purchase order is not ever issued in respect of a quote, and it sometimes happens that a customer does not pay. It therefore may suit the organizations to refrain from identifying a whole transaction until a part comes into existence which usually leads to completion, say *Purchase Order*. But of course the identification of a not-completed transaction does not guarantee that it will ever be completed. We will return to this issue in the discussion of subclasses and subsumption in the next Chapter.

5.6 Comment on the Examples

Some of the ontologies in Chapter 4 have whole-part relationships:

- Z39.50: in Figure 4 the property *in* whose domain is *ResponseRecord* and whose range is *ResultSet* can be interpreted as instances of *ResponseRecord* being wholes with parts instances of *ResultSet*. The semantics are the set-instance relationship. The identifying relation for *ResultSet* is *hasName*. *ResponseRecord* is a dependent class, with identifying relation a combination of *LocalIdentifier* and *hasName* composed with *in*. The unifying relation is that the *ResponseRecords* X and Y are parts of the same whole if *hasName*(*in*(X)) = *hasName*(*in*(Y)).
- Tic-Tac-Toe: in Figure 8 the class *Game* can be interpreted as a whole with parts instances of *Cell* via the *has* property. The identifying relation for *Cell* is the conjunction of *atPl*, *atH* and *atV* (instances X and Y of *Cell* are the same if *atPl*(X) = *atPl*(Y) and *atH*(X) = *atH*(Y) and *atV*(X) = *atV*(Y)). We noted in Chapter 4 that there is no facility in the ontology to identify instances of *Game*, so there is only one instance Exists G *Game*(G). The unifying relation is the property *has*. If the ontology were modified to include identified instances of *Game*, then *Cell* would become a dependent class in the same way as *ResponseRecord* in Z39.50.

We will return to the part-whole relationship in the examples at the end of the next Chapter, because the other examples are bound up with the class/subclass relationship.

5.7 Summary of Identity and Unity

We now know how to represent complex things in the environment of interoperating information systems. We keep the various aspects of the representation together with unifying relations and keep track of different things with identifying relations. We have seen that it is not always possible to find both a unifying relation and identifying

relation, so that some things in the environment are what we might think of as bulk rather than distinguishable objects. We have seen that our information systems deal logically both with definite objects and with classes.

5.8 Key Concepts

Complex objects need both **identity** and **unity**. Identification and unification can be **logical** or **lexical**. **Independent** classes can help identify **dependent** classes. A whole can be identified **metonymically**. Identity and unity depend on **context**. Identity can be **indexical**. **Countable** types have both identity and unity**. Bulk** classes lack one. A Bulk class can have a **pseudo-identity** if it is associated with a countable **container** class.

5.9 Exercise

1. (relevant to points 5 and 6 of the major exercise) Consider the Olympic fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise.

a. Which classes are independent and which are dependent? How are instances of the dependent classes identified?

b. Describe the shared complex objects. What are their parts? What are their identifying and unifying relations? Are the unifying relations logical or lexical? Are there any objects identified indexically? Any identified metonymically? Are the independent classes used in unifying relations? (Make plausible additions if necessary.)

c. Describe the bulk classes in the ontology, showing how they are bulk classes. What containers are they in? Do the containers give pseudo-identity?

5.10 Further Reading

Much of the material from this Chapter comes from [8].

6 Subclasses and Subproperties

We have used the concept *class* in several places, relying on its usual interpretation in database theory and practice, and in logic. We now look more deeply into how classes work, and the conditions under which our agents can make use of them.

6.1 Subclasses and Subsumption

A *class* in database or logic (often called *type*) is a set of things which is represented according to a schema. In an ER model, entity types are classes, as are some relationship types. In a database a class is represented in possibly several tables held together by unifying relations and identified using identity relations. The representation of a class in a logic system is similar to its representation in a database. All of the rectangles in Figures 4 and 9 represent classes. The set of web pages indexed by a particular search engine is a class. In particular, the set of services covered by an e-commerce exchange is a class.

6.1.1 Properties and Values

The world an agent sees consists of *individuals*. An individual is represented to the agent as a collection of property values. These individuals are generally complex, consisting of organized collections of parts. A part of a complex individual is also an individual. Some of the issues involving complex objects have been discussed in the previous Chapter.

But the individuals are generally also organized into *classes*, sets of individuals with common characteristics. In this case the individual is said to be an *instance* of the class. A class of complex objects will consist of objects whose parts are instances of other classes.

An individual is represented as a instance of a class by a subset of its property values. We generally associate the properties relevant to a class with the class. In database or logical representations, properties are represented by attributes in relations.

In a tradition going back to Aristotle, it is common to think of a class as having two sorts of property values, essential and accidental. An *essential* value of a class is a property value that a thing must have if it is to be a member of that class. An *accidental* value is one that a thing may or may not have. It is common to use everyday objects as examples to distinguish essential from accidental properties. *Having a brain* might be considered an essential property value of a human being, while *having red hair* might be considered accidental. *Having feathers* might be considered an essential property value of a bird, but *being able to fly* might be considered *accidental*.

The parts of a complex object, as represented in the population of its unifying relation, are a value of a property of that object. So it is possible for something to be an *essential whole* if its unifying relation is an essential property. Once the whole exists, it is always a whole. In Figure 17, a completed transaction is an essential whole, since all of its parts must exist in order for the transaction to be completed. The sequence of

RFQs handled by the exchange up to a given time on a given day is an accidental whole, since at a later time the earlier sequence is simply incorporated into the later one and is no longer thought of as a whole.

The distinction between value and property can be seen in this example. *Having hair of a colour* is a property of the class human being, the boolean *ability to fly* is a property of the class bird. The concept of property applies mainly to accidental values, since an instance of a class can have different values drawn from the value set of the property. An essential value must be the same for all instances of a class, by definition. However, an essential value of a class might be an accidental value of a superclass, so it makes sense to think of properties here, too. *Having a brain* is a boolean property of a superclass of human being that includes dead people as well as living. *Having feathers* is a boolean property of a superclass of bird, say vertebrate.

In keeping with our emphasis on interoperating information systems, we will confine our examples to institutional facts. In the application of Figure 16, a *Customer* might have properties *CustomerID*, *Name*, *Address*, *Balance Due* and *Credit Rating*, but of these only *CustomerID* is essential. A particular customer always has the same value of *CustomerID*. The others are accidental, since they may change.

Note that in an ERA model, these attributes would all be indicated as mandatory. Mandatory means that an instance must have a value of that property, but not necessarily the same value for all instances nor the same value over time for a particular instance. *Customer* might have optional properties *Contact Name*, *E-mail address* and *Date of last purchase* (a customer might be registered with an account but not have yet made a purchase). An instance of a class might not have a value drawn from an optional property, so an optional property cannot be essential.

We have that an essential property for a class is necessarily present for every instance of that class, and always has the same value. There is a related, but stronger, kind of property called a *rigid* property which is not only essential, but sufficient to determine the class. Rigid properties and the distinction between rigid and essential properties are common in the natural sciences. A species of tree, say the Ironbark common in Australia, is identified by a pattern of properties called a key. These properties are largely characteristics of its flowers. The properties used in the key are not only essential, but also rigid. The Ironbark has other properties which are essential but not part of the key, so not rigid. One such property is that its wood is extremely hard and resistant to rotting and to termites, so that it makes excellent fence posts which can last in the ground for 50 years or more. The same kind of identification is used for minerals. For each mineral there is a standard test which constitutes its rigid properties. Let us say that a diamond is characterized by the fact that it is extremely hard and also burns – its rigid properties. A diamond also glitters if it is cut as a gem, but this property is not used to characterize it as a diamond. It is an essential but not rigid property. A rigid property is something like a candidate key for a class. All instances of a class have the same values drawn from the class' rigid properties.

We have already encountered the problem of rigid properties for institutional facts without saying so in the discussion of identification of classes in Chapter 5. The class of an institutional fact is generally indicated by a representation of its name. The only way to identify a record as a record of an invoice is that the record says so. If the invoice is a paper record, written on the record will be the word "invoice" or some equivalent. If the invoice is in a computerized system, it will be stored in a relation with a name "invoice" or some equivalent. Alternatively, it may be stored in a relation with a name like "financial transaction" one of whose attributes having a name

"transaction type" with value "invoice", or something like it. Similarly customers' details are known as customer details because they appear in a customer file, and product details are known as such because they appear in a product catalog. In other words, rigid properties for classes of institutional facts are largely lexical, and are often essentially the same thing as the name of the class.

6.1.2 Subclasses and Metaproperties

Classes are interesting mainly because we can have subclasses. Class B is a *subclass* of class A if every instance of class B is also an instance of class A. We say in this case that class A *subsumes* class B, and that A is a *superclass* of B. The population of instances of a subclass is a subset of the population of instances of a superclass. So if *Book* is a class, *OntologyBook* can be modeled as a subclass, and if *FinancialTransaction* is a class, *Invoice* and *Payment* can both be modeled as subclasses. Every *OntologyBook* is a *Book*. Every *Invoice* and every *Payment* is a *FinancialTransaction*.

In Chapter 5 we looked at complex objects, where wholes are composed of parts. As we saw, we often have a class of whole objects, whose parts are drawn from different classes of parts. An airline has many aircraft, so it would be usual to model a class *Aircraft*. Each aircraft consists of a number of parts which are bookable seats, so it would be usual to model the situation with a class *BookableSeat*, and the connection with a property *seatOf* whose domain is *BookableSeat* and whose range is *Aircraft*.

The class of wholes, *Aircraft*, is a broader concept than the class of parts, *BookableSeat*, so there is a temptation to model *BookableSeat* as a subclass of *Aircraft*. This model is incorrect, because it fails what we might call the ***instance test***: ***is every instance of the subclass*** (*BookableSeat*) ***also an instance of the superclass*** (*Aircraft*)***?*** Clearly not.

There are two different ways of defining subclasses, called defined and declared[5].

The defined subclass is most common in database systems. A *defined* subclass is the result of a predicate on the superclass, so it is defined logically. We may have two subclasses of *Customer*: *Individual* and *Organizational*, distinguished by the value of an attribute called *Customer Type*. We may have two subclasses of *Product*: *Inventory* and *Service*, distinguished by whether or not there is an inventory record for the product. We may have two subclasses of *Student*: *Passed* and *Failed*, distinguished by whether the grade is respectively greater than or equal to 4 or less than 4. The predicate defining a subclass gives a rigid property for the subclass. An object which is an instance of *Student* and which has *grade* greater than or equal to 4 is an instance of *PassedStudent*, and every instance of *PassedStudent* is an instance of *Student* with *grade* greater than or equal to 4.

Many classes have mandatory properties. A purchase must be associated with a buyer and a seller. A student must be associated with a degree program. That a property that is mandatory for a class can be expressed as an existentially quantified predicate:

- if an object is an instance of *Purchase*, there exists an instance of *Customer* associated with it via the *buyer* property and there exists an instance of *Supplier* associated with it via the *seller* property

[5] This terminology comes from description logics

- if an object is an instance of *Student*, there exists an instance of *DegreeProgram* associated with it via the property *enrolledIn*.

If no other class has a particular combination of mandatory properties, the existentially quantified predicate functions as a class definition, and therefore the predicate being true is the rigid property for the class.

A *declared* subclass is determined by a declaration, that is lexically. We may have separate tables for invoices and payments but declare them both as declared subclasses of *Financial transaction*. Hierarchical term lists found in classification systems, such as the SIC discussed in Chapter 4, are considered declared subclasses.

The distinction between declared and defined is blurred by the practice common in database applications for a class to have a property which tags instances with their subclasses, often called *tag attributes*. *FinancialTransaction* might be the domain of a property *finType* whose range is the enumeration {'invoice', 'payment'). So it is possible to collect the subclass *Invoice* with an SQL statement selecting on the property *finType*. A subclass defined in this way is much closer to being a declared subclass, since the decision as to which subclass an individual belongs is made outside the application. The decision is simply recorded in the tag attribute. In the examples above, the subclasses of *Student* and the subclasses of *Product* are clearly defined, since their definition is based on other properties, while the subclasses of *Customer* are distinguished by a tag attribute so fall into the gray area.

In Chapter 5 we introduced the concept of metaproperty: unity – how we tell which parts belong to which whole, identity – how we tell one object from another, necessity – what we know about an object once we know its class, and rigidity – how we know whether an object belongs to a given class. Unity, identity, necessity and rigidity are closely related to subsumption. These metaproperties are called *formal* properties. They represent how the property is used with respect to a class. Paying attention to the effect of subsumption on the formal properties gives a much sounder basis for the reasoning of agents, as we will see.

We say that a class *supplies a property* if it holds for that class and not for any superclasses, while a class *carries a property* it inherits from a superclass. So if we think of a class *Person* with a subclass *Student* which has a further subclass *ResearchHigherDegreeStudent*, we have that the class *Person* supplies the property *name*, which is carried by *Student* and *ResearchHigherDegreeStudent*. The class *Student* supplies the property *studentID*, which is carried by *ResearchHigherDegreeStudent*. Finally, *ResearchHigher DegreeStudent* supplies the property *thesisAdvisor*. The same terminology applies to values and to the meta-properties of values and properties: identity, unity, essentiality and rigidity.

The OntoClean method annotates properties and values with their formal meta-properties:

1. identity +I - How do you identify an instance of a class?
2. unity +U - How do you tell which parts belong to which instance of a class?
3. essentiality +E - What must be true if an individual is an instance of a class?
4. rigidity +R - How do you tell what class an individual is an instance of?

In general, if the annotation of a property is +A, where A is one of the meta-property annotations, this means that in all instances of the class the property necessarily has meta-property A, that is to say the property is used for purpose A in that class. So the annotation +E on a property of a class says that the class carries

essentiality. Recall that essentiality means that all instances of the class have the same value of that property.

There are weaker attributions of meta-properties. If a property is annotated -A, then the meta-property A does not necessarily hold for all instances. It might hold by accident, but not necessarily. A property annotated -E is not essential for all instances, so that different instances can have different values. The sign '-' indicates 'not', so -R is 'not rigid', -E is 'not essential', -I is 'not identity', -U is 'not unity'. If a property is annotated ~A, then the meta-property necessarily does not hold for any instance. A property annotated ~E can be updated, so that the value associated with any instance may change. The sign '~' indicates 'anti', so ~R is 'anti-rigid', ~E is 'anti-essential', ~I is 'anti-identity', ~U is 'anti-unity'. We have a concept of *strength* of a meta-property. For meta-property A, +A is stronger than –A, since A is necessary implies that A is possible. We also have that ~A is *inconsistent with* +A, since necessarily not A is inconsistent with necessarily A.

The relationship between +, -, ~ E/ R and mandatory/optional is

1. +R must be mandatory (note +R -> +E)
2. +E must be mandatory
3. -R, -E, ~R, ~E can be optional

6.1.3 Metaproperties and Subsumption

These meta-properties are property restrictions which regulate subsumption. A property restriction on a class can never be weakened in a subclass. That a property is rigid (+R) for a class is the restriction that it has the same value in all instances, so must have the same value in all instances of all subclasses. Further, no instance of a class not a subclass has that value for the property. (Of course a property rigid for a class is ~R for any subclass, but will always be essential +E.) That a property is essential (+E) also is the restriction that it has the same value in all instances, so must have the same value in all instances of all subclasses. But the restriction +E is weaker than +R because the property can be essential for classes other than subclasses.

Being a instance of *Student* and having *grade* greater than or equal to 4 is rigid for *PassedStudent*, so is true for no class other than a subclass of *PassedStudent* (say *ExcellentStudent* instances of which have *grade* = 7). But having borrowing privileges at the University Library is an essential property for *Student*, and also for *UniversityStaffMember*. The class *UniversityStaffMember* is not a subclass of *Student*.

A property which carries identity (+I) can identify any instance of a class, so must be able to identify any instance of any subclass. A property which carries unity (+U) is the basis of keeping together the parts of any instance of the class, so must be able to keep together any instance of any subclass.

By similar reasoning, a subclass can never have a property ~E, ~I or ~U if the superclass has the property +E, +I or +U respectively. However, a subclass can have a property +E, +I, +U if the superclass has that property ~E, ~I, ~U respectively. An anti-essential property can be updated in all instances. An anti-identity property is not capable of being the basis of an identity relation for any instance. An anti-unity property is not capable of being the basis of a unifying relation for any instance. However, a superclass can have an antirigid property which is rigid for a subclass. The conventional *Customer Type* attribute is anti-rigid for the class *Customer*, but rigid for the subclasses *Individual Customer* and *Organizational Customer*, having e.g. the

value 'I' and 'O' respectively. The same is true for essentiality. A property ~E for a superclass can be +E for a subclass.

On the other hand, a subclass can always strengthen a property. A non-rigid or non-essential property can be rigid or essential in a subclass, A non-identity or non-unity property can be the basis for an identity or unity relation in a subclass.

Rigid properties behave according to a somewhat different pattern. A rigid property for a class is the property by which an individual is determined to be an instance of that class. So a rigid property for a class is analogous to the name of the class. In a system of classes, if a property is +R for one class it must be ~R for all other classes.

For example, assume we have a class *Student* at a university in Australia. Some students have passports, and for those students the combination of country and passport number is an identifier. But not all students have passports, so the property *passportNumber* is -I for *Student*. But we may have a group of students from say Malaysia who form a subclass *MalaysianStudent*. For this subclass, we might use *passportNumber* as an identifier (+I).

Some of the consequences of these regulations are:

The set-instance relationship is not a superclass-subclass relationship. The property that identifies a set cannot identify its instances, since a set can have many instances. The set of invoices has the +I property e.g. "table name *invoice*", which is ~I for individual invoices since it is always the same. The instances have the +I property e.g. "invoice number", which is ~I for the set of invoices, since it has many different values. The same argument works for the famous Clyde the elephant example. If we have a knowledge representation system where classes are associated with an *is-a* relationship and everything is a class, then we may have a class *Clyde* where Clyde's history is held and a class *Elephant* where what it means to be an elephant is held. One of the things it means to be an elephant is that elephant is a species of animal. So what it means to be a species is held in another class *Species*. The only relationship between classes is the is-a relationship, so we have *Clyde is-a Elephant* and *Elephant is-a Species*. The question is whether *is-a* can be interpreted as the subclass/superclass relationship. In particular, can we make the inference *Clyde is-a Species*?

The answer is obviously not, on semantic grounds. The Ontoclean metaproperty mechanism gives us a formal reason why not. The genus/differentia which identify the species elephant are the same for both Clyde and Jumbo, indeed all elephants, so cannot identify Clyde. The identifying markings of individual elephants cannot identify the species, since by definition they are not the same for all members of the species. So the property which is +I for *Species* is ~I for *Elephant* (indeed it is +R).

The meta-relationship is not a superclass-subclass relationship. Imagine we have an ER diagram with no subclasses, so all the entities represent classes with rigid properties. One might be tempted to create a single superclass of which all the entities are subclasses on the basis that all the entities have rigid properties. The superclass would be the set of entities with rigid properties. This cannot be a superclass of the other classes for essentially the same reason that the set-instance relationship fails, namely that the rigid property for the proposed superclass is that the proposed subclasses all have a rigid property. The individual classes are identified by which property is rigid for them. This criterion cannot identify instances of the individual entities.

The whole-part relationship is not necessarily a superclass-subclass relationship. A whole may have essential properties which the individual parts cannot have. Assume

that a whole must have at least two proper parts, P_1 and P_2, so that *has* P_1 and *has* P_2 are both essential properties of the whole. But they cannot both be essential properties of either part, since by definition a thing cannot be a proper part of itself. We might be tempted to interpret the refinement of *Activity* in Figure 16 as a superclass-subclass relationship, but cannot since an essential property of a completed instance of *Activity* is a completed *Acknowledge* followed by a completed *Pack* followed by a completed *Ship*, and this cannot be a property at all of any of *Acknowledge*, *Pack* or *Ship* taken separately.

The table-view relationship is possibly a superclass-subclass relationship. If a view includes a key of the table, then each instance of the view identifies an instance of the table. If the view excludes only optional attributes, then it must include all +R +E +I and +U properties represented in the table among all the mandatory attributes included. A row of the view is a valid row of the table, so that under these circumstances the table-view relationship is a superclass-subclass relationship.

6.2 Defined Classes Versus Declared Classes

To someone used to defined classes, say someone from the information systems conceptual modeling community, it will seem strange to see problems in the relationship between subsumption and identity. A defined class is defined by a predicate on its superclass which defines a subset of the instances of the superclass. No attributes are lost. The most that can happen to an attribute is that an optional attribute of the superclass can be either mandatory or always absent in the subclass. Identifying attributes are always preserved. Consider in Figure 18 the superclass Student, which has the identifying attribute (+I) *student#* and the optional attribute *e-mail address*. The two subclasses *E-student* and *S-student* are defined by the predicates *e-mail address* is present and *e-mail address* is absent, respectively. Both subclasses of student are still identified by student#, while for *E-student e-mail address* is mandatory and for *S-student e-mail address* is necessarily not present.

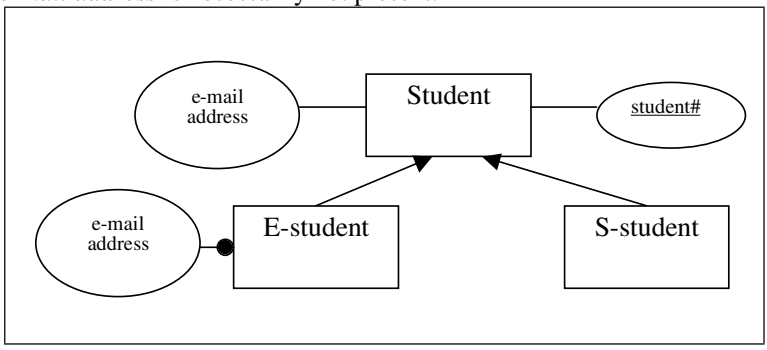


Figure 18. Defined subclass

So problems of distribution of identity over subsumption come from the unconstrained use of declared subclasses.

The issues with unity are a little subtler. Identity can never be affected by a predicate, but parts can. If a property is the basis of an essential unifying relation (+U) for a class, then an instance of that class must have instances of all the parts called for in the unifying relation. Therefore, so must every instance of every subclass. For

example, if we create a class *EDI-Transaction* from Figure 17, and stipulate the unifying relation that an essential property *a* is that every instance must have at least one instance of the seven classes of parts (*RFQ*, …, *Payment*), then every instance of every subclass must also have at least one instance of each class of part. Since the database structure involves several tables, a selection can be constructed that excludes some of the part classes, so that *a* no longer holds. Say we retain only the top table *EDI-Transaction*, *RFQ* and *Quotation* to obtain a table of quotations. This selection preserves identity, but not unity. We cannot have the population of these views be a subclass of *EDI-Transaction*.

For the same reason, if we have a population of partial transactions which have only proceeded to the Quotation stage, the class defined by this population cannot be declared as a declared subclass of EDI-Transaction. The unifying relation *a* of the superclass (+U) does not apply to the subclass. The relation *a* is ~U for the subclass. It never applies.

Similarly, if a relation is anti-unity (~U) for a superclass, then it can be an essential unifying relation for none of its instances, therefore it can not be an essential unifying relation (+U) for any subclass. For example, we can say that the relation *c*, that a student has passed all courses, never includes all the speech acts necessary for a student to be awarded a degree. There is always a further request to the proper authority for the granting of the credential, which must be agreed to by the authority. In some fields this requires an additional examination (bar exams, medical specialist exams, etc.). Suppose we have a superclass *Qualified* with subclasses the various qualifications and kinds of qualifications. One might be tempted to include a subclass of people who had completed the coursework and but not sat the examination (call the class *Ready for qualification*). Our relation *c* is +U for the new class. Therefore making *Ready for qualification* a subclass of *Qualified* would violate the unity constraints on subsumption, even though it satisfies the identity constraints (*student#* is +I for all subclasses).

6.3 Interoperation Example

We will now illustrate our new terminology by re-interpreting the electronic commerce interoperation example from Figure 17. That example developed the one-to-one interoperation between the two. Here we want to expand the example to consider the whole exchange, with multiple purchasers and multiple suppliers.

The first step in that example was the integration of the *Product* classes on the purchaser and supplier sides. Since we know that the physical products are the same on both sides, the problem was the correlation of the identifying relations in each of the two systems. Further, since we are looking at views we don't see possible complex structure within the two *Product* classes, only the identifiers. The rigid property of the *Product* class in each system is the product catalog name qualified by the name of the company, while the identifying relation makes use of an anti-essential property in each case. The speech act necessary to establish the relationship is to set up a table pairing the equivalent identifying properties, both at the class and instance levels. This need be done by one party only, often the purchaser; although there are industries where the supplier would establish the relationship, and some where both would.

The second step was the purchaser becoming established as an instance of *Customer* in the supplier system, and the supplier becoming established as an instance

of *Supplier* in the purchaser system. As with *Product*, speech acts are required, which can be interpreted as establishing a pairing between the identifying terminal object of the purchaser system with an instance identifier for *Customer* in the supplier system, and between the identifying terminal object of the supplier system and an instance identifier of *Supplier* in the purchaser system. Both instance identifiers are a rigid property of the corresponding class associated with the anti-essential instance identifying property. These correspondences will be kept by both sides, since they each need to know who they are dealing with.

We now look at these relationships at the level of the exchange. At least for its own accounting purposes, the exchange will maintain classes for both purchasers and suppliers. An exchange may have thousands of players, each of which can play more than one role. For example, a company may be a supplier with respect to stationery but a purchaser with respect to telecommunications services. The establishment of instances of *Customer* and *Supplier* separately by each pair of players clearly does not scale. The exchange will generally maintain a register of players, and will additionally allow players to register for the roles they intend to play. So the speech acts establishing correspondences will be between individual players and the exchange, rather than among the players.

The role classes in the exchange's system may have overlapping subclasses, namely identifiers of organizations who play multiple roles. We have seen that some companies may be both purchasers and suppliers. The anti-essential instance-level identifying property of an overlapping subclass will be an equivalence class of the identifying property of the superclasses where the individual system identifiers are the same. Concretely, if Acme Ltd is a supplier, there will be a pair <Acme Ltd, 338> in the exchange's supplier table and a pair <Acme Ltd, 901> in the exchange's purchaser table. Acme Ltd will be identified as both a purchaser and supplier by the name 'Acme Ltd' appearing in both pairs, so that the exchange will know that supplier 338 and purchaser 901 are the same organization. This is a defined subclass. The exchange may or may not publish its supplier, purchaser and overlapping classes.

In a similar way, the method of associating product instances by pairwise correspondences among purchasers and suppliers does not scale. Exchanges will generally establish an exchange-level product catalog, and the individual players will establish correspondences between their internal catalogs and that of the exchange. This is the sort of function served by ISBN numbers or bar codes on retail products. It is very common in such catalogs for there to be a taxonomy of subclasses of product, which would need also to be developed in a series of speech acts. It might be possible to take advantage of existing ontologies. For example, books published in the USA have Library of Congress in-publishing catalog information attached, which gives a classification and a set of subject descriptors, while chemicals and pharmaceuticals have industry-standard nomenclatures and taxonomies as do many categories of primary products.

A consequence of the autonomy of players in the exchange is that each player will decide individually which products they will deal in, and which roles they will take. Since one of the functions of the exchange is to facilitate interoperation among players, the exchange may keep a record of which player has decided to offer for sale which product, and possibly which player has expressed interest in purchasing which product. These records can be viewed as a collection of subclasses of the product.

For example, in the bookseller exchange of Chapter 1, each bookseller is a player. Besides the universal booksellers like amazon.com, barnesandnoble.com and

borders.com there are thousands of specialized or small booksellers, some selling used books. Putting a book up for sale in the exchange is a speech act performed by a particular bookseller. So if *Experience and Nature* by John Dewey is offered for sale by PhilosophyBooks.com, the details of that book will be recorded by PhilosophyBooks.com. If *Fundamentals of Database Systems* by Elmasri and Navathe is offered for sale by DatabaseBooks.com, the details of that book will be recorded by DatabaseBooks.com. Both of these books are also offered for sale by the universal booksellers, so details of both will also be held by amazon.com, barnesandnoble.com and borders.com. They might also be offered for sale by OntologyBooks.com, so recorded there. The universal booksellers will also offer *Harry Potter and the Goblet of Fire*, so will record its details, but none of PhilosophyBooks.com, DatabaseBooks.com nor OntologyBooks.com will. And so on.

We have a class *Book*, with subclasses

- BooksSoldByAmazon.com
- BooksSoldByBarnesandnoble.com
- BooksSoldByBorders.com
- BooksSoldByPhilosophybooks.com
- BooksSoldByDatabasebooks.com
- BooksSoldByOntologybooks.com

The first three subclasses, the books sold by the universal booksellers, will be equivalent to the superclass, while the last three will be proper subclasses. *BooksSoldByPhilosophybooks.com* and *BooksSoldByDatabasebooks.com* will be disjoint (no book is offered for sale by both). *BooksSoldByOntologybooks.com* will overlap with all the others, but there will be books offered for sale by PhilosophyBooks.com or DatabaseBooks.com which won't be offered by OntologyBooks.com, and vice versa.

Are the subclasses defined or declared? The question is given a particular book, how do you know which booksellers will stock it? For example, we know that *Fundamentals of Database Systems* and *Harry Potter and the Goblet of Fire* will both be sold by the universal bookstores, since all three of those subclasses are defined as being equivalent to the superclass *Book*.

Let us assume that there is a property *subject* whose domain is Book and whose range is the subject classification system defined by the U.S. Library of Congress. If PhilosophyBooks.com undertakes to offer for sale all and only books with the value "Philosophy" for the *subject* property, then *BooksSoldByPhilosophybooks.com* is a defined subclass.

If DatabaseBooks.com sells only books with the value "Database Systems" for the *subject* property, but only the books judged relevant to a particular group of companies in its neighborhood, then we can tell that it won't sell *Harry Potter and the Goblet of Fire* because that book has the value "Children's Fiction" for the *subject* property. But we can't sell whether DatabaseBooks.com offers *Fundamentals of Database Systems* unless we ask. The subclass *BooksSoldByDatabasebooks.com* is *partly defined* (with a necessary but not sufficient condition) and partly declared.

Conversely, DatabaseBooks.com may decide to carry all books with the more specific subject "Database Systems: Oracle". So we can tell in advance that *Oracle for Dummies* will be offered for sale by DatabaseBooks.com even though we have to ask about *Fundamentals of Database Systems*. In this case the subclass

BooksSoldByDatabasebooks.com is *partly defined* (with a sufficient but not necessary condition) and partly declared.

Finally, if OntologyBooks.com sells whatever books its management thinks might be interesting to ontology developers, then the only way to find out whether it sells a particular book is to ask it. The value of the *subject* property doesn't help. The subclass *BooksSoldByOntologybooks.com* is declared.

Note that in practice none of the subclasses would be fully defined. If *Book* is the class of objects with ISBN numbers, then there are many privately published books, or books published by very small publishers, that are not sold by the universal booksellers, so even the universal booksellers would handle at best partially defined subclasses of *Book*. For the same reason, it would be unlikely that PhilosophyBooks.com could sell all instances of *Book* with *subject* property "Philosophy". The Library of Congress by law must be given copies of all books published in the United States, no matter if the publisher is too small to deal with barnesandnoble.com.

A taxonomy based on product properties is orthogonal to a taxonomy based on supplier and purchaser, so the resulting subclass structure of a unified product catalog would be a faceted system.

Finally, the exchange operates by means of the message classes indicated in Figure 17. The seven message classes indicated in the upper part of Figure 17 may be considered as a decomposition into parts of the message class *Purchase Transaction* of the lower part of the Figure. Both the parts and the whole are classes, but we have seen that the parts are not subclasses of the whole.

Figure 17 shows that the whole is dependent on *Supplier*, *Customer* and *Product*, and that the parts are dependent on each other. We have seen already that the parts come into existence over time, and in fact may never come into existence.

6.4 A More Complex Example

Our first extended example was based on an e-commerce system, where the subclass structure of the product catalog is based on what in database theory is called horizontal partitioning. Each product instance has the same structure as every other. Which subclass a product belongs to is determined by a mixture of declared and defined classes which from a database perspective are essentially views based on selection. We will now look at an example where the products are complex, and the views are more vertical – that is different views have different properties.

The present case is based on aircraft A which appeared as an example in Chapter 3. To its manufacturer's parts management system, essential attributes might include a serial number, model number, and the entire bill of materials (on the theory that an aircraft with different parts would be a different model). To the airline using it, aircraft A appears as a collection of seats. Essential attributes might include a configuration identifier, a serial number, and the seat identifiers. To the manufacturer's production scheduling department, aircraft A appears as a process with the various stages of production. Essential attributes here would be a model number, an in-production serial number, and the various production stages with anti-essential but mandatory *completed/ date completed* or *incomplete/ date scheduled* status indicators. To a safety inspection system, essential attributes might be model number, serial number, date into service, and anti-essential but mandatory number of flying hours since last service for

each of a number of subsystems. Even though the aircraft is a physical object, all the attributes recorded are the result of speech acts, so the aircraft is an institutional fact in each of the systems.

The first thing we notice is that there is no single attribute common to all the systems. Aircraft model appears in all but the airline, where its place is taken by a configuration identifier. We assume that the same model can have different configuration identifiers and different models the same. Serial number appears in all but the production scheduling system, where its place is taken by an in-production serial number local say to the factory producing the aircraft. How do we integrate these systems at all, not to mention how do we use our machinery to help us do so?

If we are to integrate these systems, we must integrate them around a class "aircraft". The first thing we need to do is to establish that there is something "outside" any of the systems that corresponds to the class *Aircraft* in each of the individual systems. This is not necessarily the case.

Consider a company with two divisions: order taking, which performs routine sales tasks; and accounting, which among other things collects debts. Order Taking has a *Customer* table including people who the company will sell to. Accounting has an *Accounts* table, which includes some people who have long-outstanding balances owing. The order taking division will no longer accept orders from these latter people, so they are not in the *Customer* table. Rather, they are in a *Blacklist* table so they can be prevented from establishing themselves as new customers. Further, some customers pay cash, so do not appear in the *Accounts* table. Is there a class outside either system to which both *Customer* and *Accounts* correspond? Clearly not. There are people in *Customer* not in *Accounts*, and people in *Accounts* also in *Blacklist* so necessarily not in *Customer*. The two tables are related at the instance level, so integration is in fact possible for many purposes, but not on the basis that there is a common class corresponding to both *Customer* and *Accounts*.

A plausible integration is shown in Figure 19. Since cash customers, account customers and bad debts are all people, we need a more inclusive common superclass *Person*. This more inclusive common superclass has as subclasses the people of interest to Accounts and the people of interest to Order Taking. There is a common subclass *Account Customer*, but no class to which both *Customer* and *Person of Interest to Accounts* correspond.

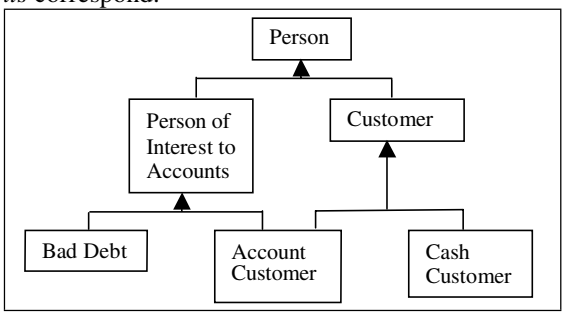


Figure 19. Integration of Order Taking and Accounts

Returning to the Aircraft example, it should be clear that a given physical aircraft A is the brute fact which counts as institutional facts in each of the systems. So we should in principle be able to link the institutional facts resulting from aircraft A. Since, however, there will be many aircraft, before we look at that problem we need to

establish *Aircraft* as a class independently of any of the systems. We will make some more or less plausible assumptions, which we may be able to relax later.

Aircraft are large and very expensive pieces of equipment, so it may be economically feasible for the various organizations concerned to adjust their information systems to include only the right instances of aircraft. Let us include all the aircraft which are actually in service by each of the airlines. The manufacturers create views on their parts management systems and production scheduling systems which present in total all and only those aircraft in service by any of the airlines. For the production scheduling system, this perforce includes only completed aircraft. The safety inspection system also plausibly can create a view that includes all and only aircraft presently in service by the participating airlines.

So we can plausibly make the assumption that there is a class *Aircraft* which includes all and only those aircraft which are actually in service by any of the participating airlines. Recall that we couldn't do this for the *Customer/Accounts* example, at least not so easily.

If we are going to have a class, it needs a rigid property. Since we are making a class of institutional facts, we can expect that its scattered records are somehow brought together in a structure with the name *Aircraft*, so that is its rigid property is lexical. We also need an anti-essential property which we can use to identify the instances. Assuming that two aircraft from different manufacturers never have the same serial number, *Serial Number* is a natural candidate, since it is included in the records of three of the four systems. The view of the production control departments of the manufacturers includes only completed aircraft, so that it is possible to augment the records by a table giving the correspondence between the in-process serial number and the final official serial number. This requires an additional speech act, but one can expect that the manufacturers' records would contain sufficient information to make it. *Serial Number* therefore can be the basis of an identifying relation.

Each of the systems is based on a complex institutional fact regarding the aircraft. Each of the constituent facts of each complex fact is tied together by dependency on *Serial Number*. Dependency on *Serial Number* is therefore the unifying relation for each of the institutional facts.

In contrast to the e-commerce application, where each subclass of *Product* has the same properties, in the present example each system deals with a different set of properties. Can we think of each system as being concerned with a different subclass of *Aircraft*?

We have that a view can be considered a subclass if all the mandatory properties of the superclass are in the view, necessarily including the rigid property for the class and the anti-essential instance identifying property. So far, we have the rigid property for the class and the identifying property, namely *Serial Number*. There are many properties mandatory for one or more of the systems, but no property shared by all. What happens if we consider all the other properties to be optional properties of the superclass?

If properties are optional, they may not be present. Let us imagine that a particular aircraft is no longer or not yet in passenger service, so does not have a collection of (bookable) seats. Let us assume that the physical aircraft exists, and that it has its normal complement of physical seats. Does it make sense to say that it does not have (bookable) seats?

Remember that the contents of all the information systems consist of institutional facts, not brute facts. It is indeed a brute fact that the aircraft in question has physical

seats. But the speech act has not been made that these physical seats count as bookable seats in the context of the aircraft being in passenger service for airline Y. Therefore the institutional fact does not exist. We can see that it is valid in this case to think of the institutional fact as optional. It is therefore valid to treat the vertical fragments of the complex institutional fact *Aircraft* as subclasses.

We have previously remarked that an instance of the complex institutional fact *Aircraft* consists of many simpler institutional facts. It would therefore be convenient to think of the contents of the various interoperating systems as parts of the whole. Does it make sense to think of them as subclasses and parts at the same time? The argument given above is that the whole-part relationship is not a superclass-subclass relationship if the whole necessarily has proper parts. Since all the parts are optional, the whole of an instance of *Aircraft* does not necessarily have any proper parts, so there are no essential properties of the whole which an individual part cannot have. It therefore makes sense to think of the contents of each of the interoperating systems as parts of the aircraft as well as subclasses.

That we can view an instance of *Aircraft* as a whole with no essential parts allows us to remove the assumptions made earlier that an instance of *Aircraft* identified by *Serial Number* be present in each of the interoperating systems. Each system can have its population of its subclass/part of the institutional fact *Aircraft* independently of any of the others. Successful interoperation between any two systems requires simply that those two systems each have an instance of their respective subclass/part identified by *Serial Number*. If an instance is deleted from all the systems, then the institutional fact that the physical aircraft counts as … ceases to exist, even though the physical aircraft may be in a hangar somewhere. The brute fact no longer counts as any of the institutional facts recorded by any of the systems of concern.

6.5 Subproperties

Properties are formally similar to classes. A class is represented as a set of tuples of property values (its extent), at least the values of the properties in its identifying relation. A property is represented as a set of tuples of the values of identifying properties from both its domain and range classes. So both properties and classes have extents which are represented as tuples of property values. A subclass is a class whose extent is a subset of the extent of its superclass. We can subset the extent of a property just as well as the extent of a class, so it makes formal sense to think about *subproperties*.

For example, there are all sorts of properties with domain and range both the class of persons. *Sibling* is such a property. *Brother* and *sister* are both subproperties. As are *oldest sibling* and *younger sibling*. The subset affects the range class, but also the domain class. A person with sisters but no brothers participates in the *sibling* property but not the *brother* subproperty. A youngest child can participate in the *sibling* property but not in the *younger sibling* property.

A more structured set of examples illustrated in Figure 20 is based on a student records system. Assume a property *completedCourses* whose domain is *Student* and whose range is *Course*. Assume further that there is a subclass structure on *Student* given by the degree program in which a student is enrolled, and a subclass structure on *Course* given by the department which offers it. So it would make sense to have a property *artsCompletedCourses* which is the subproperty of *completedCourses*

restricted in domain to the subclass of *Student* consisting of students enrolled in a Bachelor of Arts program. It would also make sense to have a property *completedMathCourses* which is the subproperty of *completedCourses* restricted in range to the subclass of *Course* consisting of courses offered by the Mathematics department. We could further intersect the two subproperties in a property *artsCompletedMathCourses*, which is a subproperty of *artsCompletedCourses*, *completedMathCourses* and since subsumption is transitive, *completedCourses*.

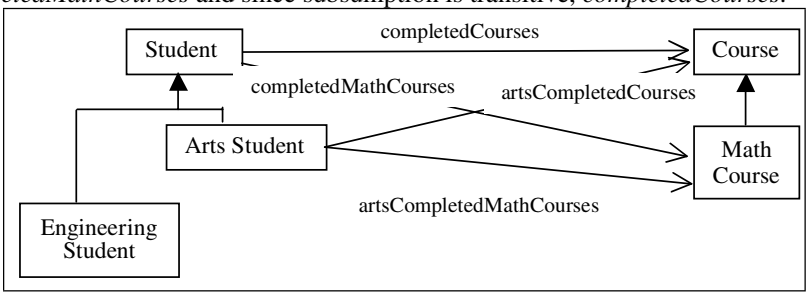


Figure 20. Subproperty structure

Properties often have restrictions. For example a property may be mandatory or optional (an instance of its domain class may or may not be required to participate in the property). A subproperty of an optional property can be mandatory (participating in *completedMathCourses* may be optional for an arbitrary instance of *Student*, but mandatory for the *EngineeringStudent* subclass). But a subproperty of a mandatory property must also be mandatory unless the subproperty is also restricted in range. (If *completedCourses* is mandatory, so must be *artsCompletedCourses*; but *completedMathCourses* may not be.)

Subproperties have been less studied than subclasses, so there is not a discipline comparable to OntoClean for them. The primary test for P_1 to be a subproperty of P_2 is that every instance of P_1 must also be an instance of P_2.

But subproperties are important in ontologies. For example, an e-commerce exchange may have thousands of suppliers with different specializations, like the bookseller exchange described in Chapter 1, and hundreds of thousands of customers with different interests. The exchange ontology would support a property *offersForSale* whose domain is a universal catalog of books identified by ISBN and whose range is {*true*, *false*} (a Boolean property). But there would be an advantage both to sellers and purchasers to introduce a range of subproperties based on genres of book (fiction, non-fiction; within fiction literature, adventure, romance, horror, crime, etc.) Individual suppliers could advertise the subproperties they participate in, or offer to allow customers to restrict their searches to the domains of particular subproperties. Note that this example shows the close relationship between properties and subclasses. The instances of the domain class associated with a particular value of the range class constitute a defined subclass of the domain class.

6.6 Commentary on the Examples

Subclasses are important in some of the examples of Chapter 4.

- Tic-Tac-Toe: the ontology in Figure 8 has many systems of subclasses. The subclasses of *Directions* are declared, while the subclasses of *Cell* and *Row* are

defined by obvious predicates. The subclasses of *State* are also defined by predicates involving the property *evidencedBy* (e.g. *Game* is in subclass *Blocked* of *State* if all instances of *Row* in the range of *evidencedBy* are in the subclass *Mixed*). The subclasses *of* State suggest a collection of subproperties of *hasState* (e.g. *isBlocked*, *isWonByX*, *isWinnable*). (See Chapter 13.)

- Standard Industrial Classification: the ontology in Figure 12 is essentially a subclass system, where the subclasses are declared (an instance of *OrganizationalSubunit* is placed in a subclass by a process not defined in the ontology). The rectangles *Industry*, *IndustryGroup*, *MajorGroup* and *Division* denote disjoint subsets of the set of classifications, but are not a nested system of subclasses. We can tell this because no instance of *Industry* is also an instance of *IndustryGroup*, and so on. But on the other hand the instances of *OrganizationalSubunit* classified by an instance of *Industry*, say *Employment Agencies*, is a subclass of one of the instances of *IndustryGroup*, namely *Personnel Supply Services*, and so on. So we can say that the SIC is a system of subclasses of *OrganizationalSubunit* which is a whole with parts in the class *Division*, each instance of which is a whole with parts in the class *MajorGroup*, each instance of which is a whole with parts in the class *IndustryGroup*, each instance of which is a whole with parts in the class *Industry*.
- SNOMED: the system in Table 1 can be seen as a collection of declared subclass systems. It can be viewed as a whole with the various facets as parts, and further analyzed in a way similar to the SIC.
- Periodic Table: The system of Figure 13 has two systems of defined subtypes whose defining predicates are based on information not shown in the figure (population of electrons in shells). The further subclass system *Element Group* is possibly better thought of as declared, but it would take us too far afield to give a proper account. The whole system can be thought of as a whole with parts given by the subclass systems *Group* and *Period*.
- Dimensions: An instance of *Dimension* (e.g. “distance”) is not an instance of *DimensionSystem* (e.g. “Standard International Units”), nor can an instance of one unit be an instance of another (10 *metres* is not an instance of *centimetres*). The whole structure of dimensions is accounted for by the whole-part relationship rather than by the subclass relationship. We will return to dimensions in Chapter 13.

6.7 Discussion

We have introduced a collection of meta-properties of objects, and shown that these meta-properties can allow us to explicitly define unity of complex objects and identity criteria for objects. These meta-properties allow us to regulate the use of complex object constructors including the set-instance, whole-part and superclass-subclass relationships as well as the meta-level and object level distinctions and the view mechanism.

Using this regulation, we can write agent programs which navigate among fragments of complex objects much more reliably than without, and can more confidently build widely distributed interoperating systems with rich structures permitting interaction among complex intelligent agents.

6.8 Key Concepts

It is common to represent things as **individuals** in **classes**. Individuals have **properties**. Superclass **subsumes** subclass, superproperty **subsumes** subproperty. Properties have **metaproperties rigid**, **essential**, **identity**, **unity** which govern subsumption. Subclasses can be **defined** or **declared**.

6.9 Exercise

1. (relevant to points 7 and 8 of the major exercise) Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a. Describe the classes in the ontology. Give their rigid properties, indicating whether the rigid properties are lexical or logical, and their essential properties. Show a system of subclasses for each class, inventing a plausible system if necessary. Each subclass is either defined or declared. If it is defined, give the defining predicates. If it is declared, tell how objects are classified into the subclass and by which role. Show that the identifying and unifying relations are preserved in the subclass structure. Make plausible additions to the system if necessary.

b. Describe a property (relationship, association) involving at least one of the classes from a. This property should have a subproperty structure. Invent a plausible structure if necessary. Show a population of property instances, including at least one instance of each subproperty.

6.10 Further Reading

A concise source of the key ideas in this Chapter can be found in [9]. A more leisurely exposition is in [10].

7 Formal Upper Ontologies

We have so far looked at ontologies using fairly low-level knowledge representation tools, either one of the conceptual modeling languages or the metamodel based on OWL introduced in Chapter 4. In this chapter we consider the benefits of making use of richer formal ontologies as representation systems.

7.1 Structures So Far Not Enough

In the previous Chapters we have seen that interoperating systems exist in a world of objects, and that these objects can be organized into classes, the classes into subclass structures, and that objects can be organized as wholes with parts. The point of the present Chapter is that the structure so far is not nearly enough.

Remember that our task is to specify ontologies so that we can develop software agents to perform the interoperations more or less automatically. Using the structures we have so far, a world we develop may contain a large number of objects and a rich structure, but it is pretty featureless. An example will make this point clearer.

Consider a human making a shopping trip to a major department store, called say David Jones. In the course of the trip, our human may encounter a variety of objects of different classes. Let us list some of them:

Table 2. Object classes in department store

David Jones itself	Top	Accounts clerk
Displays	Belt	Dressing Room
Camera	Handbag	Ask clerk for price
Camera Case	Shoes	Clerk tells price
Memory stick	Payment Point	Hand over money
Recharging Dock	Accounts Desk	Clerk gives purchase
Skirt	Floor clerk	Clerk gets item from back room

Let us say that the human purchases the *camera*, *case*, *memory* and *dock*; and also purchases the *skirt*, *top*, *belt*, *handbag* and *shoes*. They try on clothes in the *dressing room*, establish an account at the *accounts desk* assisted by the *accounts clerk*, and make the purchases at *payment points* with the help of *floor clerks*. In the case of the clothing the protocol is *ask clerk for price*, c*lerk tells price*, *hand over money* and *clerk gives purchase*. In the case of the *camera*, the floor model is not purchased, so the additional step *clerk gets item from back room* is needed.

Our human sees a rich variety of kinds of classes of object. They know that they can purchase the *camera*, *skirt*, etc, but not the *displays*, *dressing room*, *purchase points*, *clerks*, etc. They know that to make a purchase they must deal with a *floor clerk* at a *purchase point*. Dealing with an *accounts clerk* at the *accounts desk* won't do. If fact they know to deal with the *clerk*, not the *purchase point* directly. And so on.

The human knows all this because they know how to operate in a department store. They generally will have learned as a child in the same sort of way they learn their native language. Should they come from a radically different environment, say one where shopping is done in vast bazaars where each purchase involves intense haggling,

they will need to learn how to operate in a department store. (And vice versa, of course.)

Our ontology representation system can allow us to add some structure to the bare list of classes in Table 2. We know about the class/subclass structure, so can represent:

- *floor clerk* and *accounts clerk* as subclasses of a more general class *store staff*;
- the various transaction classes as subclasses of a more general class *transaction*; and
- *payment point* and *accounts desk* as subclasses of a more general *business access point*.

We also know about the part/whole structure. So we can model:

- *David Jones* as a whole with parts classes *dressing room*, *displays*, *business access point*, *store staff*;
- *Camera system* with part classes *camera*, *camera case*, *memory stick*, *recharging dock*;
- *Outfit* with part classes *skirt*, *top*, *belt*, *handbag*, *shoes*;
- *floor purchase transaction* with parts *ask clerk for price*, *clerk tells price*, *hand over money*, *clerk gives purchase*
- *stock purchase transaction* with parts *ask clerk for price*, *clerk tells price*, *clerk gets item from back room*, *hand over money*, *clerk gives purchase*

And can add the superclass:

- *purchase transaction* with subclasses *floor purchase transaction*, *stock purchase transaction*

Finally, we know about the set-instance relationship, so can model a class

- *Product* with instances the classes *camera*, *camera case*, *memory stick*, *recharging dock, skirt*, *top*, *belt*, *handbag*, *shoes*

(Recall the discussion in Chapter 5 of objects which can be both classes and instances.)

Now assume that you want to write a program to interact with an on-line equivalent of David Jones to make the same purchases. Ignoring the differences between being able to physically see and try on things and working from images, assume you are working from an ontology as in Table 2 with the additional structures specified above.

Your program will include messages from the *purchase transaction class*, addressed to parts of the *David Jones* whole, concerning instances of instances of the *product* class. Your program will attempt to assemble instances of the *camera system* and *outfit* classes, which involves selecting instances from their respective parts classes.

How does the program distinguish procedures needed to deal with the *purchase transaction* class from those needed to deal with the *camera system* or *outfit* class? How does it know to relate the *David Jones* part classes and *product* classes to *purchase transaction* part instances? How does it know to treat the *David Jones* part classes differently from the *camera system* part classes from the *purchase transaction* part classes? Certainly nothing in the ontology helps. The program knows this because you as a human understand the differences from implicit knowledge, and design the program appropriately.

Now suppose instead of writing a program to interoperate with David Jones, you write a program to interoperate with any of the several thousand players in an e-commerce exchange. It is clearly far too expensive to write a separate program for

dealing with each player. The ontology supporting the exchange needs to distinguish different kinds of classes and to represent the different kinds of ways different kinds of classes relate to each other. This Chapter is a start in that direction, but by no means provides a complete solution.

7.2 Upper Ontologies

All of the ontologies we have discussed so far have been conceived of as in the context of some more or less general application domain. A number of people have been developing ontologies which are conceived of as independent of particular applications, including Cyc, SUMO and WordNet. These ontologies attempt to describe how the world is, and come mainly from the artificial intelligence and natural language processing communities.

We saw in Chapters 2 and 3 that information systems are largely concerned with institutional facts, which are enormously variable. Institutional facts depend very heavily on context and on background. An order entry system can be relied on for inventory control only if all and only movements in and out of inventory are recorded in the order entry system. This requires a set of background practices which prevent un-recorded inventory movements, which can be quite difficult to implement. It took a long time for supermarket bar-code readers at checkout to become sufficiently reliable that the record of sales produced could be used as accurate indicators of stock remaining on the shelves. This accuracy also depends on a rigorous control of shoplifting. So even an object as apparently simple as quantity in stock of a particular product can vary enormously in meaning from system to system. It is therefore extremely doubtful that these universal ontologies can be used as the basis for ontologies supporting interoperating information systems. That similar terms can differ significantly in meaning from system to system was called *semantic heterogeneity* in Chapter 3.

The desire for a universal ontology comes from the desire to be able to build systems faster, more cheaply, and more reliably so therefore being able to re-use work previously done. It happens that besides the ontologies already referred to (Cyc, SUMO, WordNet), there are systems called upper ontologies which are formal rather than material. A *material ontology* is concerned with what there is, while a *formal ontology* is concerned with how things appear, what forms they can take. A formal ontology is an advanced knowledge representation system.

Formal ontologies are neutral with respect to content, so are no help with semantic heterogeneity. The stock on hand example above shows that two inventory records can have the same form but be semantically incompatible, while we have seen in the aircraft example of the previous Chapter that the same complex object can be represented as a supertype with subtypes and as a whole with parts, two quite different forms.

A number of formal upper ontologies have been proposed. We will here present some aspects of two of the better established, the Bunge-Wand-Weber (BWW) system and the Dolce system developed by the OntoClean project from which come the meta-properties of Chapters 5 and 6. These ontologies are different, but are compatible with each other. Each emphasizes different aspects of form.

7.3 BWW System

Following is a sketch of the BWW system. It consists of a number of concepts.

1. Thing

The basic unit in the BWW ontology is a *thing*, taken as primitive. A useful way to think about things is that they can have a somewhat independent existence in the world. In the Bookseller example of Chapter 1, the buyers, sellers and books are all things, as are amounts of money. The messages RFQ, Invoice and so on are things. So are speech acts like purchasing. The collection of purchasers, of sellers, of books are each things. So is the entire exchange.

2. Property and Attribute

Things have *properties*. There are no things without properties, and all properties must attach to things. Properties give things a form. We describe things in terms of the properties they possess. The world is made up of things that possess properties.

We need to distinguish between *attributes* and properties. The model upon which ontology is based assumes that properties exist in the world independently of our ability to know them. They are there because nature has bestowed things with properties. Because our knowledge of the world is often imperfect, we use attributes to proxy for or act as a surrogate for properties. Indeed, much of science focuses on teasing out the properties of things.

Attributes are the names we use to represent properties of things.

Our representation language of Chapter 4 has comparable concepts. The BWW *attribute* is our *property* and the BWW *property* is our *value of a property*.

3. State of a Thing

At a point in time, the attributes of a thing have values. For example, the attribute "quantity on hand" of a product inventory thing called "product CA999-200" has a value in litres at a point in time. If we think of the vector of values at some point in time of all attributes that we use to describe a thing, we have the *state* of the thing at that point in time. For example, we might describe product CA999-200 via three attributes: description, quantity on hand, price/litre. CA999-200's state at some point in time might be (CA999-200, Citric acid 99.9% purity in 200 litre drums, 10,000 litres, $1.32).

4. Event

An *event* is a change of state in a thing. The values of one or more attributes of the thing change when an event occurs. For example, CA999-200's state might change from (CA999-200, Citric acid 99.9% purity in 200 litre drums, 10,000 litres, $1.32) (CA999-200, Citric acid 99.9% purity in 200 litre drums, 9,000 litres, $1.32). The value of the quantity on hand attribute has changed from 10,000 litres to 9,000 litres (because a sale has been made of 1000 litres).

Events are sometimes described simply by listing the prior state and the subsequent state. To be complete, however, we should also specify the transformation that brought about the event. For example, the change in quantity on hand might be because of a sale. Alternatively, it might have come about because of an inventory check which discovered only 9000 litres in the warehouse. In other words, we have the same prior and subsequent states but we have different transformations and thus different events. In short, to fully describe an event, we need the prior and subsequent states plus the transformation that evoked the change of state.

In the David Jones example, execution of an instance of the *purchase transaction* class is an event.

5. History of a Thing

A *history* of a thing is a sequence of events in a thing. For example, let's assume that we change the price of product CA999-200. If we disregard the transformation, the following event occurs: <(CA999-200, Citric acid 99.9% purity in 200 litre drums, 9,000 litres, $1.32), (CA999-200, Citric acid 99.9% purity in 200 litre drums, 9,000 litres, $1.40)>. We now have the following two events in product CA999-200:

First event:

<(CA999-200, Citric acid 99.9% purity in 200 litre drums, 10,000 litres, $1.32),
(CA999-200, Citric acid 99.9% purity in 200 litre drums, 9,000 litres, $1.32)>

Second event:

<(CA999-200, Citric acid 99.9% purity in 200 litre drums, 9,000 litres, $1.32),
(CA999-200, Citric acid 99.9% purity in 200 litre drums, 9,000 litres, $1.40)>

A history of product CA999-200 is thus the first event followed by the second event.

Note that we talk about *a* history rather than *the* history. Things can undergo many different sequences of events. Potentially they have many histories. Moreover, how we characterize a history depends upon the set of attributes we use to describe a thing. If we were to choose different attributes to describe the thing, we would end up with a different history of the thing.

The sequence of events obtained by execution of instances of the David Jones *purchase transaction* class constitute part of the histories of the objects associated with the events.

6. Coupling/Interaction

Two things are *coupled* (or interact) when the history of at least one of things depends on the history of the other thing. In other words, the history of the thing is not independent of the history of the other thing. For example, product CA999-200 citric acid of Acme Chemicals Ltd and product Z65123 orangeade of Zeno Soft Drinks Ltd might be coupled because the citric acid is an ingredient of the soft drink. Sales of the soft drink affect sales of the citric acid. An increase in price of the citric acid might prompt Zeno Ltd to change supplier. Product CA999-200's history is not independent of product Z65123's history, and *vice versa*.

The coupling of one thing to another can be viewed as property of the first thing. The property is derived from the history of the thing by a query. How much of CA999-200 citric acid has gone into Z65123 orangeade in the year 2004 is a property of CA999-200 citric acid.

7. System

Systems are things that are made up of other things that satisfy two conditions. First, every thing in the system must be coupled to at least one other thing. Second, it must not be possible to divide the things that make up the system into two subsets such that the history of one subset of things is independent of the other subset of things (in other words, the subsets are not coupled).

The order processing system fragment of Figure 16 is a system in this ontological sense. So also are each of the interoperating purchaser/supplier systems of Figure 17 and also the interoperation as a whole a single system in this sense. Without the transactions, however, the two subsystems become decoupled and not one system in the ontological sense.

8. Composition, Environment, and Structure of a System

The *composition* of a system is the set of things that are in the system. The entities and relationships in the purchaser system of Figure 17 are the composition of that system.

The *environment* of a system is the set of things that are not in the system's composition but interact with (are coupled to) at least one other thing in the system's composition. So the remainder of Figure 17 is the environment of the purchasing system.

The *structure* of a system is the set of internal couplings (between things in the composition of the system) and external couplings (between things in the composition of the system and things in the environment of the system). So the histories of the internal events in the purchasing system are the internal couplings, and the histories of the transactions are the external couplings.

9. Subsystem

A *subsystem* is a system that satisfies three conditions:

Its composition is a subset of another system's (a *supersystem*) composition. In other words, all things in the subsystem are also things in the supersystem.

Its environment is a subset of the environment of the supersystem joined with the difference between the composition of the supersystem and composition of the subsystem. In other words, first we take the things that are in environment of the supersystem. To these we add the things that are in the composition of the supersystem but not in the composition of the subsystem. The things in the environment of the subsystem will be a subset of this newly formed set of things.

Its structure is a subset of the supersystem's structure. In other words, all internal couplings and all external couplings in the subsystem are also internal couplings and external couplings of the supersystem.

Each of the purchaser, supplier and transaction fragments of Figure 17 is a subsystem.

10. Input and Output

The *input* of a thing is the set of state changes (events) that have affected a thing by virtue of the actions of things in its environment. We can conceive of the input of a thing in terms of the history of a thing. Let's assume the x is a thing and y is a thing that is coupled to x. In other words, y is part of x's environment. Let's denote the history of x if it were *not* coupled to y as $h(x)$. An example is the history of *Product* in the supplier system of Figure 17. Thus, $h(x)$ is the set of events that occur in x in the absence of any action by y on x. It is the history of *Product* in the absence of any sales to the organization identified as *purchaser*. Given that x is in fact coupled to y, however, let's denote the history of x given that y acts on x as $h(x|y)$. Thus, $h(x|y)$ is the set of events that occur in x given that y is in fact acting on x. (Events concerning *Product* in *supplier* including those concerning sales to *purchaser*.) Now the difference between the two sets, $h(x|y) - h(x)$, is the input to x by virtue of its coupling to y. (Events concerning *Product* in *supplier* caused by sales to *purchaser*.) This difference is the set of events that have occurred in x that otherwise would not have occurred had it not been coupled to y. When we consider all events in x that have occurred by virtue of the existence of all things in the environment of x, we have the total input to x as a result of it being coupled to things in its environment.

In the same way, we define the *output* of a thing as the set of all events that occur to things in the environment of the thing by virtue of the action of the thing. In other words, if we identify those events that have occurred to things in the environment of a

thing only because they are coupled to the thing, we have the output of the thing. For example, changes in *Product* in the various purchasing systems in the environment of *supplier* are in the output of *supplier*.

11. Type/Class and Subtype/Subclass

A *type* or *class* is a set of things that possess a common property. We have already encountered this concept – the common property is called a rigid property.

Note how types differ from systems. With systems, we focus on couplings between things. With types, we focus on things possessing a common property.

A set of things may be a *subtype* of a type (subclass of a class) if the things possess the property of the type plus at least one other property that is not possessed by all instances of the type. Subtypes may also be disjoint. Again, we have already encountered this concept. A subtype has a rigid property which is anti-essential for its type.

12. Hereditary and Emergent Properties

The *hereditary properties* of a thing are properties that belong also to things in the thing's composition. If we think of the US government as a system, one of its parts is the President, and we think of the government as the George W. Bush administration. Things that the administration does are typically all emergent properties, since they typically depend on several parts. This is of course an example of metonymy. A metonymic name of a whole is a hereditary property.

We can define a subsystem by a selection on some of its parts or on their histories. For example we can define a subsystem of the bookseller exchange where customers and booksellers are both located in the United States. In this case *location* = "United States" is a hereditary property. We can also define a subsystem of customers, booksellers and books where the subject of the book is "Ontology" by a selection on the histories of the books. In this case *subject* = "Ontology" is a hereditary property.

The *emergent properties* of a thing are properties that are *not* properties of any of its components. Emergent properties are always functions of other properties, although often we cannot clearly articulate the nature of the relationship that exists. Simple emergent properties are aggregates (count, sum, max, min). Properties of a system derived from the history of its components are emergent, including in particular their couplings. Total sales of ontology books is an emergent property of the bookseller exchange. So is the percentage of its activity which is among customers and booksellers located in the United States. In the Aircraft example of Chapter 5, an instance of *Aircraft* might be considered to have the emergent property *fully present* if it had all four parts, one in each of the interoperating systems.

A special relationship exists between systems and emergent properties. All systems must have an emergent property of some kind. The only reason for our conceiving a set of things as a system is that the composite thing (system) possesses at least one emergent property that is of interest to us for some purpose.

We are often interested in the emergent properties of *types*, however. Aggregates in particular are often of interest for types. An obvious example is the number of instances of a type.

13. Intrinsic and Mutual Properties

Properties can be classified in still other ways. One is as intrinsic properties and mutual properties. *Intrinsic properties* pertain only to a single thing. *Mutual properties* pertain to two or more things. In the ERA model, these are represented as attributes of entities and relationships respectively. In Figure 17 we might have the intrinsic properties of *total purchases* in the *purchaser* system and *total sales* in the *supplier*

system, together with *total purchases of one purchaser from one supplier* as a mutual property of the two. Couplings between two things are mutual properties.

14. Properties in General and Properties in Particular

Another way of classifying properties is as properties in general and properties in particular. *Properties in general* are properties that belong to all instances of a type. For example, all instances of Part XYZ have common properties like length, width, weight, and colour. The values of these properties could be the same for all instances of Part XYZ–that is, they all have the same length, width, weight, and colour. (Essential properties of the type.) On the other hand, different instances of Part XYZ could have different lengths, widths, weights, and colours. *Properties in particular* are properties of a particular thing. For example, a particular instance of Part XYZ will have a particular length, width, weight, and colour.

15. Part-of Property

A difficult notion to explain precisely is the *part-of* relation between two things[6]. We will be content with the intuitive idea. A thing is called a *composite thing* if it is composed of (made up of) things other than itself (has proper parts). Things in the composite are *part-of* the composite.

We have seen that part-of relations should not be confused with subtype or subclass relations. The latter relations are based on the notion of a subset, whereas the former are not.

7.4 Dolce System

The Dolce system also consists of a number of concepts, which are organized into a hierarchy. Many of the classes in the system are based on the part/whole relationship[7].

0. Entity – the top of the hierarchy. Everything in the system is an entity.
1. Abstract – mathematical entities
   - 1.1. Fact – a logical proposition
   - 1.2. Set – a mathematical set
   - 1.3. Region
     - 1.3.1. Temporal Region – a set of time intervals or points
     - 1.3.2. Spatial Region – a subset of space
2. Endurant – an entity which exists in time. An endurant can have parts, but all of its parts as at any time are present at that time. An endurant can change in time. Institutional facts are endurants. In the David Jones example, the classes *store staff*, *business access point*, *product*, *camera system* and *outfit* are all endurants.
   - 2.1. Object: something in the world. An object exists as a bundle of qualities, which can change over time while the object persists. Aircraft 4501 is an object.

[6] See Mereology under Explanations, Appendix C
[7] See Mereology under Explanations, Appendix C

2.2. Aggregate: An endurant which is not an essential whole. The inventory of parts for the type of aircraft 4501 is an aggregate.

2.3. Feature: A feature is an essential whole which depends for its existence on another object. The human-computer interface or the applications programming interface (API) of a suite of computer software is a feature. A sales pattern of a store chain represented as Euros per store over the last year can have a bump which is a feature (e.g. associated with stores whose clientele tend to participate in a certain religious festival like Easter). Or the market demographics of the chain may have a bump (catering for women aged 18-25, say). A discount store may have an upper limit on the price it feels it can charge, which limits the types of products it can sell. This price limit is a feature.

2.4. Quality: a quality is a value of a property which comes with a particular entity, and persists so long as the entity persists. Having a bookable seat numbered 3A is a quality of an aircraft in an airline booking system. That aircraft 4501 has a bookable seat numbered 3A is a different quality from that aircraft 8220 has a bookable seat numbered 3A, because the two aircraft are different.

2.4.1. Temporal Quality: A quality whose value is a temporal region.

2.4.2. Spatial Quality: A quality whose value is a spatial region.

3. Perdurant/Occurrence: A perdurant or occurrence is an entity which extends in time by accumulating temporal parts. Except for occurrences which exist only at a moment of time, an occurrence is never wholly present. Its past parts are present no longer. The sending of a message is an occurrence, as is that a system is in a state of readiness to receive a message of a certain kind. Making a purchase transaction of Figure 17 is an occurrence, with the temporal parts *RFQ*, *Quote*, etc. In general, the speech act which creates an institutional fact is an occurrence. In the David Jones example, the instances of *purchase transaction* and instances of its part classes are all perdurants.

3.1. Event: An occurrence which is an essential whole. Making a purchase transaction is an event.

3.1.1. Achievement: An event which is atomic. Sending an RFQ is an achievement.

3.1.2. Accomplishment: An event which is not atomic. Making a purchase transaction is an accomplishment.

3.2. Stative: An occurrence which is not an essential whole. Being willing to receive a *Quote* in respect of an *RFQ* is a stative.

3.3. State: A stative all of whose temporal parts are of the same type as the stative itself. Being willing to receive a *Quote* in respect of an *RFQ* is a state.

3.3.1. Process: A stative all of whose temporal parts are not of the same type. A business trading is a process, whose temporal parts are transactions of various kinds.

7.4.1 Endurants and Perdurants

The distinction between endurants and perdurants is central to the Dolce formal ontology and also (under different names) to the BWW system. These terms will be clearer with some discussion.

Endurants are entities that *exist* in time. If they have parts, all of their parts exist at the same time. Ordinary objects are endurants, as are records such as are stored in information systems. In the BWW system an endurant is called a *thing*.

Perdurants are entities that *happen* in time. A perdurant can happen at a point in time, or can have temporal parts. There are two subclasses of perdurant: event and stative. An *event* is definite. Even if it is complex, it is capable of completion. If it has temporal parts, none of them are of the same kind as the whole. A *stative* is indefinite. It can have temporal parts which are the same kind as the whole. We might think of an event as happening, while a stative is going on.

Examples of events are falling over, blowing up, coming into and going out of existence, including being born and dying. Almost all speech acts are events: buying, selling, being inaugurated President of the USA, getting married, getting divorced, being given a name, winning or losing a contest, being hired or fired, enrolling in a university program, being awarded grades, earning a degree, graduating. Some of these have temporal parts. In particular, earning a degree has parts including enrolling, being awarded grades and graduating, none of which are earning a degree.

Examples of statives include raining, sitting, being alive, being dead, being ready, running, working, cleaning – all are states (cleaning indefinitely is a state, but cleaning the kitchen floor is a process, since its temporal parts are things like cleaning in front of the stove, cleaning under the table, and so on, which are not cleaning the kitchen floor).

The BWW system includes events but not statives.

An endurant is created by an event. Its existence is a sort of memory of the happening of the event. Any change in an endurant is created by an event. An endurant is destroyed by an event. Unless the event destroying an endurant also creates another endurant (perhaps of a different kind, a log file transaction say), there is no longer any memory of the event.

A stative is started and stopped by events. The rain starts and stops. Inauguration of a President of the USA begins his presidency while inauguration of the next president stops his presidency. Sending a request for quotation message starts the sending site's being receptive to a quotation, which is stopped by the receipt of the quotation or by the expiration of a deadline. Enrolling in a degree program starts the studying for the degree, which is finished by the degree being awarded.

The BWW concept of *state* is the unchanging representation of an endurant between events. It is parallel to the Dolce stative, but does not have an explicit behavioural dimension.

Since the creation, destruction and any change to an endurant is performed by an event, the BWW concept of *history* of an endurant is a perdurant. Any temporal part of the history of an endurant is a history of the endurant, so the history of an endurant is a stative.

But a record is an endurant. So the record of the history of an endurant is an endurant. As is the record of any perdurant.

A perdurant may have an extension in time, but in its interaction with the universe at any point in time can only be determined by its physical state at that point in time. Historical causality is not admissible. Neither can the future have any causal effect. So

associated with any perdurant is a collection of endurants. The record of a perdurant is identical with the history of its associated endurants.

7.5 Comparison of BWW and Dolce Ontologies

The BWW and Dolce ontologies although developed independently are compatible with each other. This is perhaps not surprising, since they both are based on Aristotle's ontology. They differ by emphasizing different aspects.

A *thing* in BWW is equivalent to an *entity* in Dolce. BWW's *properties* and *attributes* are Dolce's *qualities* and *properties*, respectively. BWW's *state of a thing* in Dolce is the representation of an entity as a bundle of qualities. An *event* in BWW is an *event* in Dolce. The more general class *occurrence* with its other subclass *stative* in Dolce do not feature as such in BWW. However, BWW's *history of a thing* is a particular kind of *process* in Dolce. The derivative concepts *coupling*, *input* and *output* are *properties* of *history* in BWW, so qualities in Dolce.

A *system* in BWW is a kind of *endurant* in Dolce which uses the concept of *coupling*. The *part* concepts in BWW and Dolce are the same. *Composition*, *environment* and *structure* are all collection of *parts* of a *system*, as is *subsystem*. The *type*/*subtype* relationship is the same in both ontological systems. The classifications of properties into *hereditary*/*emergent* and *intrinsic*/*mutual* apply to Dolce's *qualities*. *Properties in general* in BWW are *essential qualities* in Dolce, while *properties in particular* are *anti-essential qualities*.

The *abstract* subtype of *entity* in Dolce is not explicit in BWW. Neither is the *endurant*/*perdurant* distinction. However, BWW's *history of a thing* captures the relationship between *endurants* and *perdurants*. *Endurants* participate in *perdurants* and *endurants* have histories which consist of *perdurants*. The mereological subclasses of *endurant* do not feature in BWW. However, in all cases the features of one system missing in the other can be constructed using the features of the other.

In particular, consider the Dolce concept *stative*, for example a dog's being angry. A dog's being angry is a Thing since everything is a Thing, but it is not a BWW event, at least not on the scale of someone standing terrified in front of a large angry dog. The way a dog's being angry fits into the BWW system is via the BWW concept *state of a thing*, which is a collection of BWW properties. After all, the only way we can know about a thing is by its properties. One property of an angry dog is that it is growling. So we can think of a dog as an endurant which has behavioural propensities angry, friendly, eating, sleeping, and so on. Each of these behavioural propensities is a stative when it is ongoing. But each of these statives is indicated by BWW properties (Dolce qualities) of the endurant: growling, tail wagging, chewing, being quiet with eyes closed, and so on. So a Dolce stative is a subclass of BWW Thing not specifically recognized in the BWW system. Associated with each class of stative is a rigid property corresponding to the BWW state of a thing, through which we can identify the ongoing behaviour which is the stative.

Moving into our preferred institutional fact environment, a store's being open for business is a stative, represented in BWW by the store's (a Thing) having an "Open" sign on its front door (a state of the Thing). In the EDI example, a purchaser's being receptive to a quote is a stative, represented in BWW by the record of an RFQ having been issued (a Thing) and two properties of the RFQ, time since issue (*tsi*) and time the RFQ expires (*te*). If $te > tsi$ then the stative is ongoing, otherwise not.

7.6 Benefits of Using a Formal Upper Ontology

The examples of interoperating systems of earlier Chapters have been developed first without benefit of any of the formal material of the last two Chapters, and then in Chapter 6 using only some of the formal metaproperties from the OntoClean system. The formal ontologies BWW and Dolce are another entire level of concepts. What can be gained from their use?

There are two kinds of benefits from using these formal upper ontologies. First, they provide a rich vocabulary for describing the systems we are building and working with, so they can assist us in their design and in understanding them. In particular they can make much richer the application domain specific material ontologies we build. Second, they form the basis of a suite of abstract data types which can be provided as libraries to the designers of agents which will automatically interoperate with our systems and with each other. We will consider these in turn.

7.6.1 Providing a Rich Vocabulary

In this text we have pursued an extensive electronic commerce example, which has been presented in two different ways. In Figure 16, we have shown the structure of the individual systems and in Figure 17 their interoperation. Using the formal upper ontologies, we can look at this example in a different, and perhaps surprising way. In the following, we will use the Dolce system, augmented with elements of BWW.

Figure 16 shows an articulation of an order entry system into parts, then an articulation of some of its parts into further parts. The parts *Customer* and *Product* are endurants, subclass object, while the parts *Purchase Transaction* and *Activity* can be viewed either as perdurants, subclass event, or as endurants, the records of the events having taken place.

Now a perdurant or occurrence exists in time, and can have temporal parts. Once an occurrence or part of an occurrence has passed, it can only exist in the present as a memory of some kind. The memory of the occurrence of an event is either an object (a record) or a stative, subclass state which is a propensity for behaviour of some kind.

In this case the memory is both object and state. The (ERA) entities in the model become implemented as places to store records of the events. The records of the various classes of event in Figure 17 all are the domains of properties representing dependencies on the existence of records of earlier events. For example, the property whose domain is *Quote* and whose range is *RFQ* links an instance of *Quote* to an instance of *RFQ*. The property is mandatory, so that the instance of *RFQ* must have existed prior to the instance of *Quote* coming into existence. The temporal parts of the Purchase event are represented as increasingly complex collections of linked records.

These dependencies can be used as BWW states to tell us when corresponding Dolce states are ongoing. These Dolce states are when the system is receptive to certain events. A delivery event is only recognized by the system if an order event has previously occurred, for example.

This representation leads us to think about the system in a state-transition model, as in Figure 21. Before an RFQ event occurs (an RFQ is issued), the system is in a *start* state. Issuing an RFQ initiates a set of message exchanges in which at first the issuer is receptive to receipt of a Quote message. Receipt of a Quote message marks a new state, called in Figure 21 *RFQ+Quote*. In this state, the issuer of a Quote is receptive to a Purchase Order message. But this receptivity has a limited lifetime. If this time expires,

the receptivity stops. The system goes into a new state, *Failed Quote*, in which no further messages are accepted. The process continues through the successive states *+Purchase Order*, *+Delivery*, *+Acknowledgement*, *+Invoice*, *+Payment* to *Completed*.

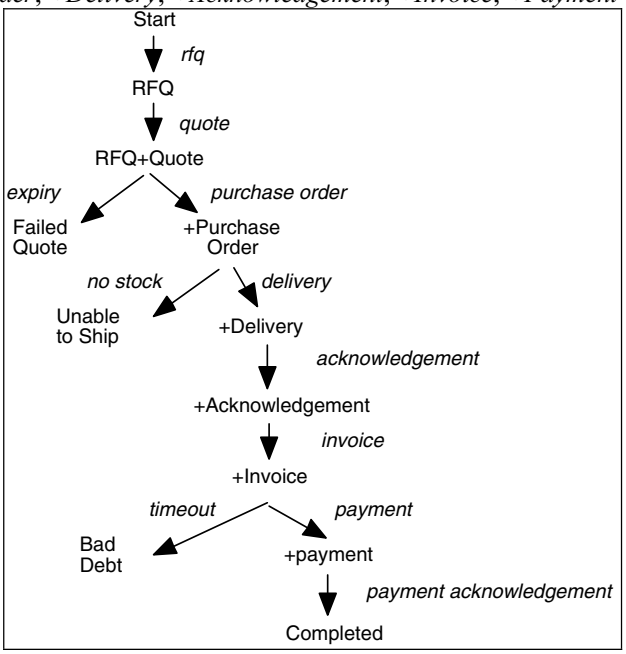


Figure 21. A stative representation of the EDI system of Figure 17

As with *RFQ+Quote*, the stative view of the system suggests that other events can occur in subsequent states, leading to states off the main sequence like *Failed Quote*. Two of these are shown in Figure 21. In the state *+Purchase Order*, it is possible that stock may not be available, leading to the state *Unable to Ship*. In this state the purchase order must be allowed to lapse. In the state *+Invoice*, it sometimes happens that a payment event does not occur within a reasonable time, so there is a transition to a new state *Bad Debt*.

Each of these Dolce states is marked in BWW fashion by the records of the occurrence of the events.

So we can see that the formal ontology improves our modeling by providing a language to describe the system at several levels of generality, with each level articulated to the next using well-defined constructors (part, instance, subtype). The semantic distinctions between endurant and perdurant and their subclasses increases our understanding of the system and its parts, in this case leading to a perdurant view of the application as a sequence of statives connected by events as well as a system of endurant records connected by properties representing existence dependencies.

The usefulness of formal ontologies in development of material ontologies for interoperation in specific application domains can be seen from the example of the Gateway for Educational Materials (GEM)[8], developed by a consortium of web-based education providers supported by the US Department of Education. The GEM ontology

[8] http://www.geminfo.org/index.html

is published as a set of attribute names shown in Figure 22, most of which have a set of permitted values. (The version shown is as published in August, 2002.)

Title	GEMSubject	Other Subject
Keywords	Description	Coverage
Creator	Quality Indicators	Standards
Resource Type	Grade Levels	Essential Resources
Duration	Audience	Pedagogy
Date	Identifier	Rights Management
Relation	Format	Cataloging Agency
Online Provider	Publisher	Language

Figure 22. The GEM ontology

If we look at Figure 22 from the perspective of the Dolce system, the first thing we notice is that there are properties but no object. In GEM, the properties inhere to a *Resource*, which Dolce leads us to explicitly show. Now, the property *Resource Type* (Figure 23) becomes relevant as a list of subtypes of *Resource*, giving a primitive type system with one level. The *Subject* property is organized into a type-subtype system of two levels (e.g. *Film* is a subclass of *Arts*). Using a formal ontology leads us to make implicit structure explicit.

Pursuing implicit structure, we now look at a selection of resource types shown in Figure 23. Notice that far from being independent types, the selected types in fact form a single complex structure under the *part-of* relationship. *Course* has parts *Instructional Units*, which have parts *Lesson Plans* which have parts *Activities*. An *Educator's guide* can be part of either a *Lesson Plan* or an *Instructional Unit*. Without the part-of form in the formal ontology, we could not express this structure.

Further and richer structure comes from the attribute *Relation Type*, a sample of the terms in which is in Figure 24. These relation types are all relationships between resources, which can be represented in the Dolce and BWW ontologies using various forms. For example,

hasBibliographicInfoIn is a dependency relationship;

isRevisionHistoryFor connects a whole to a part, which is a BWW History of a Thing;

isCriticalReviewOf can be represented as a causal relation (the resource causes the review resource);

isContentRatingFor can probably be seen as a Dolce Property.

(Dublin Core[9] is a standard of metadata for web data items, consisting of ten properties including *Name*, *Language*, *Version* and *DataType*. See Chapter 10.)

[9] http://dublincore.org/

Course A sequence of instructional units, often a semester long, designed by a teacher (or a faculty or other group of teachers) to advance significantly student skills, knowledge, and habits of mind significantly in a particular discipline and to help students meet specified requirements (as set forth in curricula or district or state policy).
Unit of instruction A sequence of lesson plans designed to teach a set of skills, knowledge, and habits of mind.
Lesson Plan A plan for helping students learn a particular set of skills, knowledge, or habits of mind. Often includes student activities as well as teaching ideas, instructional materials, and other resources. Is shorter (in duration) than, and often part of, a unit of instruction. Goals and outcomes are focused.
Activity A task or exercise that students are asked to do–often as part of a lesson plan or other larger unit of instruction–to help them develop particular skills, knowledge, or habits of mind. Usually, the goals and outcomes are broad.
Educator's guide A guide intended for use by educator's as a supplement to a lesson or unit plan.

Figure 23. Selection of GEM resource types

hasBibliographicInfoIn The resource being pointed to by this relation provides bibliographic information for the main resource.
isRevisionHistoryFor The resource being pointed to by this relation provides a history of revisions made to the main resource which the Dublin Core metadata was created for.
isCriticalReviewOf The resource being pointed to by this relation provides a review of the resource being described by the Dublin Core metadata.
isOverviewOf The resource being described by the Dublin Core metadata is overviewed in the resource pointed to by this relation.
isContentRatingFor The resource pointed to by this relation is a set of content ratings for the resource being described by the Dublin Core metadata.
isDataFor Provides dataset or programs for another resource.
isSourceOf The resource being pointed to by this relation is the published source that the resource being cataloged is extracted or excerpted from.

Figure 24. Selection of terms from *Relation Type*

Having used the formal ontology to explicitly express the implicit structure of the GEM ontology, we can now turn to the use of the formal ontology to suggest some things missing from the GEM system. Nearly all the elements of GEM are represented as endurants, either substantives, qualities or properties. We know from Dolce that endurants are associated with perdurants (occurrences), at least in their creation and destruction. This suggests we need standard messages for creation and destruction of all the endurant objects. Most of them need as well messages for updates. All of these are *events*. If we have events, then we also have the subclass of *process* which is the BWW *History of a Thing*.

Now that we are thinking about *occurrences*, it becomes apparent that we need additional events. At least we need to be able to query the objects represented in the database. Here we can import and perhaps adapt the Z39.50 information retrieval standard messages with their prescribed behaviour. And so on.

Finally, the Dolce system is built upon metadata. So every part-whole structure needs to have a unifying relation and an identifying relation. Every type has essential and rigid properties, and also an anti-essential property used to identify instances (a key). Some of the metadata consists of predicates, so we need to know whether for example a whole is an essential whole, or a dependency is essential.

We see that having a rich knowledge representation language such as Dolce leads us not only to the representation of complex structure, but to the representation of the dynamics of the application, which will be necessary for interoperability.

7.6.2 Abstract Data Types

Data structures generally have collection of operations or functions associated with them. An *abstract data type* is a data structure together with its collection of associated operations and functions. This is a familiar concept:

- the integers support equality together with arithmetical operations including division with remainder and binary relational predicates;
- reals, in their computer incarnation as floating point numbers, support unqualified division (except by 0) but not equality;
- sets support union, intersection, difference, membership test, add and delete member, and so on;
- the subtype relation supports navigation between subtype and supertype, including a transitive closure operation;
- the part-whole relationship supports the mereological operations and predicates[10], including mereological sum and transitive closure of parthood.

So if an agent program is written in a programming language supporting these abstract data types, the programmer has access to a library of functions and operations which not only simplifies programming but also standardizes the complex data structures and operations used among the community of people subscribing to those abstract data types. The temptation for an individual programmer writing a program for a specific problem is to develop idiosyncratic data structures with idiosyncratic and incomplete collections of functions and operations. Development of agents interoperating in an open world is much easier if standard abstract data types are employed. These functions and operations can be provided as aspects of the query language for an information source as well as a software library.

Some of the functions and operations stemming from the formal upper ontologies of this Chapter include:

- Every perdurant/ occurrence has participating endurants
- Every event has an associated time
- An occurrence persists through a record, which is an object
- Every endurant has an associated history, which is a sequence of objects which are records of events
- Accomplishments have participating endurants which also participate in statives which are induced by the occurrence of parts of the accomplishment (as we have seen in the previous section).
- Every entity has its associated metadata, so needs methods to present its metadata. This metadata can consist of formulas (say unifying or identifying relations).

If the application-dependent material ontology is built using a formal ontology as a knowledge representation system, then the supporting ontology server can provide methods for all these rich abstract data types, thereby improving the richness and reducing the cost of building the material ontology.

[10] See Mereology in Explanations, Appendix C

7.7 Application to the Examples

The formal upper ontologies can enhance our understanding of the examples in Chapter 4.

Z39.50 with the world of Figure 4 and the facilities taken together has both endurants (the world) and perdurants (the facilities), so has the machinery to both create and use the world. But the ontology as shown does not have provision for histories, which in this case would be audit trails or access logs.

Tic-Tac-Toe similarly has both endurants and perdurants (*PlayerRole* and *Action*), but no provision for history. The subclasses of *State* could be interpreted as perdurants, specifically statives, since they determine whether the game can proceed or not.

The remaining examples, SIC, SNOMED, the Periodic Table and dimensions are all endurants. There is no provision for change of any kind. But they are all very much examples of systems with subsystems. as we have seen in the application of the part-whole relationship in the previous two Chapters.

7.8 Discussion

With the introduction of formal upper ontologies, we now have a much richer representation language for ontologies, and a rough specification of some abstract data types which would be implemented in ontology development tools and servers.

7.9 Key Concepts

BWW and **Dolce** are **formal upper ontologies**. Formal structures include **systems**, **endurants**, **perdurants**, **subclasses**, etc. Formal upper ontologies provide a rich meta vocabulary to help develop ontologies and suggest **abstract data types** which can be supported by ontology servers to make it easier to build rich ontologies.

7.10 Exercise

1. (relevant to point 9 of the major exercise) Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a. Describe examples of systems, subsystems, and environments of systems in the ontology. Find some coupled objects in the ontology. Identify hereditary and emergent properties of the systems. Make plausible additions to the system if necessary.

b. Describe endurants and perdurants in the ontology. What endurants participate in the perdurants? How are the histories of the endurants represented in the ontology?

c. Think of an improvement in the ontology suggested by some aspect of the BWW/Dolce formal upper ontologies. Show how the improvement follows from the upper ontologies.

7.11 Further Reading

The BWW system is explained in [11], especially chapter 2. The Dolce system is presented in [12]. Dewey [13], especially Chapter 2 "Existence", is relevant to the distinction between endurant and perdurant, and why the distinction is not absolute.

8 Quality

There is more than one way to create an ontology for a given application. Ontology is therefore a design problem. How to choose from among design alternatives leads to the question of how we assess the quality of a design.

8.1 Quality of Ontologies

Quality is a judgment of an object against an intended use. In order to evaluate the quality of a particular ontology we need to test alternatives against the requirements of the application as seen by the various stakeholders. A very general framework for doing this is to assess the ontology against three linguistic dimensions: syntactic, semantic and pragmatic. These three dimensions answer respectively the questions:

- Is the ontology syntactically correct?
- Does the ontology cover the domain of interest?
- Is the ontology comprehensible by the user?

The *syntactic quality* dimension is on its face straightforward. If the ontology contains syntactic errors, then the software tool supporting the language used in model should be able to point these errors out to the analyst. However, a deeper issue is how complex a syntax should be used. There are, after all, many modeling languages. In tool-supported ontologies, the syntactic issue is richness and complexity of syntax rather than correctness.

Semantic quality is how well the ontology reflects its universe of discourse, while *pragmatic quality* is how easy the ontology is to understand. Assessment along these dimensions is in practice an assessment of fitness for purpose. The closer an ontology reflects its universe of discourse, the larger and more complex it will tend to be and therefore the more difficult to understand. There are comparable tradeoffs between syntax and semantics, and between syntax and pragmatics.

8.2 Gruber's Design Principles

A number of people have considered the problem of how to achieve high quality ontologies along the semantic dimension. The way this is normally done in engineering is to develop a set of design principles which encapsulate the tradeoffs necessary to achieve reasonable quality at a reasonable cost. One of the best worked-out sets of principles was published in the early 1990s by Thomas Gruber.

Gruber is famous for a very widely cited definition of ontology

An ontology is an explicit specification of a conceptualization

The conceptualization is of the world we wish to share, and is the source of the semantics. An ontology is a representation of a specification of that world. We have seen in Chapter 1 that an ontology is a kind of conceptual data model. We know from conceptual modeling that there are many different modeling systems, and that a given application can be modeled in different ways even within a single system. A

representation of a specification of a conceptualization combines syntactic and semantic issues. But if the human user can't understand the ontology, it is probably wrong. This is why there is a need for design principles.

Gruber is also famous for the terminology of committing to an ontology. We say that an agent *commits* to an ontology if its observable actions are consistent with the definitions in the ontology. In a system of interoperating autonomous agents, we of course can not know what is going on within any of the agents. We can know only their observable behaviour. So if an agent behaves according to our understanding of the ontology, then so far as we know all is well. So pragmatically, a common ontology defines the vocabulary with which queries and assertions are exchanged among agents.

Gruber formulated ontology design principles as five design criteria:

- **Clarity** - effectively communicates intended meaning
- **Coherence** - sanctions consistent inferences
- **Extendibility** - anticipates uses of shared vocabulary
- **Minimal encoding bias** - as free as possible from implementation decisions
- **Minimal ontological commitment** - as few claims as possible about the world being modeled.

We will discuss each of these in turn, then look at how they apply to some examples.

8.2.1 Clarity

> 'There's glory for you', [said Humpty]. `I don't know what you mean by "glory",' Alice said. Humpty Dumpty smiled contemptuously. `Of course you don't -- till I tell you. I meant "there's a nice knock-down argument for you!"' `But "glory" doesn't mean "a nice knock-down argument",' Alice objected. `When *I* use a word,' Humpty Dumpty said, in rather a scornful tone, `it means just what I choose it to mean -- neither more nor less.'
>
> *Alice Through the Looking Glass*, Chapter VI

A perfect example of semantic heterogeneity. If programs are to interoperate, they must all give up the power to choose by themselves what the words mean and commit to the ontology.

At bottom, an ontology is a list of words. The Bib-1 attribute set in Chapter 4 is an example of an ontology consisting of nothing more than words. If we want to effectively communicate intended meaning, we need to have some way to prevent unintended interpretations. This sort of problem arises for example in filling out a tax return. The ontology provided by the Taxation Office has a class "total taxable income". It is a considerable advantage to a taxpayer for that class to have the value 0, at least in that the taxpayer won't have to pay any tax. For most taxpayers, though, the Taxation office would consider a taxable income of 0 an unintended interpretation. Perhaps an honest mistake, perhaps fraud, but definitely to be discouraged.

The Taxation office expends an enormous effort to prevent unintended interpretations. It publishes detailed regulations specifying what records must be kept and what the various types of records mean. It has the authority to send tax auditors to taxpayers who examine the records to make sure the various types are complete and commit to the Taxation Office's ontology. It can punish fraudulent interpretations by fines and jail sentences.

But even with all that effort, some taxpayers manage to get unintended interpretations to slip through. So it is practically impossible to completely exclude unintended interpretations. There is even a theorem in mathematics (called the Skolem-Lowenheim theorem) which says that for theories whose intended interpretation is uncountable there always exists an unintended countable interpretation.

We can help avoid honest mistakes. It is very common to separate words into different categories and to stipulate that some words can occur only together with others. The Tic-Tac-Toe example in Chapter 4, for example, restricts the word *atPl* to occur only in the context of a *Cell* which a *Game has*, and the words "corner", "centre" or "side". Further restrictions are possible. For example, in the conceptualization of Tic-Tac-Toe, a cell is at exactly one place. So the specification can put cardinality restrictions on the *atPl* property so that every instance of *Cell* is associated with exactly one instance of *Places* by the property *atPl*. Conceptual modeling languages and ontology representation languages typically permit the specification of a wide range of restrictions.

Some ontology representation languages permit restrictions or definitions to be expressed in logic, generally some variant of the first order predicate calculus. For example, in Tic-Tac-Toe, an instance of *Game hasState Blocked* if all *Rows* the *State* is *evidencedBy* are in the subclass *Mixed*. The predicate calculus introduces some additional words (the connectors, quantifiers and so on, called syncategorematic terms) which have very widely accepted and very precise interpretations. (This definition and the others in the Tic-Tac-Toe ontology are expressed in Common Logic in Chapter 13.) So ontologies expressed using formal languages can be clearer than those expressed using less formal languages.

There are a number of proposed standard rule languages intended to allow expression of rules of different kinds, including RuleML[11] and the Object Management Group Business Semantics for Business Rules Request for Proposal[12].

A particular application of rules is to provide definitions for subclasses. A defined subclass is higher in clarity than a declared subclass. The definition can take an instance of the superclass and assign it to a subclass, while with a declared subclass, each individual must be assigned to the subclass by a process outside the ontology. Partially defined subclasses such as described in Chapter 6 are intermediate in clarity.

Generally speaking, the richer and more formal the restrictions placed on the use of the words in the ontology, the less likely there is an unintended interpretation caused by an honest mistake. But of course the richer and more complex the representation the harder it is to understand. To build ontologies high in clarity we need software tools.

8.2.2 Coherence

If an ontology is going to sanction consistent inferences, it must sanction inferences at all. An ontology is a specification of a conceptualization, so we can distinguish inferences on the specification from inferences on the conceptualization. When we are talking about inferences on the conceptualization, we are talking about real-world inferencing using the human powers of reasoning with whatever tools of logic and mathematics are available. When we are talking about inferences on the specification, we are talking about computer-based inferencing tools.

---

[11] http://www.ruleml.org/
[12] http://omg.org

Consider an ontology supporting on-line forms submission to departments of the government of a certain country. The conceptualization is the whole system of speech acts and institutional facts constituting the regulations administered by the departments. The specification can be more or less clear. One feasible specification is as a simple list of terms which function as class names, and are used as field labels on forms. Another feasible specification is as a complete theory of all the institutional facts expressed in the predicate calculus.

It sometimes happens that the system of regulations has a Catch-22, for example in order for an immigrant to get a residency permit they must have a job, but it turns out that for some classes of immigrant one must have a residency permit before it is legal for someone to employ them. This situation fails to support consistent inference, in that the regulations are not satisfiable.

However, the ontology as designed is the specification, not the conceptualization. The specification consisting entirely of class names does not support any inferences. An immigrant could use the forms to submit an application, and the system implementing the ontology could send it off to the department, only to have it rejected. The representation is coherent, if only vacuously, even though the conceptualization is not. On the other hand, the specification consisting of a complete theory in the predicate calculus could discover the Catch-22 by an automated reasoning process, so would be incoherent as well as the conceptualization.

We can't be concerned with how consistent the world is, but only about the reasoning supported by the representations we have of the specification of the conceptualization. So only the richer representations, higher in clarity because they exclude more unintended interpretations, have the possibility of lack of coherence.

Failure to sanction consistent inferences covers a number of more specific types of reasoning failure. Besides the circular reasoning discussed above, there is inconsistency: a person does and does not have a valid residency permit; and failure to sanction simple obvious conclusions. Suppose a system were able to conclude that a person had been a resident for one year, and also to conclude that they needed to be a resident for six months to be eligible for a benefit, but could not conclude that one year was a longer period of time than six months. That situation would be considered lack of coherence. But the specification which does not have any reasoning capability can not be accused of lack of coherence. (The concept of coherence is itself somewhat low on clarity.)

8.2.3 Extendibility

In order to anticipate uses of shared vocabulary we have to anticipate the future. This is of course not possible. But a good formulation of the problem of future-proofing data models was given by John Debenham in the late 1980s.

Debenham's work includes a large number of specific design principles, but they are all expressions of the fundamental principle of avoiding redundancy – each object in the world being shared (that is, the conceptualization) is represented in the specification of the conceptualization in one place and one place only. This is an extension of the database design principle of normalization, where the schemas are designed to store various kinds of dependencies non-redundantly. The reason for normalization is that it avoids update anomalies – a change in a dependency can be recorded by a single update without introducing inconsistencies. The reason given for Debenham's design principles is maintainability – a change in an object in the

conceptualization can be represented in the ontology by one possibly complex change in the specification.

For example, universities have degree programs, each of which has a set of rules. There will often be aspects of rules which apply to more general groups of students – say undergraduates versus postgraduates, coursework versus research degrees, degrees offered by different Faculties. If the common rule sets applying to each group of students is factored out of the rule definition, then a general change to say a research degree type of program will automatically propagate to each specific research degree.

The price paid for normalization is increased complexity of structure (more tables) and increased complexity of processing (more joins). Applications where performance is a quality issue often denormalize the most-used tables. Some database systems support view materialization, where a denormalized table is represented as a view whose population is computed as soon as the data is available, and where the redundant updates are performed automatically. Similarly, application of Debenham's design principles tends to introduce additional complexity both of structure and processing. In the university rule example, the rules expressed in this way will be very difficult for a student to understand since the rules relevant to a particular student will be scattered through a number of modules. Quality will be enhanced if the student-rule interface is able to automatically incorporate all the modules into a single relevant story.

So a high-quality ontology will be designed in the awareness of redundancy, and where redundancy needs to be introduced into the design, it will be properly controlled. The complexity introduced will be compensated.

Faceted ontologies like SNOMED are more easily extended than are single hierarchical systems like the SIC. If a new drug or class of drugs becomes available a change needs to be made only to the *Chemicals and Drugs* facet, leaving the other ten facets unchanged. A major change is underway in the SIC at the time of writing, which is a massive exercise. (The new system is called North American Industry Classification System (NAICS)[13].)

Sometimes redundancy is not actual, but potential. For example, consider a primitive form of the Z39.50 Bib-1 use attributes of Chapter 4 which includes the term 'LC call number' (LC = U.S. Library of Congress). This is an atomic specification of a potentially complex concept. There is the possibility that there will be other types of call number. In fact, in Bib-1 there are several different call numbers. Besides LC, there are NLM (National Library of Medicine) and NAL (National Agricultural Library), among others. So even though there may be only 'LC call number', it is better to separate the two elements of the conceptualization into two elements of the specification.

Debenham's rule paraphrased for ontologies "each conceptual object represented in one specification object and one specification object only" is a good measure of extendibility.

But whether extendibility is a problem depends on what use is made of the ontology. A simple term list used as a set of field labels on forms, where the documentation of the fields resides in the user community, can be extended simply by adding more terms. If the applications using the terms do not attempt to generalize, then for example the different kinds of call numbers in the Bib-1 use attributes are not a problem. Only if the conceptualization of the world includes the concept "call number"

[13] http://www.bls.gov/bls/naics.htm

with qualifications does the potentially redundant representation in Bib-1 become a problem. Extendibility becomes more of a concern as clarity increases.

8.2.4 Encoding Bias

Ultimately any working computer system must be implemented. An ontology, though, is a long way from an implementation. If the ontology is intended to facilitate interoperation, like the Z39.50 or Tic-Tac-Toe examples from Chapter 4, the inner workings of the agents doing the interoperation are hidden. There are typically very many different implementations capable of generating the correct actions in the correct circumstances so that an agent can be said to commit to the ontology. If the ontology is intended as a standard component of the design of a class of systems, like the Periodic Table or Dimensions examples, there are typically very many different implementations of the design. Good software engineering practice calls for implementation decisions to be made as late in the software development process as is practical, since a change at the design level is much cheaper to make if it doesn't require re-doing a significant amount of implementation.

Encoding bias is the unnecessary incorporation of implementation decisions in the ontology.

What counts as encoding bias depends on what the ontology is used for. If the ontology is used as a component in a software design, it is further from an implementation than if it supports ongoing interoperations among agents. Consider the dimension example. If the dimension ontology is used as a component in a system design, the relevant terms are the dimensions such as distance, area, volume, time, speed, etc.; the formulas for computing one dimension from others; and the basic geometry. We know that if we multiply distance by distance we get area, and if the geometrical figure is a circle that its area is the product of the square of the radius (distance) and the constant pi. The same dimensions and geometry apply whether we are looking at a subatomic scale where distance is measured in Angsrom units or at a galactic scale where distance is measured in parsecs. Within the same dimension and unit of measure, the precision of the representation required differs from application to application. An application supporting carpenters building houses needs metres to be accurate to a few millimetres, while an application involving precision machinery needs metres to be accurate to a few micrometers. Similarly for the number of decimals used to represent pi.

So an ontology offering basic dimensional analysis and geometry has least encoding bias, one including a choice of units more encoding bias, and one prescribing the precision of numbers even more. The most encoding bias is if we not only specify the precision of numbers but their computer representation, such as IEEE double float. An ontology like the SIC, SNOMED or the Periodic Table can have encoding bias if it specifies whether the terms are represented as ASCII or Unicode.

On the other hand, if the ontology supports interoperation, then the different agents need to know not only the dimensions, but what units of measure to use and how many digits of precision are required. So there is more encoding, but it is not bias. But specifying the internal number representation as IEEE double float is still encoding bias, since that imposes an unnecessary constraint on the implementation of the agent. The other agents don't need to know how the messages they receive are constructed, only what they look like. In the case of text fields, there is more encoding bias if fixed-

length fields are specified in the ontology, less if the fields are allowed to be variable length, represented perhaps as marked-up text using an XML-specified markup.

How complex data structures are represented is a significant source of encoding bias. If an object is a whole consisting of a number of parts or as a set together with its members, or as a sequence of more primitive objects, one way to send the structure to a player in the exchange implementing the ontology is as an array. (SQL:1999 provides an array construct but not a set construct, for example.) An array is one way to implement these semantically richer structures, but not the only way. Less encoding bias is achieved if the complex structures are sent using for example XML serializations. So that an instance which is a set would be designated by begin and end "set" markup elements, and each member of the set by begin and end "member" markup elements. UML for example has an XML serialization called XMI (XML Metadata Interchange).

The abstract data types of the formal upper ontologies of Chapter 7 can be the basis for systems of XML markups, so that the complex data structures can be specified without regard for how a particular system may implement them. This is one way formal upper ontologies can contribute to construction of higher quality ontologies.

It might be thought that a simple taxonomy (low in clarity) would be low in encoding bias also. After all, a list of names represented as variable length strings with unspecified encoding is about as little implementation specification as possible. However, there are often semantic relationships among the terms that are in the conceptualization but not represented in the specification. For example, the SIC is simply a list of names of industry classifications. However, the way an organizational subunit is classified into a particular SIC industry is part of the conceptualization of the SIC. But it is represented only in the regulations and bureaucratic practices of the US Department of Labor, not in the ontology. Therefore, for another body to adopt the SIC and to independently classify organizational subunits they would need to duplicate these regulations and bureaucratic practices, an enormous encoding bias. So underspecification of the conceptualization constitutes encoding bias, since for the conceptualization to work, it must be implemented somehow.

8.2.5 Ontological Commitment

Encoding bias relates to how much the specification of the conceptualization over-constrains implementation. Ontological commitment relates to the conceptualization rather than the specification.

The conceptualization of a world whose specification is to be represented in an ontology is related to the application which is going to use that world. But an ontology takes resources to develop, so it is frequently attractive to be able to re-use an ontology in a different application from that for which it was developed. Clearly the world of the second application must be in some degree similar to the world of the first. It doesn't make sense to try to re-use the Periodic Table ontology for an application supporting chess tournaments. So our proposed re-use must be plausible.

When we are making a plausible reuse of an ontology, *ontological commitment* is the extent to which we must modify the world of the present application in order to commit to the ontology. For example, it is well known that organizations adopting enterprise computing packages like SAP, Peoplesoft or Oracle Financials must make substantial changes in their business practices to make good use of the software. The

author's University a few years ago replaced its student records system with a very good package from Peoplesoft, but had to make extensive changes not only in terminology (what were "subjects" became "courses") but also in degree rules. The Peoplesoft package had a fairly high level of ontological commitment, even though it has considerable flexibility.

Our discussion of semantic heterogeneity in Chapter 3 leads us to expect that any ontology supporting a conceptualization of a world of speech acts and institutional facts would have a high degree of ontological commitment if used outside the scope of the governing institution. The author's University has an ontology used for student enrolments and grades, delivered via the student records system. Each of the thousands of lecturers is able to represent the students enrolled in their courses and the grades to be assigned using that ontology. But courses differ greatly in their assessment practices. Most lecturers use computer-based systems to keep track of assignment of students to groups, who has completed what assessment item with what result, late submissions with or without permission, and so on. These more specific systems can operate essentially as specializations of the student records ontology.

Since the lecturers are already committed to working within the University's enrolment and grade structure, the level of ontological commitment for the student records ontology is low. But one University's student records ontology would have a much higher degree of ontological commitment if used by lecturers at another university, especially if in a different country.

In a similar way, the Taxation offices in many countries have published ontologies permitting taxpayers to submit electronic tax returns. Since taxpayers in a given jurisdiction are by law subject to the regulations of the local Taxation office, the local tax return ontology has a low ontological commitment if used to submit local tax returns, but a very high ontological commitment if used in another jurisdiction. Generally speaking, if an institution has the role of defining the rules of the game in a particular jurisdiction, the ontology published by that institution has a low ontological commitment for agents playing the game within that context, and a high commitment elsewhere.

Many ontologies are created with a very wide context. The Periodic Table represents laws of physics and chemistry that are assumed to be universal in the Universe. Hydrogen is hydrogen here and now, and everywhere else in the Universe since the Big Bang. So the Periodic Table has a low ontological commitment. The same can be said of the dimension ontologies for physical objects, for the same reasons.

The rules of games are also ontologies with a low ontological commitment, since if you are going to play that game, you must play according to its rules. The Tic-Tac-Toe ontology or a Chess ontology would have a low ontological commitment.

What we have so far is that an ontology has a level of ontological commitment relative to context. We can speak of absolute ontological commitment only for ontologies with a universal scope. But these are only peripherally design issues, they are more just the way things are.

Ontological commitment can be a design issue, though. Consider the Z39.50 example of Chapter 4. In its original formulation, Z39.50 included not only the ontology of Figure 4 but also the Bib-1 use attributes. This meant that any agent committing to the Z39.50 ontology needed to commit both to an information access world and to a set of attributes developed to support library catalog access. A plausible re-use of Z39.50 is for access to news stories. In this context, the ontological

commitment of Z39.50 is fairly high, since very few of the Bib-1 attributes are useful in a news access application. In the most recent version of Z39.50, the Bib-1 attribute set has been separated from the information access world, so an application can commit to Z39.50 without committing to Bib-1. Ontological commitment for each of the components is much less than for the combination.

Consider now an ontology supporting movies, including production, reviews, and showings in theaters. Plausible re-uses include television production, which is similar to movie production; book reviews which are similar to movie reviews, and live theater, which has show times and so on similar to movie screenings. The ontological commitment of the movie ontology in these related contexts is quite high since in each case two of the three aspects of the movie world are not relevant even if the third is. Re-usability is greatly increased if the movie ontology is divided into three freestanding ontologies for production, reviews and theatres. The application which the original monolithic movie ontology supported could equally well be implemented by a commitment to the three separate ontologies, with a small amount of integration.

Modularization is good ontological engineering practice just as it is good software engineering practice generally.

Even the Taxation office ontologies tend to be highly modularized. The personal income tax ontology is generally separated from the corporate tax ontology, sales tax ontology, and the excise tax ontology. Even within the personal income tax context, there is generally a basic return for the simplest taxpayer, with additional modules for people with investment income or depreciation of business assets. Even though the ontological commitment of each of the modules is high outside a particular tax jurisdiction, the modularization reduces the commitment for a particular user.

The Semantic Web encourages responsible bodies to publish data structures they maintain as ontology modules, like the Z39.60 use attribute sets, the SIC and SNOMED. Another possibility is postal codes. The postal service in many countries maintains a set of postal codes to facilitate mail sorting and delivery. These are typically published as printed books or supplements to telephone directories. They could be published electronically. An ontology using addresses could import modules of postal codes from the relevant postal services. It would be possible for a body like the International Postal Union to maintain a repository of such modules for more convenient access. Use of such modules reduces ontological commitment and facilitates interoperability among exchanges.

Judicious abstraction can also reduce ontological commitment by widening the context in which an ontology can be reused. Gruber describes an ontology of publications, intended to support bibliographic citations. One aspect of the world the ontology conceptualizes is that publications are published in time. It is important to be able to say that one paper was published before, after, or about the same time as another. ("About the same time as" is a significant concept because in the scholarly literature a paper can take many months or years between the time it is first written to the time it appears.) Journals or conferences of record are published at a wide variety of frequencies: weekly, biweekly, monthly, quarterly, annually, biennially and so on. Other sorts of publications are published on specific dates, or even at specific times (think of breaking news stories or e-mail messages). Different calendars are used in various parts of the world.

Gruber's example conceptualizes the time of publication as *timepoint*, which is sufficient to support before, after and the same. This very abstract concept has a low ontological commitment, because it can be re-used in a wide variety of contexts. A

particular context like a bibliographic system supporting a branch of science would introduce classes of representations of *timepoint* sufficient for the publications covered by that system, and a system of conversion among the various representations. We could see timepoint.(Gregorian)year, timepoint.muslimyear, timepoint.(Gregorian) month, and so on. People could publish specific time ontologies as specializations suited to particular classes of applications, so that the designers of a given application could commit to the general bibliographic ontology and also to a more specific time ontology.

Instead of combining modules at about the same level of abstraction and about the same level of ontological commitment relative to the context, the system builders would commit to a low-commitment ontology and a higher-commitment specialization.

8.3 Cost Considerations

Increased quality is a good thing, but everything done to increase quality increases cost as well. So we need to think about the balance between cost and benefit.

First, we must understand that an ontology is not a finished product, but a component of the design specification of possibly very many systems. It is widely recognized in software engineering that increased effort spent in the design of a system can often be repaid many times over by savings in the implementation. Mistakes discovered later in the project cost much more to fix than mistakes discovered earlier.

In order to assess the cost of an increase in quality, we therefore need to know about the system of which it is a part. There is a huge variety of systems using ontologies, as we will see in Chapter 9. For the present purpose, we can distinguish three kinds of situation:

- Ontology supports a single exchange. It will be implemented once by each player in the exchange. The bookseller exchange of Chapter 1 is an example. So is the Tic-Tac-Toe ontology of Chapter 4.
- Ontology supports a class of exchanges. It will be implemented, perhaps with modifications, for each exchange. The resulting modified ontology will then be implemented once by each player in each exchange. In Chapter 4, the Z39.50 ontology taken together with one of the sets of use attributes like Bib-1 is an example.
- Ontology is intended as a component of a wide variety of ontologies supporting a wide variety of exchanges. From Chapter 4, Z39.50 excluding any of the use attribute sets, the SIC, SNOMED, Periodic Table and the dimension ontologies are all examples of this kind.

We can see immediately that even in the first case an ontology will be a component of the design of many implementations, so we must take quality seriously.

Cost of a particular aspect of quality is strongly affected by the software tools available to support it. When graphical conceptual modeling languages like ERA were first introduced, the diagrams had to be drawn by hand. This made them extremely difficult to modify. They tended therefore to be used to gain a preliminary understanding of the universe of discourse which was then represented and maintained further in SQL table schemas. The cost of using the conceptual model as an integral part of the system design and document throughout the system's life was too high. Not until the development of computer-aided software engineering (CASE) tools which

allowed the diagrams to be edited, would automatically render them, and would automatically create the SQL table schemas did ERA and other such methods become fully integrated into the design process and software life cycle.

The implementation benefit of an increase in quality of design depends on the software tools available to support the implementation. One potential benefit of using a conceptual modeling language like ERA in the design of an information system is that it makes the structure and content of the system more visible to users, so can suggest queries those users might wish to make. There are even tools designed to enable a user to formulate a query by navigating through a conceptual model. But specifying a query is only part of the problem. In order for the query to produce a result, a program must be written. Without a database manager implementation platform, it is extremely difficult to generate the programs needed to execute a query. So in the absence of such platforms, it is impossible to take advantage in an implementation of increased quality in the design.

To be more concrete, we will look in more detail at one of the recommendations for increased clarity: replacing declared subclasses with defined subclasses.

We increase clarity when we replace a simple subclass declaration with a predicate we can use to determine whether an instance of the superclass is an instance of the subclass. In the Tic-Tac-Toe ontology, the subclasses are all definable in terms of values of properties and membership in other subclasses. In Chapter 13 we present definitions of all the subclasses in the Tic-Tac-Toe ontology in Common Logic. But there are many languages available for representing these predicates, including SQL.

In order to think about the cost-benefit tradeoff of defining subclasses, we need to know what we intend to do with the predicates. One possible use is for players to assess the state of the game during play as factors in their choice of move. In this case, the implementations need to be able to execute the rules on a game instance. One way to do this is to use SQL. The rules are implemented in SQL as view definitions, so can be evaluated by the database manager. A second and perhaps simpler way is to implement the rules as a series of if-then-else statements in the programming language used by the player to implement their agent. The predicates are fairly simple, so the implementation in a programming language is not too difficult. Further, the predicates are very stable. It is hard to see how they would ever change. So it is unlikely to be necessary to revise the implementations. The decision would very likely be to define the predicates in the ontology.

A second possible use is to check the rules for consistency and redundancy. There are also tools to assist with this, most notably description logics. Description logics can automatically test collections of rules for consistency and redundancy, but there are restrictions on the predicates which can be expressed in description logics. OWL DL, presented in Chapter 11, is an example of a description logic. In particular, not every predicate expressible in SQL or in Common Logic can be expressed in a description logic. But even these richer predicate definition languages have theorem prover tools available which can semi-automatically help the designers do the necessary checks. Rule checking is possible, but more expensive than populating the subclasses. But the rules need be checked only once, by the ontology designer. All the players in all the exchanges which use the ontology need only know that the ontology has been certified as consistent and non-redundant. So the benefit could easily exceed the cost many times.

Now consider the SIC ontology, which consists entirely of a system of declared subclasses. The SIC was developed by the US Bureau of Labor Statistics as a

classification system to publish a wide variety of industry-specific statistical tabulations, as can be seen on their web site[14]. The information used to prepare these statistical tabulations comes from questionnaires answered by the relevant establishments. Some of the information on the questionnaires is used by staff in the Bureau to classify the establishments. The Bureau has rules for assigning establishments to classes that could in principle be represented as predicates in a formal language, so the predicates defining the subclasses could be published.

The problem in publishing the subclass definitions is that the classifications are made using information provided to the Bureau by establishments on a confidential basis. So even though the properties needed to define the predicates actually exist, they are not available outside the bureau. Turning the SIC into a system of defined subclasses is forbidden by law.

Other applications of the SIC include yellow pages style directories. In these applications, the establishments decide for themselves how they wish to be classified. There are essentially no rules. The same is true of the subclasses of the book exchange determined by the catalogs of individual players discussed in Chapter 6. There is no predicate which can describe the subclass of books offered for sale by OntologyBooks.com.

An even more extreme example comes from the SNOMED ontology. One of its facets is *Diagnoses*, which like SIC is a system of declared subclasses. Medical diagnoses are judgments made by physicians of various specialties. There are specialties, notably Radiology, most of whose content is learning to make such judgments. Even though the diagnoses are made on the basis of observable properties, the predicates are unknown. It is beyond the present state of technology to publish *SNOMED Diagnoses* as a system of defined subclasses.

In a different direction, the Periodic Table is a system of individuals in declared subclasses. It would be possible to define the subclasses using underlying physical properties of the element individuals. But it is hard to see what benefit these definitions would bring for most applications, since the population of individuals is very stable. It would be an unusual application which needed to classify an unknown individual.

8.4 Discussion

In this Chapter we have looked at some dimensions of quality of ontology, in particular Gruber's principles: clarity, coherence, extendibility, encoding bias and ontological commitment. We make here a few comments about these dimensions as a system:

- Clarity is central, in that the other four dimensions make less sense if the ontology is not clear.
- The first two are aspects of quality of a particular ontology. Extendibility measures quality of a particular ontology under its evolution. The last two are aspects of ontology re-use.
- Ontological commitment is strongly relative to context, while the first four dimensions are pretty well universal.
- We try to maximize the first three and to minimize the last two.

[14] http://www.bls.gov/

8.5 Key Concepts

Quality principles for ontologies include **clarity, coherence, extendibility, encoding bias** and **ontological commitment.** We want to maximise the first three and minimise the last two.

8.6 Exercise

1. (relevant to points 10 and 11 of the major exercise) Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a. Criticise the ontology in terms of the five principles of Gruber.

- Clarity: suggest a plausible unintended interpretation of one of the concepts. How does (or could) the ontology prevent that unintended interpretation?
- Coherence: suggest a plausible inference that an agent could be expected to draw from the ontology. How does (or could) the ontology support the reasoning necessary to make the inference?
- Extendibility: suggest a plausible extension to the ontology. Show what changes would need to be made. Are any of the changes redundant? If so, show how. If not, show how the ontology design anticipated the extension.
- Encoding bias: Show how one of the actions of the exercise from Chapter 2 might be implemented. Does it make sense for it to be implemented in a different way? If not, why not? If so, does the ontology make the different implementation difficult? Consider each element of the implementation of the action.
- Ontological commitment: There are many different systems of interoperations in which the ontology could be reused. Describe one such system and how the ontology could be adapted to the re-use.
- In each quality dimension, indicate whether in your judgment the quality is high or low.

a. Propose an improvement to the ontology. Argue why this improvement is a good idea in terms of at least one of Gruber's principles.

b. Examine the cost and benefits of the improvement, taking into account the generality of the ontology and the number of implementations one might expect it to have. On balance, is the improvement a good idea? Take a position and justify it.

8.7 Further Reading

The quality principles were first published in [14]. Debenham's principles are presented in [15].

9 Uses of Ontology

We have alluded to classes of applications of ontology. There is an enormous variety. In this Chapter we attempt a systematic survey of the kinds of ontology there are.

9.1 Perspectives

In order to ensure a relatively complete representation of kinds of ontology use and their associated example applications, the analysis uses a set of perspectives that characterize the domain. Table 3 provides an overview of these perspectives. This analysis is based on the Object Management Group (OMG) Ontology Development Metamodel (ODM) standards development. (The OMG is responsible for systems engineering standards like UML.)

We have seen in Chapter 8 that an ontology is a part of a software development process. The ontology itself is a specification of a conceptualization in some area. Different areas have different sorts of conceptualizations. They also differ in the costs and benefits associated with different specifications. The perspectives associated with the conceptualizations and the organization that produce the specifications are called *model centric*.

On the other hand, the ontology is used in the software development process in different ways. We have seen three of these in Chapter 8. The perspectives that take account of how the ontology participates in the software development process are called *application centric*.

The *model centric* perspectives characterize the ontologies themselves and are concerned with the structure, formalism and dynamics of the ontologies, they are:

- *Level of Authoritativeness*

  The conceptualization from which an ontology is developed is always produced by someone. The institutional facts conceptualized are produced by some institution. If the ontology is developed by the institution which is responsible for producing the conceptualization, then it is definitive, therefore highly authoritative. If the ontology is developed by an organization distant from the producing institution, it is generally not very authoritative.

  Highly authoritative ontologies are part of the institutional environment of the organizations which will use them. If the conceptualization is complex, it often pays to develop the specification in great depth. But the authority of the responsible institution is limited, so the specification will generally have sharp boundaries, so will be relatively narrow. Ontologies which are *not authoritative* are often broad, since the creator can pick the most accessible concepts from many conceptualizations, but generally not very deep. The creator may not have access to the detail, or to the current definitive detail. The ontology cannot be relied upon by its users, so will generally not attract the resources to be developed in great detail.

  SNOMED is a very large and authoritative ontology. The periodic table of the elements is very authoritative, but small. However, it can be safely used as a

component of larger ontologies in physics or chemistry. Ontologies used for demonstration or pedagogic purposes, like the Wine Ontology[15], are not very authoritative. Table 3 can be seen as an ontology which at present is not very authoritative, since it was recently proposed by a small group of ontology workers. Should the classifications gain wide use in the ontology community, the ontology in Table 3 would become more authoritative.

- *Source of Structure*

  An ontology is a structure eventually implemented in software of some kind. In some cases, the structure is essentially the rules of the game. It is impossible to interoperate without using the published structure, and there are no actions which allow the structure to be changed. If the organization responsible for the ontology decides a change is needed, it will be made as a software update and published as a new version of the ontology. All the ontologies we consider in this text are of this kind. An ontology that defines the rules of the game is called *transcendent*. SNOMED is a transcendent ontology defined by the various governing bodies of medicine. E-commerce exchanges are generally supported by transcendent ontologies.

  On the other hand, the structure can be defined by patterns arising from the data produced by the interoperations, inferred using applications. This sort of ontology is called *immanent*. There are many applications where the ontologies are immanent, but the topic has not been well studied in the ontology field. For example, consider a newsfeed. A useful structure to the users of the newsfeed is the topics a news item relates to. These topics arise from the news itself. Most changes in the structure of topics are fairly minor. A new person is elected as the head of government, but the government stays the same, for example. But when something new happens, the structure of the topics can change radically. The outbreak of a war can do this, as can the introduction of a new technology like the World-Wide Web or mobile telephones.

  The applications extracting the structure from ongoing interoperations are generally statistical, often called some form of data mining. Besides news they are widely used in customer relationship management (think of the suggestions Amazon.com makes), search engines, and security applications. An example of the last is the detection of unusual patterns of credit card activity which may be indicators of fraudulent use.

- *Degree of Formality*

  We have seen in Chapter 8 that the specification of the conceptualization can be higher or lower on the quality dimension *clarity*. Degree of formality refers to the level of formality of the specification of the conceptualization, ranging from *highly informal* or taxonomic in nature, where the ontologies may be tree-like, involving inheritance relations, to semantic networks, which may include complex subclass/superclass relations but no formal axiom expressions, to ontologies containing both subclass/superclass relations and *highly formal* axioms that explicitly define concepts. SNOMED is taxonomic, as is the SIC, while engineering ontologies like dimensions are highly formal. The Tic-Tac-Toe ontology is intermediate.

[15] http://www.w3.org/2001/sw/WebOnt/guide-src/wine.owl

- *Model Dynamics*

  All ontologies have structure. It is often necessary to change the structure. If the ontology is transcendent, the responsible organization may decide to make a change, while if the ontology is immanent, new patterns may arise in the interoperation data. The question is, how often is the structure changed? One extreme in the model dynamics dimension is *stable* or *read-only*. The ontology rarely changes its structure. The Periodic Table is very stable in its structure, as is the Tic-Tac-Toe ontology, or generally the rules to any game. SNOMED is pretty stable, as is the SIC (the SIC is in process of being replaced by the North American Industry Classification System or NAICS[16], after 60 years, due to structural changes in the American economy in that period).

  The other extreme in model dynamics is ontologies whose structure is *volatile*, changing often. An ontology supporting tax accounting in Australia would be volatile at the model level, since the system of taxation institutional facts can change, sometimes quite radically, with any Budget.

- *Instance Dynamics*

  An ontology is generally specified as a system of classes and properties (the structure) which is populated by instances (the extents). As with model dynamics, the instances in an ontology can be *stable* (*read-only*) or *volatile*. The Periodic Table is stable at the instance level (e.g. particular elements) as well as the model level (e.g. classes like noble gasses or rare earths). New elements are possible but rarely discovered. On the other hand, an ontology supporting an e-commerce exchange would be volatile at the instance level but possibly not at the model level. Z39.50 based applications are very stable in their model dynamics, but very volatile in their instance dynamics. Libraries are continually turning over their collections.

Table 3. Perspectives of applications that use ontologies that are considered in this analysis.

Perspective	***One Extreme***	***Other Extreme***
Level of Authoritativeness	Least authoritative, broader, shallowly defined ontologies	Most authoritative, narrower, more deeply defined ontologies
Source of Structure	Passive (Transcendent) – structure originates outside the system	Active (Immanent) – structure emerges from data or application
Degree of Formality	Informal, or primarily taxonomic	Formal, having rigorously defined types, relations, and theories or axioms
Model Dynamics	Read-only, ontologies are static	Volatile, ontologies are fluid and changing.
Instance Dynamics	Read-only, resource instances are static	Volatile, resource instances change continuously
Control / Degree of Manageability	Externally focused, public (little or no control)	Internally focused, private (full control)
Application Changeability	Static (with periodic updates)	Dynamic
Coupling	Loosely-coupled	Tightly-coupled
Integration Focus	Information integration	Application integration
Lifecycle Usage	Design Time	Run Time

Application centric perspectives are concerned with how applications use and manipulate the ontologies, they are:

[16] http://www.bls.gov/bls/naics.htm

- *Control / Degree of Manageability*

  An ontology, like any piece of software, is subject to change. The issue in this dimension is who decides when and how much change to make. One extreme is when the body responsible for the ontology has sole decision on change (*internally focused*). The SIC is internally focused. Change is required because the structure of the US economy has changed over the years, but the Bureau of Labor Statistics makes the decision as to how to change and when the change is introduced.

  The other extreme is when changes to the ontology are mandated by outside agencies (*externally focused*). In the US, ontologies in the finance industry were required to change by the Sarbanes-Oxley Act of 2002, and changes in ontologies in many areas were mandated by the Patriot Act, passed shortly after the World Trade Center attacks in 2001. An ontology on taxation matters managed by a trade association of accountants is subject to change as the relevant taxation acts are changed by governments.

- *Application Changeability*

  An ontology is eventually implemented in applications. The applications may be developed once, as for an e-commerce exchange (*static*). Of course there may be periodic updates. On the other extreme the applications may be constructed dynamically on the fly (*dynamic*), as in an application that composes web services at run time. In this case, the applications available to the end user come and go according to user requests.

- *Coupling*

  An ontology is generally implemented many times in many applications. The issue in this dimension is how closely coupled the applications are to each other. The applications in an e-commerce exchange are *tightly coupled* to each other, since they must interoperate at run time. At the other extreme, the applications using the Periodic Table or the dimension ontology may have nothing in common at run time. They are *loosely coupled*, solely because they share a component.

- *Integration Focus*

  Some ontologies specify the structure of interoperation but not the content. Z39.50 exclusive of the use attribute sets is a good example. The MPEG-21 multimedia framework[17] is another example. It specifies the structure of multimedia objects without regard for their content. This extreme is called *application integration*, because they can be used to link programs together so that the output of one is a valid input for the other.

  Other ontologies specify content. The ontology may be used to specify the structure of a shared database, so that the different players can exchange information about shared objects by withdrawing them from the shared database, updating them, and replacing them in the shared database. This extreme is called *information integration*.

  An application can have both application and information focus.

- *Lifecycle Usage*

  An ontology is implemented in application software. In some cases, the application is used in the specification or design phases of the software life cycle and never again referred to explicitly. Use of the Periodic Table or dimension ontology in the specification of an engineering or scientific application is an

[17] http://www.chiariglione.org/mpeg/standards/mpeg-21/mpeg-21.htm

example of this extreme. The ontology is used at *design time*. In other cases, the ontology is used continually in the operation of the application. In a large e-commerce exchange, the exchange may check every message to see whether it conforms to the ontology and if so, what version. The message is then sent to the recipient with a certification, therefore relieving the players from having to do the checks themselves. In this case, the ontology is used at *run time*.

9.2 Application Types

As might be expected, some of these perspectives tend to correlate across different applications, forming application areas with similar characteristics. Table 4 shows three major clusters of application types that share some set of perspective values:

Table 4. Perspective values for different kinds of application of ontology

Characteristic Perspective Values										
	Model Centric					**Application Centric**				
Description	**Authori-tative-ness**	**Struct-ure From**	**Formality**	**Model Dy-namics**	**Instance Dy-namics**	**Control**	**Change-ability**	**Coupl-ing**	**Focus**	**Life Cycle**
Business Applications										
Run-time Inter-operation	Least/ Broad	Outside	Formal	Read-Only	Volatile	External	Static	Tight	Infor-mation	Real Time
Application Generation	Most/ Deep	Outside	Formal	Read-Only	Read-Only	External	Static	Loose	Applica-tion	All
Ontology Lifecycle	Middle/ Broad& Deep	Outside	Semi-Formal /Formal	Read-Only	Read-Only	External	Static	Tight	Infor-mation	Real Time
Analytic Applications										
Emergent Property Discovery	Broad & Deep	Inside	Informal	Volatile	Read-Only	Internal & External	Dy-namic	Flexible	Infor-mation	Real Time
Exchange of Complex Data Sets	Broad & Deep	Inside	Informal	Volatile	Read-Only/ Volatile	Internal & External	Dy-namic	Flexible	Infor-mation	Real Time
Engineering Applications										
Information System Development	Broad & Deep	Outside	Semi-Formal/ Formal	Read-Only	Volatile	Internal	Change-able	Tight	Infor-mation	Design Time
Ontology Engineering	Broad & Deep	Outside	Semi-Formal/ Formal	Volatile	Volatile	Internal	Change-able	Flexible	???	Design Time

- *Business Applications* are characterized by having transcendent source of structure, a high degree of formality and external control relative to nearly all users.
- *Analytic Applications* are characterized by highly changeable and flexible ontologies, using large collections of mostly read-only instance data.
- *Engineering Applications* are characterized by again having transcendent source of structure, but as opposed to business applications their users control them primarily internally and they are considered more independent of context.

We will now look at each of the kinds of application in more detail.

9.3 Business Applications

9.3.1 Run Time Interoperation

Applications involving information interoperation among autonomous agents, like the e-commerce example or a Z39.50 application with one of the sets of use attributes, are typically characterized by tight coupling of the components realizing the applications. They are focused specifically on information rather than application integration.

Here we include some semantic web service applications. These may involve composition of vocabularies, for example a gazetteer of names of geographical features from the US Geological Survey, a collection of place names and zip codes from the US Postal Service, and a collection of interstate highway names and the cities they interconnect from the US Federal Highway Administration could all be components of a tourism application. The same application could involve composition of services and processes, for example plane, train, car and hotel reservations, but not necessarily application program interfaces (APIs) or database schemas. The application would not support an SQL query across all hotels, nor the ability to pipeline a service request through an airline to a car rental company. Interaction with these two services would be facilitated by the application, but it would not build new data structures or compositions like unified airline reservation/ car rental packages which could be re-used by others.

Because the community using them must agree upon the ontologies in advance, their application changeability tends to be static in nature rather than dynamic.

Perspectives that strongly affect design in these scenarios include:

- That the ontologies and information resources be sufficiently authoritative to support the development investment.
- Whether control is externally or internally focused.
- Whether there is a design time component to ontology development and usage, or whether the ontology is adopted as a package.
- Whether or not the knowledge bases and information resources that implement the ontologies are modified at run time (since the source of structure remains relatively unchanged in these cases, or the ontologies are only changed in a highly controlled, limited manner).
- These applications may require mediation middleware that leverages the ontologies and knowledge bases that implement them, so lifecycle usage tends be run-time.

Further examples of run-time interoperating applications include:

- Semantically grounded information interoperability, supporting highly distributed, intra- and inter-organizational environments with dynamic participation of potential community members, with diverse and often conflicting organizational goals. When multiple emergency services organizations come together to address a specific crisis, the nature of the crisis determines which services become involved and whose concerns take precedence. In a fire in an industrial area, prevention of spread of the fire to chemical storage sites might take precedence over ambulance

services or extinguishing the fire itself, while in a train wreck ambulance services might be a priority.

- Semantically grounded discovery and composition of information and computing resources, including Web services. Business process integration, for instance getting hospital department information systems and the systems of associated practitioners and health insurance bodies to work together, is of this type, as is grid computing, where scientific computing resources implement specific data reduction, analysis and visualization algorithms which can be employed in sequence on a particular data set.
- In electronic commerce exchange applications based on stateful protocols such as EDI or Z39.50, where there are multiple players taking roles performing acts by sending and receiving messages whose content refers to a common world.

In these cases, we envision a number of agents and/or applications interoperating with one another using fully specified ontologies. Support for query interoperation across multiple, heterogeneous databases is considered a part of this scenario.

While the engineering requirements for ontologies to support these kinds of applications are extensive, key features include:

- the ability to represent situational concepts, such as player/actor – role – action – object – state,
- the necessity for multiple representations and/or views of the same concepts and relations,
- separation of concerns, such as separating the vocabularies and semantics relevant to particular interfaces, protocols, processes, and services from the semantics of the domain.
- Service checking that messages commit to the ontology at run time. These communities can have thousands of autonomous players, so that no player can trust any other to send messages properly committed to the ontology.

9.3.2 Application Generation

In this kind of application, the ontology is used to provide a common worldview, universe of discourse, or domain, which is used to constrain or even automatically generate applications or agents in particular environments. Flexible applications packages commonly found in manufacturing, the finance and insurance industries, and enterprise software like SAP can be seen as in this category, although in these systems the ontologies are not necessarily clearly separated from the applications.

Model-Driven Architecture (MDA) is an emerging method of systems development in which the ontology is much more visible. In MDA, the idea is to specify the application, including an ontology, in a formal modeling language like UML, with sufficient precision that the programs implementing the application can be automatically generated. One implementation of the application will differ from another because of different values of parameters in the model or different settings of technology choices in the application generation. This contrasts with application packages, which generally include the programs themselves.

For example, a specific hospital might have multiple, loosely coupled clinics. Patient management would be specified in a model, which would be specialized for each clinic's requirements by parameter settings in the model. The application might be generated to use CORBA or DCOM for communication for clinics connected to a

network within the hospital's firewall, but XML over HTTP for remote clinics. Standard reporting requirements and supply chain management applications are other candidates for MDA.

Characteristics include:

- Authoritative environments, with fairly tight coupling between resources and applications.
- Ontologies shared among organizations are highly controlled from a standards perspective, but may be specialized by the individual organizations that use them within agreed parameters.
- The knowledge bases implementing the ontologies are likely to be dynamically modified, augmented at run time by new metadata, gathered or inferred by the applications using them. (A *knowledge base* is a database of structured information, typically implemented in languages like RDFS or OWL.)
- The ontologies themselves are likely to be deeper and narrower, with a high degree of formality in their definition, focused on the specific domain of interest or concepts and perspectives related to those domains.

Other examples include:

- Dynamic regulatory compliance and policy administration applications for security, logistics, manufacturing, financial services, or other industries. This includes things like anti-money-laundering regulations, security of baggage handling by airlines and airports, and regulations that people convicted of certain kinds of crimes not be employed in tasks where they might be subject to undue temptation.
- Applications that support sharing clinical observation, test results, medical imagery, prescription and non-prescription drug information (with resolution support for interaction), relevant insurance coverage information, and so forth across clinical environments, enabling true continuity of patient care. This is a much tighter integration than the semantically grounded discovery and interaction described as run-time interoperation in the previous section.

Requirements:

- The ontologies used by the applications may be fully specified where they interoperate with external organizations and components, but not necessarily fully specified where the interaction is internal. The prescribing doctor does not need to know the pharmacy's inventory management procedures, nor does the pharmacy need to know the doctor's schedule for making hospital rounds, but they do need to agree on identification of patients, drugs and the specification of doses and frequency of administration.
- Conceptual knowledge representing priorities and precedence operations, time and temporal relevance, bulk domains where individuals don't make sense, rich manufacturing processes, and other complex notions may be required, depending on the domain and application requirements.

9.4 Ontology Lifecycle

In this scenario we are concerned with building and maintaining ontology-based applications. The principal objectives of ontology lifecycle applications are conceptual knowledge analysis, capture, representation, and maintenance. Ontology repositories

should be able to support rich ontologies suitable for use in knowledge-based applications, intelligent agents, and semantic web services. Examples include:

- Maintenance, storage and archiving of ontologies for legal, administrative and historical purposes,
- Test suite generation, and
- Audits and controllability analysis.

Ontological information will be included in a standard repository for management, storage and archiving. This may be to satisfy legal or operations requirements to maintain version histories.

These types of applications require that knowledge engineers interact with subject matter experts to collect knowledge to be captured. UML models provide a visual representation of ontologies facilitating interaction. The existence of meta-data standards, such as XMI (XML metadata interchange which enables UML models to be serialized and exchanged) and ODM (the Ontology Development Metamodel, an integration of UML, OWL and other representation systems), will support the development of tools specifically for quality assurance engineers and repository librarians.

Requirements include:

- Full life-cycle support will be needed to provide managed and controlled progression from analysis, through design, implementation, test and deployment, continuing on through the supported systems maintenance period.
- Part of the lifecycle of ontologies must include collaboration with development teams and their tools, specifically in this case configuration and requirements management tools. Ideally, any ontology management tool will also be ontology aware.
    - It will provide an inherent quality assurance capability by providing consistency checking and validation.
    - It will also provide mappings and similarity analysis support to integrate multiple internal and external ontologies into a federated web.

More detail on the tasks performed by ontology servers is presented in Chapter 15.

9.5 Analytic Applications

9.5.1 Emergent Property Discovery

By this we mean applications that analyze, observe, learn from and evolve as a result of, or manage other applications and environments. The ontologies required to support such applications include ontologies that express properties of these external applications or the resources they use. The environments may or may not be authoritative; the ontologies they use may be specific to the application or may be standard or utility ontologies used by a broader community. The knowledge bases that implement the ontologies are likely to be dynamically augmented with metadata gathered as a part of the work performed by these applications. External information resources and applications are accessed in a read-only mode.

Examples include:

- Semantically grounded knowledge discovery and analysis (*e.g.,* financial, market research, intelligence operations). Specific ontologies include patterns estimated

from the data using data mining techniques. Utility ontologies might include lists of names of well-known people.

- Semantics assisted search of data stored in databases or content stored on the Web (*e.g.*, using domain ontologies to assist database search, using linguistic ontologies like WordNet to assist Web content search).
- Semantically assisted systems, network, and / or applications management. Knowing about the environment of a telecommunication system can be useful. For example, knowing that it is Mothers' day in a particular country can enable setting network parameters to support historically-experienced very high call volumes, perhaps at the expense of low-value bandwith-consuming applications like television program transfer.
- Conflict discovery and prediction in information resources for self-service and manned support operations (*e.g.,* technology call center operations, clinical response centers, drug interaction). In a call centre we might discover that emergency calls frequently come in on a fault-reporting service, so we might introduce a facility for forwarding the calls quickly to the appropriate center.

What these have in common is that the ontology is typically not directly expressed in the data of interest, but represents theories about the processes generating the data or emergent properties of the data. These ontologies are often immanent.

Requirements include representation of the objects in the ontology as rules, predicates, queries or patterns in the underlying primary data.

9.5.2 Exchange of Complex Data Sets

Applications in this class are primarily interested in the exchange of complex (multi-media) data in scientific, engineering or other cooperative work. The ontologies are typically used to describe the often complex multimedia containers for data, but typically not the contents or interpretation of the data, which is often either at issue or proprietary to particular players.

Examples include:

- Suites of applications which can create, edit, modify and annotate MPEG-21 multimedia objects, possibly as dynamically composed web services
- Suites of applications which can perform simulations, extract subsets of the outputs of simulations, analyze and visualize the extracts, possibly also dynamically composed
- Suites of applications which record DNA and protein sequences, perform searches for similar sequences with given parameters, permit addition of annotations as metadata, and make hypotheses about potential combinations given similarity of structure.

Here the ontology functions more like a rich type system. It would often be combined with ontologies of other kinds (for example an ontology of radiological images might be linked to SNOMED for medical records and insurance reimbursement purposes).

Requirements include

- Representation of complex objects (aggregations of parts)
- Multiple inheritance where each semantic dimension or facet can have complex structure.

- Tools to assemble and disassemble complex sets of scientific and multi-media data.
- Facilities for mapping ontologies to create a cross-reference. These do not need to be at the same level of granularity. For the purposes of information exchange, the lower levels of two ontologies may be mapped to a higher-level common abstraction of a third, creating a sort of index. We might map a detailed DNA sequence structure ontology and a detailed protein sequence structure ontology to an ontology of genetic abnormalities.

9.6 Engineering Applications

The focus of these kinds of applications is on the ontology development environment, rather than on the ontology itself. The requirements for ontology development environments need to consider both externally and internally focused applications, as externally focused but authoritative environments may require collaborative ontology development.

9.6.1 Information Systems Development

The kinds of applications considered here are those that use ontologies and knowledge bases to support enterprise systems design and interoperation. They may include:

- methodology and tooling, where an application actually composes various components and/or creates software to implement a world that is described by one or more component ontologies. The ontologies relevant here describe the components and how they can fit together.
- Semantic integration of heterogeneous data sources and applications (involving diverse types of data schema formats and structures, applicable in information integration, data warehousing and enterprise application integration). The relevant ontologies describe the abstract data structures used and how structures can be transformed and mapped.
- Application development for knowledge based systems in general. Here the relevant ontologies include the structures used to represent knowledge and how they fit together.

In the case of model-based (MDA) applications, *extent-descriptive metaclasses* are needed to provide enough meta-information to exercise design options in the generated software (e.g. describing class size, probability of realization of optional classes). An example paradigm might reflect how an SQL query optimizer uses system catalog information to generate a query plan to satisfy the specification provided by an SQL query. Similar sorts of predicates are needed to represent quality-type meta-attributes in semantic web type applications (comprehensiveness, authoritativeness, currency). This is described further in Chapter 12.

9.7 Ontology Engineering

Applications in this class are intended for use by an information systems development team, for use in the development and exploitation of ontologies that make implicit

design artifacts explicit, such as ontologies representing process or service vocabularies relevant to some set of components. Examples include:

- Tools for ontology analysis, visualization, and interface generation.
- Reverse engineering and design recovery applications.

The ontologies are used throughout the enterprise system development life cycle process to augment and enhance the target system as well as to support validation and maintenance. Such ontologies should be complementary to and augment other modeling artifacts developed as part of the enterprise software development process. The UML standards and supporting tools are a system of this kind.

Knowledge engineering requirements may include some ontology development for traditional domain, process, or service ontologies, but may also include:

- Generation of standard ontology descriptions (e.g. OWL) from UML models.
- Generation of UML models from standard ontology descriptions (e.g. OWL).
- Integration of standard ontology descriptions (e.g. OWL) with UML models.

Key requirements for ontology development environments supporting such activities include:

- Collaborative development
- Concurrent access and ontology sharing capabilities, including configuration management and version control of ontologies in conjunction with other software models and artifacts at the atomic level within a given ontology, including deprecated and deleted ontology elements
- Forward and reverse engineering of ontologies throughout all phases of the software development lifecycle. (*Forward engineering* means that the ontology is developed as part of the systems design phase of the software lifecycle. *Reverse engineering* means that the ontology is extracted from the documentation of previously existing applications, possibly with a view to re-engineering them.)
- Ease of use, with the user being able to hide as much of knowledge engineering detail as possible
- Interoperation with other tools in the software development environment; integrated development environments
- Localization support (different natural languages for user interface, cultural differences etc.)
- Cross-language support (ontology languages as opposed to natural or software languages, such as generation of ontologies in the XML/RDF(S)/OWL family of description logics languages, or in the Knowledge Interchange Format (KIF) or Common Logic (See Chapter 13), where first or higher order logics are required)
- Support for ontology analysis, including deductive closure; ontology comparison, merging, alignment and transformation
- Support for import/reverse engineering of RDBMS schemas, XML schemas and other semi-structured resources as a basis for ontology development

Some of these issues are discussed in more detail when we look at ontology servers in Chapter 15.

9.7.1 Goals for Generic Ontologies and Tools

The diversity of the usage scenarios illustrates the wide applicability of ontologies within many domains. Table 5 brings these requirements together. The table classifies the requirements into

- structural features – knowledge representation abstract syntax
- generic content – aspects of the world common to many applications
- run-time tools – use of the ontology during interoperation
- design-time tools – needed for the design of ontologies

To address all of these requirements would be an enormous task, and no existing tool or language supports all. The following Chapters will address some of them.

Table 5. Summary of requirements

Requirement
Structural features
Support ontologies expressed in existing description logic, (e.g. OWL/DL) and higher order logic languages (e.g. OWL Full and KIF), as well as emerging and new formalisms.
Represent complex objects as aggregations of parts
Multiple inheritance of complex types
Separation of concerns
Full or partial specification
Model-based architectures require extent-descriptive predicates to provide a description of a resource in an ontology, then generating a specific instantiation of that resource.
Efficient mechanisms will be needed to represent large numbers of similar classes or instances.
Generic content
Support physical world concepts, including time, space, bulk or mass nouns like 'water', and things that do not have identifiable instances.
Support object concepts that have multiple facets of representations, e.g., conceptual versus representational classes.
Provide a basis for describing stateful representations, such as finite state automaton to support an autonomous agent's world representation.
Provide a basis for information systems process descriptions to support interoperability, including such concepts as player, role, action, and object.
Other generic concepts supporting particular kinds of domains
Run-time tools
Tools to assemble and disassemble complex sets of scientific and multi-media data.
Service to check message commitment to ontology
Design-time tools
Full life-cycle support
Support for collaborative teams
Ease of use, transparency with respect to details
Support for modules and version control.
Consistency checking and validation, deductive closure
Mappings and similarity analysis

9.8 Key Concepts

There are many ways of looking at ontologies that can be organized as a number of **perspectives**. Ontologies can be classified using these perspectives as properties into a collection of more or less well defined **application types**. Different application types have different **engineering requirements**.

9.9 Further Reading

This taxonomy was published in [16].

10 Representations of Ontologies: RDFS

Now that we know how to design and build an ontology, we need to think about deploying it. This means that we need to represent it in some way that is understood by the systems that need to use it. This Chapter looks at various machine-readable ways, from XML to RDFS, leading up to OWL in Chapter 11.

10.1 Airlines Example

We begin with a simple example, which we will first explain then represent in various ways. Figure 25 shows a simple airline ontology in expressed in the representation language of Chapter 4. The visualization is changed so that some properties, those whose range is a literal rather than a class, are shown in the box with the class which is their domain: *depart_time*, *arrive_time*, *name* and *airport*.

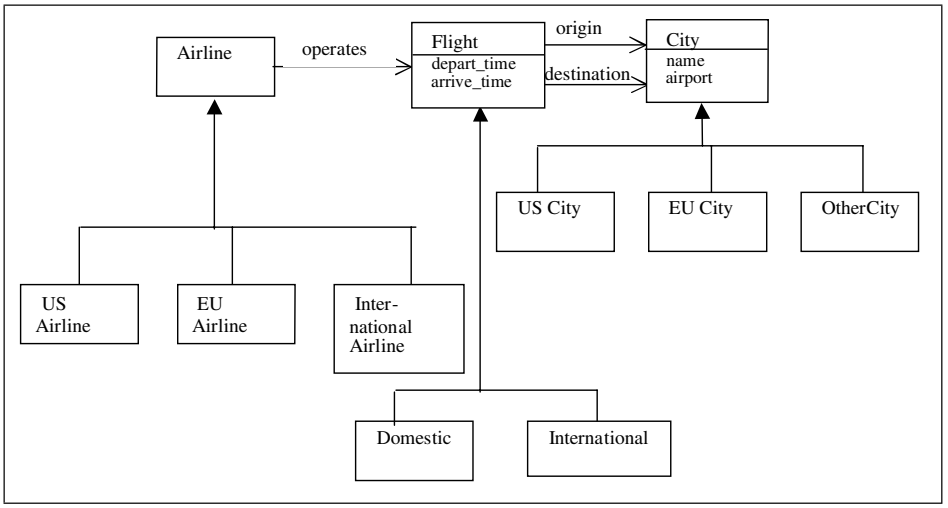


Figure 25. Airline Ontology

Airlines operate flights which have origin and destination in cities. There are US, EU and International airlines, and US, EU and other cities. Flights whose origin and destination are both in either the US or the EU are domestic, otherwise international. US and EU airlines only operate domestic flights. Each of these classes would have numerous instances. For example, QANTAS operates flight QF1 with origin Sydney and destination Los Angeles. This is an international flight operated by an international airline.

10.1.1 Advantages and Disadvantages of UML for Ontology

It happens, not entirely by accident, that Figure 25 can be read as a UML diagram as well as a visualization of our ontology representation language. An obvious question is to point out that if we already have a representation of the ontology in UML, why do we need to go further?

Representation in UML has the advantages that the UML Class Diagram is a rich representation system, widely used, and well suited to data structures of institutional facts, as we see in Figure 25.

But a UML Class Diagram is a specification for a system. It shows schemas, but does not fully specify instances. We know that the shared worlds modeled with ontologies contain instances as well as schemas, sometimes individuals which are outside the class system altogether. UML is intended to be used with some sort of implementation, like an SQL database manager, which completes the specification of the instances.

Further, a UML Class Diagram is generally used by the software engineers building a system as part of the design specification. It can be a component of a computer-aided software engineering tool which can automatically generate implementations. But class diagrams are not intended for public use, to be combined as components in larger ontologies, or to be used at run-time.

Further, and perhaps most importantly, an ontology by definition is intended to be reused, or to have multiple implementations. At least two. It must be possible for all parties committing to an ontology to have a way of verifying that everyone else is committed to it, too. The only way to do this is to check that each implementation generates the expected messages in the appropriate situations. This is done by the ontology representation language having a formal semantics expressed as a model theory. Two implementations which generate the same objects by definition agree. UML does not at present have a published model theory.

Fortunately, there are a large number of ways to represent ontologies which permit individuals, and which are intended to be accessed publicly by computerized agents. Many of these involve XML. XML is important because an ontology by definition exists outside all of the applications which use it. We expect it to be stored in a repository operated by say the exchange. In order for a player to commit to the ontology, the ontology must be transported from the repository to the player's information system. XML is a widely-used transport mechanism for complex structures over the Internet.

Furthermore, since much of the activity in ontologies is associated with the semantic web, which is associated with the World-Wide Web Consortium (w3c), and all w3c standards are based on XML, we can restrict our concerns to XML-based representations.

First, we will look at using bare XML, then Resource Definition Framework (RDF), then Resource Definition Framework Schema (RDFS), and finally Web Ontology Language (OWL) in Chapter 11.

10.2 Representation in Bare XML

XML is a way of defining markup languages. A markup language is a way of annotating (marking up) a text document so that its structural parts are reliably

indicated. So the first part of an XML markup is a specification of the markup language itself using the built-in markup elements forming *XSD* (*XML schema definition*). The second part is a body of text marked up with the defined language.

The first issue then becomes what to mark up. The ontology of Figure 25 has some of its content represented as text (the class and association names), some as graphic elements (the boxes denoting classes, the lines denoting associations), and some ambiguously (the instances of classes and associations do not have unique representations in UML, since UML models do not generally completely specify instances). Further, some of the information in diagrams like Figure 25 is never represented as content at all. The cardinality or multiplicity constraints on the associations define what is permitted as instances of the associations, but are never part of any instance.

Secondly, XML permits the definition of highly parameterized markup elements, so that for many applications, most of the content of the document is actually in the markup elements rather than in the marked-up text.

What people do with XML is represent a metamodel with markup elements, then represent the objects being sent with a combination of text and markup elements. This means that we can't work with bare XML. We need a structure to represent.

There is an XML version of UML, called XMI (XML Metadata Interchange), which is used for transferring models from one repository to another.

Starting with UML raises a number of problems.

First, it emphasizes that an ontology representation system must be able to represent not only structure but also populations, which are outside the scope of UML. Imagine that the sending site is an airline scheduling service and the receiving site is a chain of duty-free shops (Duty-Free Inc) operating in international airports. The receiving site wants the ontology as a component of its staff scheduling system so it can roster people to serve international passengers between flights, but not have the stores open when there is no possibility of business. The UML model of Figure 25 tells the Duty-Free Inc management that there are flights, but the shop management wants to know more than that. They want to know which flights serve which cities at which times. This is why just transmitting the UML model is not enough. There needs to be a way of representing individuals.

Further, Duty-Free Inc does not actually want the entire airline ontology. First of all, they are interested only in international flights, because domestic flights generally use different parts of airports and their passengers do not have access to duty-free shops. Secondly, Duty-Free Inc is interested only in when flights arrive and depart particular cities. They don't care where the flights go. Flight QF1 may depart Sydney at 1500 and arrive at Los Angeles at 0800, but the shop at Sydney cares only that a QANTAS flight departs at 1500 and the shop at Los Angeles cares only that a QANTAS flight arrives at 0800.

Duty-Free Inc is interested only in the *City* class in the airline ontology, the inverses of the *origin* and *destination* associations, and the *depart_time* and *arrive_time* attributes of the *Flight* class. To require them to know any more than that is another form of ontological overcommitment. So any solution involving transmitting the entire ontology is not ideal.

We want a representation which is not only public, but includes fully-specified individuals, and which has a smaller granularity than the whole ontology.

10.3 Resource-Definition Framework (RDF)

One way to get a smaller granularity, implementation-independent representation which includes individuals is to use the World-Wide Web Consortium's *Resource Description Framework*, or *RDF*.

10.3.1 RDF Triple

In the Web, the major organizing principle is the hyperlink, which ties together web pages and links parts of pages together. A hyperlink can be thought of as a source anchor linked to a target anchor. The source anchor is block of text which can be as small as a point in the document, and unless the target anchor is a URL, it is also a point in the document. It is not clear what the scope of a point anchor is. Even if a source includes some text it is not clear what is being linked to what. Furthermore, the link is not typed, so the nature of the relationship between source and target it is not clear.

RDF elaborates the hyperlink. But rather than representing a way for a user to move from place to place within the world-wide web, RDF represents definite objects called *resources*. We will look at resources more deeply in the next section, but at this point we need to know only that they can be much smaller in scale than a document.

The RDF equivalent of a hyperlink is a *triple*. The equivalent of the source anchor is a *subject*, the equivalent of the target anchor is an *object*, and the equivalent of the link is a *predicate*. Each of subject, predicate and object is a designated resource. For example, if we wanted to say that QF1 is operated by QANTAS, we would express "QF1", "is operated by" and "QANTAS" all as resources, and the statement as a triple with subject "QF1", predicate "is operated by" and object "QANTAS". Note that the equivalent of the hyperlink, the predicate, is a named resource. This name functions as the type of the hyperlink.

There are several notations for RDF triples used for different purposes. The most human-readable notation is called N3 notation. Using N3 notation, the statement "QF1 is operated by QANTAS" would be expressed as

QF1 is operated by QANTAS (9)

following the pattern *subject predicate object*. (This is not quite right. The correct expression will be shown below when a few more concepts have been explained.)

A collection of triples is called an *RDF Graph*. A collection of triples in the same document is called an *RDF document*. If the triples are expressed in N3 notation, the document is called an *N-triples document*. Since the RDF triple is an elaboration of the hyperlink, there is no reason why any of the resources need be in the same document as the triple itself. Just as in a web page we could make a hyperlink between "QANTAS" and its booking website, we can have a triple

QANTAS booking site <http://www.qantas.com> (10)

where the booking website is identified by its URL.

It is quite possible that all the resources in a triple are in different documents from the triple itself. We could have a document with airline names, and another document containing types of links, so (10) could be expressed as

< http://www.ontologies.org/airline#QANTAS> (11)
<http://www.ontologies.org/linktypes#booking-site>
<http://www.qantas.com>

using the prefix "#" to indicate that the resource is a fragment in the nominated document.

We now need to look at how resources are identified.

10.3.2 Names

RDF is intended to operate on the Web, and RDF statements are intended to be unambiguously interpreted by anyone who can access them. It takes an enormous amount of infrastructure to do this.

For example, suppose we wanted to say that our airline ontology was expressed in the English language. A first cut at a triple might be

airline ontology language English (12)

But suppose someone retrieved this triple as a result of a Google search. How do they know what this means? They don't even know that the text is an RDF N-Triple.

Fortunately, they do know something. In order to retrieve the document containing the triple, they know the document's URL and its access method, so they know it is a hypertext document. They can read the first few lines of text and see

<?xml version="1.0" encoding="UTF-8" ?> (13)
<!DOCTYPE rdf:RDF>

so they know it is an XML-encoded RDF document. They therefore know to interpret the text (12) as an RDF N-triple.

Now to get inside the triple. The subject is "airline ontology". The airline ontology is a specific artifact stored in a file with a URL, say http://www.ontologies.org/airline. So in order to avoid ambiguity we need to represent the triple (12) as

< http://www.ontologies.org/airline > language English (14)

The predicate "language" also needs an unambiguous meaning. It happens that the question of which natural language a document is written in is a very common question on the Web. There is a standard called *Dublin Core* for describing properties like this (called *metadata*), and one of the metadata elements it supports is what we want. The standard has a URL <http://purl.org/ dc/terms>. This particular version of the standard is represented as an RDF graph, and one of its subjects is the metadata term "language", which is identified by the fragment identifier "language". (In an RDF document, a fragment identifier picks out the subject of a triple.) A URL with a fragment identifier is an example of a Universal Resource Identifier (URI), a more general concept than URL (A URL is a URI.) An unambiguous representation of the predicate is therefore the URI <http://purl.org/ dc/terms#language>, and triple (14) becomes

< http://www.ontologies.org/airline > (15)
< http://purl.org/dc/terms#language> English

(Note that very recent material uses the term IRI rather than URI. An IRI is a URI represented in Unicode, so it can use characters from any language.)

Finally, the object "English" needs an unambiguous representation. The Dublin Core metadata standard recommends using a standard set of two and three character abbreviations for languages published by the International Standards Organization (ISO 639). In that standard, English is represented by the code "en". However, in ISO 639 the language identifier codes are not represented as the subjects of RDF triples, so do not have URIs. They are simply text strings.

So knowing that the predicate is a Dublin Core metadata element, and knowing Dublin Core's recommendation, the person retrieving our triple knows how to interpret the final unambiguous representation

< http://www.ontologies.org/airline > (16)
< http://purl.org/dc/terms#language> "en"

All this infrastructure is gathered together under the concept of a namespace. There are a number of widely-used namespaces in the semantic web community shown in Table 6.

Table 6. Widely-used namespaces

Namespace	prefix	namespace URI
RDF	rdf:	http://www.w3.org/1999/02/22-rdf-syntax-ns#
RDFS	rdfs:	http://www.w3.org/2000/01/rdf-schema#
Dublin Core	dc:	http://purl.org/dc/terms
OWL	owl:	http://www.w3.org/2002/07/owl#
XML Schema	xsd:	http://www.w3.org/2001/XMLSchema#

Note that, somewhat confusingly, a URI is thought of as a name, and does not have to actually address anything. That is, there does not have to be a page and fragment in which you can find the resource named. This is the case, for example, with the URI <http://www.w3.org/2001/XMLSchema#>. In these cases, the URI is simply a name for the resource. Its interpretation would be built into the software you would use to create the RDF representation of an ontology. This is comparable to the way HTML is distributed. It has an official definition on the W3C site, but its implementation is hardwired into web servers and web browsers.

It is common to abbreviate namespaces. In the RDF declaration, namespace abbreviations can be introduced as attributes. For example, the Dublin Core prefix in Table 6 can be introduced into an RDF document with the declaration

<rdf:RDF xmlns:dc = "http://purl.org/dc/terms"> (17)

so our triple (8) would now be

< http://www.ontologies.org/airline > dc:language "en" (18)

The abbreviated URI *dc:language* is called a *qualified name* (*QName*) and is written without angle brackets.

The W3C namespaces in Table 6 are all automatically available as prefixes.

Our ontology itself defines a namespace. If we include a prefix definition for it in the document header

<rdf:RDF xmlns:air = "http://www.ontologies.org/airline" (19)
xmlns:linktypes = "http://www.ontologies.org/linktypes#">

our earlier examples (9) and (10) can be correctly represented as

air:QF1 air:is operated by air:QANTAS (20)
air:QANTAS linktypes:booking site http://www.qantas.com

Namespaces are an extremely important concept in the semantic web. Many components of ontologies are systems of institutional facts maintained by organizations with a wide scope. Table 6 lists components published by the World Wide Consortium and Dublin Core. One might expect that some time in the future, the language codes will be published as RDF resources by the International Standards Organization. There are many other such systems of codes: currencies, airports, postal codes, state or province codes. The SIC, SNOMED and the Z39.50 use attribute sets could all be published as systems of resources in globally identified namespaces. As the semantic web evolves, a large number of standard components will become available.

10.3.3 Literals

The names we have been using for our ontology in the previous section have mostly been pretty substantial. A resource name requires a declaration in a namespace, a large investment. Sometimes we want names which don't merit that level of investment. For example, in the airline ontology a flight has a departure time and an arrival time. Say flight QF1 departs at 0815 Sydney time. The time '0815' has no special status, so we wouldn't expect some standards body to have established a resource in a namespace for it.

On the other hand, a user of the ontology needs some help in understanding how to interpret the text string '0815'. (This sort of thing is called a *literal*.) The usual way literals are interpreted in computer languages is by declaring them to be an instance of a type. So we attach a declaration to '0815' that it is to be interpreted as a time in the 24-hour notation. The time 0815 may be a lightweight notion, but the type merits the investment in a resource.

As with standard namespaces as illustrated in Table 6, there are standard namespaces containing type declarations. In particular, there is an International Standards Organization standard for date and time which has been published as a resource by the World-Wide Web Consortium as a part of the XML Schema namespace. So the departure time of flight QF1 at 0815 Sydney time would be represented by the literal "08:15:00+08:00", which is interpreted as hours, minutes, seconds, 8 hours ahead of Coordinated Universal Time (UTC, very close to Greenwich Mean Time or GMT). XML Schema includes a number of general datatypes including different kinds of numbers, strings, date and time, and so on[18].

The type of a literal is indicated by appending its type declaration URI, for example (using the prefix *xsd* from Table 6)

"08:15:00+08:00"^^xsd:time (21)

Recall that this particular XML Schema URI does not actually address anything, but is simply a unique identifier for the type.

XML Schema literal types include *integer*, *string*, *date*, as well as *time* and several others.

Since the semantic web is an open environment, literals in RDF are specified in quite a different way from the way literals are specified in a typical programming language. In for example SQL literal specification includes the syntactic form of the literal tokens, a parser to interpret these syntactic forms, and a collection of procedures implementing operations on various types of literals. RDF in itself specifies none of these. For the syntactic form of literals, RDF refers to the specifications in XML. But XML includes neither parsers for literals nor procedures implementing operations on them. If an application wants either of these, it must either develop them itself or import them from some third party. They could be provided by an RDF-aware ontology server tool, as described in Chapter 15.

[18] Documented at http://www.w3.org/TR/2004/PER-xmlschema-2-20040318/datatypes.html#built-in-datatypes. The time type is documented at http://www.w3.org/TR/2004/PER-xmlschema-2-20040318/datatypes.html#time

10.3.4 Blank Nodes

Sometimes there are objects in an ontology which do not have independent names. RDF has the concept *blank node* to deal with this situation. For example, in the airline ontology in Figure 25 there is a class *City* with two attributes *name* and *airport code*. So the city New York has a name "New York" and an airport code, "JFK". The ontology has a concept "city" but representations for only city name and airport code.

This situation might not be good enough. Consider expanding the ontology to include the city's name in several different languages. The city might (like New York) have several airports. We might want to include a map of the city, weather information and so on. RDF allows us to create a resource without a definite representation which ties these together. This resource is called a blank node. Its name is prefixed by "_:". So the ontology's representation of New York would use a blank node to tie together all the other representations, for example

```
_:NewYork    air:name       "New York"^^xsd:string                    (22)
_:NewYork    air:airport    iata:JFK
_:NewYork    air:airport    iata:LGA
_:NewYork    air:airport    iata:EWR
_:NewYork    air:map        <http://www.mapquest.com>
_:NewYork    air:weather
             <http://www.wunderground.com/US/NY/New_York.html >
```

For another use of blank nodes, consider the concept of "round trip". The airline ontology of Figure 25 has only flights connecting one city with another. Suppose we wanted to add some rudimentary trip planning concepts to the ontology, in particular the concept of round trip. One way to do this is to use blank nodes, for example

```
_:home    air:flight    _:away                                         (23)
_:away    air:flight    _:home
```

The resources in (23) are class-level statements from the perspective of the UML model in Figure 25. An application for a travel agent might want to introduce instance-level concepts based on this, for example a round trip between Sydney and Los Angeles

```
_:home    air:QF1    _:away                                            (24)
_:away    air:QF2    _:home
```

But this has the problem that the blank nodes *_:home* and *_:away* are used twice with different meanings. But since the names of blank nodes are arbitrary, in much the same way as variable names in programs are, the application is free to create new blank node names to keep the two different meanings separate. We could replace (24) with

```
_:home1    air:QF1    _:away1                                          (25)
_:away1    air:QF2    _:home1
```

so the ontology could contain both (23) and (25) without conflict.

Blank node names are thought of as being under the control of the ontology server, not the user. The server will generate identifiers for blank nodes whenever it needs to, either to display a fragment of a graph to a user, or to perform some process like mapping the graph from one representation to another. There is no guarantee that blank node identifiers will be twice generated the same. An identifier can not be used to search a graph. So that in (22), the user might have used the identifier *_NewYork* when entering the blank nodes, but the server might display the node as *_2251678* when that fragment is later viewed. But it would not make sense to search for a blank node identified by either *_NewYork* or *_2251678*.

10.3.5 XML Syntax

Triples are a convenient, human-understandable, shorthand way of representing RDF graphs, but the official (called *normative*) way of representing RDF graphs is using the XML syntax. The W3C has developed an XML representation of the various kinds of structures included in RDF, packaged as the markup language known as *rdf:RDF*. The full XML syntax for RDF is more than this text can deal with, but we give enough so that the reader can see how it works.

We discussed above how to represent in RDF that our airline ontology was expressed in the English language, ultimately in the triple (16). The XML representation would be

```
<rdf:Description rdf:about = "http://www.ontologies.org/airline">          (26)
        <dc:language>en</dc:language>
</rdf:Description>
```

The triple is represented as text marked up with the markup element <*rdf:Description*>. The subject is represented as the value of the *rdf:about* attribute of the *rdf:Description* markup element. The text marked up is the object "en" itself marked up with a markup element representing the predicate *dc:language*.

Since it is very common for a resource to be the subject of many triples, multiple predicates can be included in a single <*rdf:Description*> markup. For example, we can represent (the reverse of) (9) and (11) with the marked-up text

```
<rdf:Description                                                           (27)
    rdf:about = "http://www.ontologies.org/airline#QANTAS">
        <air:operates rdf:resource = "air:QF1" />
        <air:booking-site rdf:resource = "http://www.qantas.com" />
 </rdf:Description>
```

showing that that several triples with the same subject can be packaged into a single *rdf:Description*.

Examples (26) and (27) illustrate some facets of RDF syntax. In (26), the language object is represented by a text string enclosed in a markup tag, while in (27), the objects of operates and booking-site are both represented as values of the attribute *rdf:resource*. This is because the objects of the two statements in (27) are both themselves resources, while the object of the statement in (26) is a string of text, not a resource.

Literals and blank nodes can be represented in RDF/XML. The city name triple from (22) would be represented as

```
<rdf:Description rdf:nodeID = "NewYork">                                   (28)
    <air:name rdf:datatype = "xsd:string">New York</air:name>
        </rdf:Description>
```

The attribute *rdf:nodeID* is used instead of *rdf:about* to indicate a blank node name rather than a resource name. The type of the literal is indicated by the value of the *rdf:datatype* attribute.

The QNames are declared in a markup element enclosing the entire RDF document, for example

```
<?xml version="1.0"?>                                                    (29)
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xmlns:air="http://www.ontologies.org/airline ">
    …
</rdf:RDF>
```

The markup element *rdf:RDF* is the *RDF* markup element within the *rdf* namespace. The attribute *xmlns* (*XML namespace*) defines the QName prefixes.

There is much more to RDF, including several different notations. Also, XML is a very flexible language, so there are several alternative syntaxes even within RDF/XML. Most people would interact with RDF using a tool of some kind, so would be insulated by the software from the details of the syntax.

10.3.6 Advantages and Disadvantages of RDF for Ontology

We argued that a representation of the airlines ontology using bare XML was problematic because XML is a markup definition language and so a structure is needed which the markup can represent. UML provides such a structure, implemented in XMI, but UML was also insufficient. For one thing, a UML ontology does not fully specify individuals. For another, the granularity is too large so to use it. We need too much ontological commitment. Our discussion of RDF has by no means exhausted the facilities of RDF, but we can see that RDF corrects these problems. An individual can be specified as a resource named with a URI, and the granularity is much smaller, namely resources and RDF triples.

However, we have also lost something. The UML representation has a distinction between class and instance. We can specify associations at the class level, which means that all individuals must participate in the specified associations. Because of this we can ask questions at the class level and expect an answer which will be a definite collection of individuals.

This sort of distinction, derived from logic, is very widely used in computing. We have seen that institutional facts, which form most of the content of most ontologies, are themselves based on logical structures. More generally, languages have general grammatical structures which allow similar things to be said about a great variety of specific objects. RDF does not provide very good ways of distinguishing the general from the specific, and making sure that groups of similar objects are described in similar ways.

Further, a UML Class diagram is a definite engineered object, which has a responsible party, creation date, and boundaries. An RDF graph does not have boundaries, since it links namespaces with namespaces. Its (lack of) structure mirrors the (lack of) structure of the Web.

10.4 RDF Schema

The definition of RDF is actually bound up with a more general system called *RDF Schema* (*RDFS*), which provides many of the facilities we missed in the previous section.

UML has the model elements *class*, *subclass*, *attribute* and *association* which are used to give structure to an ontology, as illustrated in the airline ontology of Figure 25. RDFS has vocabulary elements *rdfs:Class* and *rdfs:subClassOf* corresponding to the first two, and *rdf:Property* subsuming *attribute* and *association*. These vocabulary elements are themselves resources, and are attached to application-specific resources with triples having as predicate the built-in resource *rdf:type*. Some of the class structure of the airline ontology is represented in RDFS as

air:Flight	rdf:type	rdfs:Class	(30)
air:Domestic	rdf:type	rdfs:Class	
air:Domestic	rdfs:subClassOf	air:Flight	
air:City	rdf:type	rdfs:Class	

Instances are associated with classes using also triples with predicate *rdf:type*

air:QF1	rdf:type	air:Flight	(31)
city:Sydney	rdf:type	air:City	
city:Los Angeles	rdf:type	air:City	

In Figure 25 the class *Flight* has two attributes, *depart_time* and *arrive_time*, and two associations with the class *City*, *origin* and *destination*. These are all represented in RDFS as *properties*

air:depart_time	rdf:type	rdf:Property	(32)
air:arrive_time	rdf:type	rdf:Property	
air:origin	rdf:type	rdf:Property	
air:destination	rdf:type	rdf:Property	

We now come to a major difference between UML and RDFS. In UML, attributes and associations are attached to class definitions. Part of the specification of the concept *Flight* is that it has values for *depart_time*, *arrive_time*, *origin* and *destination*. Conversely, only instances of the class *Flight* can have values for these attributes and associations.

This is not true in RDFS. The declaration that a resource is of *rdf:type rdfs:Class* means only that it can be the subject or object of an RDF triple, and nothing more. Similarly, the declaration that a resource is of *rdf:type rdf:Property* means only that it can be the predicate of an RDF triple, or the subject or object of triples with built-in predicates such as *rdf:type*, *rdf:subPropertyOf*, etc., nothing more.

So the declarations in (30), (31) and (32) constitute a syntactically complete RDFS ontology, with which one can write

air:QF1	air:origin	air:Sydney	(33)

as we would expect, but we can also write

air:QF1	air:origin	air:QF2	(34)

which is also a syntactically correct statement, although not semantically meaningful.

RDFS provides the built-in class *rdfs:Resource*, the class of all RDF resources. By definition every subject of a triple with predicate *rdf:type* and object *rdfs:Class* is a subclass of *rdfs:Resource*. RDFS thinks of an *rdf:Property* as if it were a UML association whose source and target are both the class *rdfs:Resource*. Since everything in an RDF model is a resource any resource can be the subject or object of a statement whose predicate is any property.

For many purposes this open policy is not enough. RDFS allows the subjects to be restricted to resources which are instances of a given class by the built-in property *rdfs:domain*, and the objects to be similarly restricted by the built-in property

rdfs:range. So we could specify that *air:origin* can be a property only of instances of *air:Flight* with the triple

air:origin rdfs:domain air:Flight (35)

or that only instances of *air:City* can be objects of the property *air:origin* with the triple

air:origin rdfs:range air:City (36)

Combining (35) and (36) gives something like the UML specification in Figure 25. There is, however, an important difference. In the UML specification, the association says that in order to be a valid instance, the source and target instances must be instances respectively of *Flight* and *City*. That is if *Flight QF1* has origin *Sydney*, then *QF1* must already be identified as an instance of *Flight*, and *Sydney* must already be identified as an instance of *City*. The UML specification functions like a type declaration in a programming language. It is a redundant declaration that allows type checking and detection of possible errors.

The corresponding RDFS triples do not make that assumption. It is possible, given the declaration (35) and (36) to infer from the triple

air:QF2 air:origin air:LosAngeles (37)

that *air:QF2* is an instance of the class *air:Flight*, and that *air:LosAngeles* is an instance of the class *air:City*. So range and domain specifications of properties in RDFS are weaker than association specifications in UML. They can't necessarily be used to detect type errors.

The open nature of RDFS goes quite deep. In (30) we declared both *Flight* and *Domestic* as classes before declaring that *Domestic* was a subclass of *Flight*. But *rdfs:subClassOf* is a property whose domain and range are both *rdfs:Class*. So it can be inferred that *Flight* and *Domestic* are both classes simply from the subclass declaration. The explicit definitions are redundant. (This is not to say that these kinds of redundant declarations are a bad idea, just that RDFS does not require them.)

Properties can have a range which is not a class, but a datatype. RDFS does not support its own datatypes but recognizes XML types, like RDF does. So the attributes *depart_time* and *arrive_time* from Figure 25 would be declared as properties, then given range declarations

air:depart_time rdfs:range xsd:time (38)
air:arrive_time rdfs:range xsd:time

RDFS also allows one property to be declared as a subproperty of another using the builtin predicate *rdfs:subPropertyOf*. So if we have the property *air:operates* with domain *air:Airline* and range *air:Flight*, we could declare a subproperty *air:international_operates,* the international flights operated by an airline, with

air:international_operates rdfs:type rdf:Property (39)
air:international_operates rdfs:subPropertyOf air:operates
air:international_operates rdfs:range air:International

A program would be entitled to infer that the domain of *air:international_operates* is the same as the domain of *air:operates*, namely *air:Airline*. It would also be entitled to infer that the range of *air:international_operates* is *air:Flight*, in addition to the class *air:International* declared in (39). Where two or more classes are declared in either the range or domain of a property, the inference is that the range or domain is the intersection of all. Since *air:International* is a subclass of *air:Flight*, the intersection of the two is *air:International*, so there is no conflict.

The ontology of Figure 25 with a few instances is given in Appendix B. You can see that graphic notations like UML are much easier to comprehend than textual notations like the RDFS triple for a complex structure like the airline ontology.

10.4.1 N-ary Relations and Association Classes

Properties in RDF are always binary. There are many applications whose conceptualizations are conveniently represented in systems like UML, ER or Object Role Modeling as either relations among more than two classes (*n-ary associations* in UML) or as attributes associated with instances of binary relations (*association classes* in UML). A classic example of the latter is the association between *Student* and *Course* called *enrolment*, which records which courses each student is enrolled in by an educational institution. Ultimately a student will be assigned a grade for each enrolment. The grade can't be represented as a property of either *Student* or *Course*. The other modeling systems in effect represent the grade as a property of *enrolment*, which is not allowed in RDFS. Figure 26 shows this conceptualization modeled as an association class in UML.

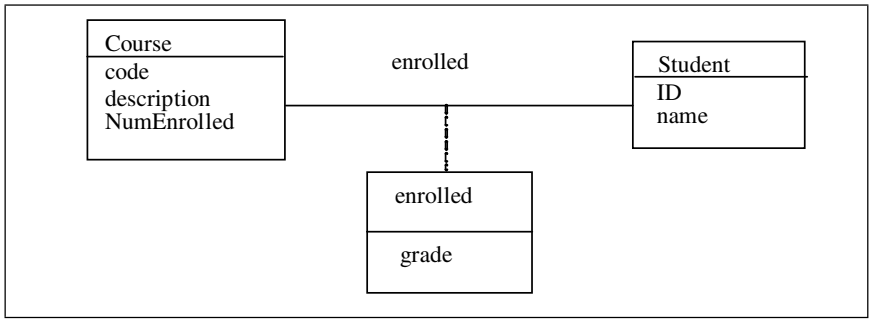


Figure 26. Conceptualization modeled as an association class in UML

An example typically modeled in UML as an n-ary association is the conceptualization that an Olympic competitor competing in an event in an Olympiad finishes in a position. A UML representation of this conceptualization could be as a quaternary association called *competes* among *Competitor*, *Olympiad*, *Event* and *Position*, as shown in Figure 27. Many dialects of ERA and ORM allow a similar construct.

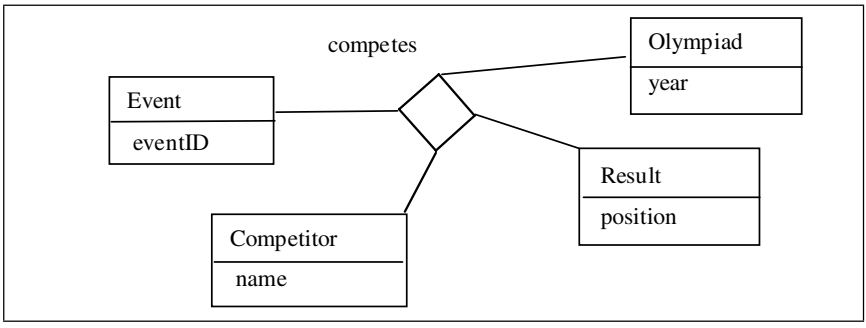


Figure 27. A quaternary association in UML

In a binary property system, it is necessary to model the conceptualization as a class rather than an association. In Figure 28, the concept *enrolled* is modeled as a class, with properties linking it to *Student*, *Course* and *Grade*. Similarly, in Figure 29, the concept *competes* is modeled as a class, with properties linking it to *Competitor*, *Olympiad*, *Event* and *Result*. The properties are not named in the diagrams to avoid clutter. They could be given names like *enrolmentStudent*, *enrolmentCourse* and *enrolmentGrade* in Figure 28, and *competesCompetitor*, *competesOlympiad*, *competesEvent* and *competesResult* in Figure 29.

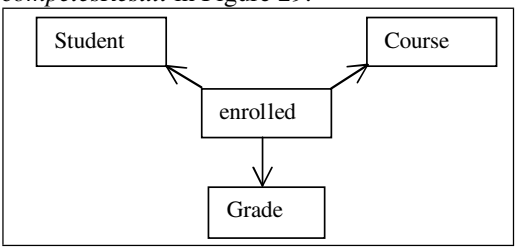


Figure 28. Association class of Figure 26 in a binary representation

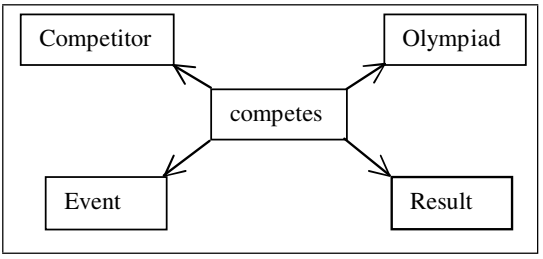


Figure 29. Quaternary association of Figure 27 in a binary representation

If we model the conceptualizations *enrolled* and *competes* as classes, we are left with the question of what their instances will look like. In a typical implementation of UML, instances of the association class of Figure 26 will be triples of instances of successively *Student*, *Course* and *Grade*, while instances of Figure 27 will be 4-tuples of instances of successively *Competitor*, *Olympiad*, *Event* and *Result*. We can't do this in RDFS because RDFS does not have any mechanisms to construct complex representations. What we can do is populate enrolled and competes with collections of differently named blank nodes, one for each linked collection of instances of the participating classes. So we might have triples

```
_enrolled000001   university:enrolled   university:student123456          (40)
_enrolled000001   university:enrolled   university:infs3101
_enrolled000001   university:enrolled   university:6^^xsd:integer
```

and so on.

A very common example of this kind is a property whose range is a dimensioned quantity. We want to say that the capacity of a container is some number of litres or that the distance between New York and London is so many miles. In RDFS we would model the range in both cases as a class populated by blank nodes with two further properties linking it to the value and the units, as in Figure 30.

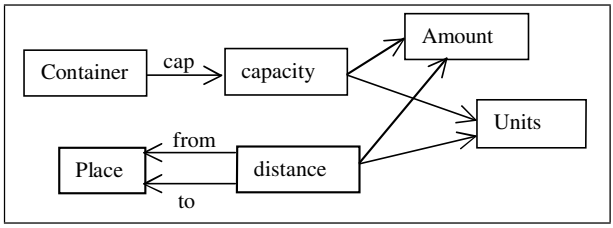


Figure 30. Properties whose range is a dimensioned quantity

Notice that capacity is modeled as a property *cap* whose range is a class *capacity* with two further properties whose ranges are respectively *Amount* and *Units*, while *distance* is modeled as a quaternary in the same way as Figure 29. We need to name the other properties whose ranges are *Amount* and *Units*. RDF provides a built-in globally defined property called *rdf:value* which would commonly be used to name the two properties whose range is *Amount*.

10.4.2 Advantages and Disadvantages of RDF Schema for Ontology

We have argued that RDF is a more convenient representation for an ontology than UML because it allows fully specified individuals and has an appropriate small granularity. But RDF has insufficient structure to support reliable reasoning systems.

RDF Schema is RDF with some additional built-in predicates which permit representation of much more structure than RDF. However, the structure represented is much less than represented in UML. For example, using RDFS we can say that there is a property *air:operates* whose domain is the class *air:Airline* and whose range is the class *air:Flight*. Given an instance, say *air:Qantas air:operates air:QF1*, we can infer that *air:QANTAS* is an instance of *air:Airline* and that *air:QF1* is an instance of *air:Flight*. But we can't say that every instance of *air:Flight* must participate in a property *air:operates* with some instance of *air:Airline*. Nor can we say that every instance of *air:Flight* must participate in a property *air:origin* with exactly one instance of *air:City*. These are structural constraints that can be represented on for example a UML Class diagram.

Further, we can't express the predicate enabling an instance of airline to be classified as one of the subclasses by looking at the flights it operates, or the predicate enabling an instance of flight to be classified as domestic or international by looking at its origin and destination cities. Additional structure like this can be expressed in an extension of RDFS called *Web Ontology Language*, or *OWL*, which is the subject of the next chapter.

Finally, an RDFS ontology is no more a definite engineered object than RDF. We need OWL for this as well.

10.5 Key Concepts

RDF is the elaboration of hyperlink as a **triple**, **resources** linking resources in **namespaces**. RDF allows **literals** and **blank nodes**. **RDFS** allows **class**, **subclass**, **subproperty**, **domain**, **range**. Neither support structure comparable to UML. We need OWL for that.

10.6 Exercise

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a. Represent a significant portion of the ontology in RDFS along the lines of Appendix B, using all the key concepts from the Chapter.

b. What concepts are you using that might be in more general namespaces? Suggest who might maintain these namespaces, if they do not already exist.

c. How would you represent the players and speech acts in RDF?

10.7 Further Reading

There is a large amount of material on the World-Wide Web Consortium web site http://www.w3.org/TR/#Recommendations including in particular [17].

11 Web Ontology Language OWL

We have seen in Chapter 10 the World-Wide Web Consortium's RDF language as a possible ontology representation language, and have noted some of its deficiencies. The W3C has addressed some of these deficiencies with the Web Ontology Language OWL, designed as a specialization of RDFS.

11.1 Why RDF Schema Is Not Enough

We have argued that RDF is a more convenient representation for an ontology than UML because it allows fully specified individuals and has an appropriate small granularity. But RDF has insufficient structure to support reliable reasoning systems.

RDF Schema is RDF with some additional built-in predicates which permit representation of much more structure than RDF. However, the structure represented is much less than represented in for example UML. Further, RDFS has no convenient mechanism to represent an ontology as an engineered object, with boundaries, versions, responsible authorities and the like.

Additional structure, including engineered objects, can be expressed in an extension of RDFS called *Web Ontology Language*, or *OWL*, also developed by the W3C.

11.2 Metamodel of OWL

11.2.1 RDFS

One way to understand the structure of OWL and its relationship to RDFS is by a metamodel as in Figure 31. This metamodel is a sort of ontology represented in a subset of UML called the Meta-Object Facility, or MOF. The MOF is used to define the structure of modeling systems, in particular UML. We will return to the MOF and some of its uses in Chapter 15.

To see how to read the metamodel, we will focus first on the part of it that represents the key structures of RDFS, namely the MOF classes *RDFSResource*, *RDFProperty* and *RDFSClass*, with their MOF subclasses and MOF associations. (We distinguish the classes, subclasses, instances, associations and attributes in a MOF model by the prefix MOF, to make it clear that the MOF constructs are different from the OWL constructs, even though they have the same form.) The Airline Ontology of Figure 25 represented in RDFS consists of a number of MOF instances of *RDFSClass* (*Airline*, *USAirline*, *EUAirline*, *InternationalAirline*, *Flight*, *Domestic*, *International*, *City*, *USCity*, *EUCity* and *OtherCity*), and a number of MOF instances of *RDFProperty* (*operates*, *origin*, *destination*, *depart_time*, *arrive_time*, *name* and *airport*). The subclass relationships in the Airline Ontology are all instances of the MOF association *RDFSsubClassOf*.

Each MOF instance of *RDFProperty* participates in a MOF instance of each of the MOF associations *RDFSdomain* and *RDFSRange*. The *domain* of *operates* is *Airline*, while the *domains* of *origin* and *destination* are *Flight*. The *range* of *operates* is *Flight*, while the *ranges* of *origin* and *destination* are both *City*. The other properties all have *range* a MOF instance of *RDFSClass* called *RDFSLiteral*, while the *domains* of *depart_time* and *arrive_time* are *Flight* and those of *name* and *airport* are *City*.

We know from Chapter 10 that every object appearing in an RDF triple is an instance of *RDFS:Resource*. The metamodel makes it clear that every MOF instance of *RDFProperty* and every MOF instance of *RDFSClass* are also MOF instances of *RDFSResource*. Every MOF instance of *RDFSResource* participates in the *RDFtype* MOF association with a MOF instance of *RDFSClass*. This says that all resources are the subject of an RDF triple whose predicate is *rdf:type* and whose object is an instance of *rdfs:Class*. Note that there is a MOF instance of *RDFSClass* called *rdfs:Resource*. Every MOF instance of the MOF class *RDFSResource* by default is of *RDFtype RDFSResource*.

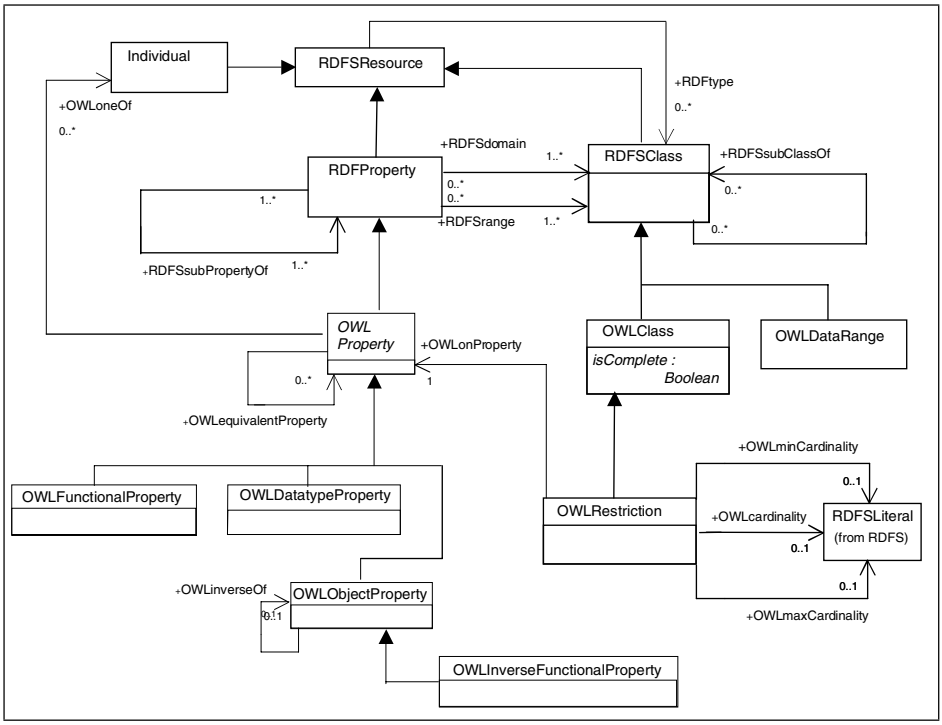


Figure 31. MOF metamodel for RDFS and OWL

11.2.2 OWL

OWL specializes RDFS. In particular, the key constructs of OWL are *Individual*, *OWLClass*, and *OWLProperty*, respectively specializations of *RDFSResource*, *RDFSClass* and *RDFProperty*. So an OWL *Individual* is an *RDFSResource*, an *OWLClass* is an *RDFSClass*, and an *OWLProperty* is an *RDFProperty*. The conventions of UML and hence the MOF mean that a MOF instance of *Individual* can participate in a MOF instance of *RDFtype* with a MOF instance of (the *OWLClass*

MOF subclass of) *RDFSClass*, and so on with the other MOF associations of the RDFS fragment of the metamodel.

11.3 OWL Properties

OWL provides a number of different kinds of property. A MOF instance of *OWLobjectProperty* has *domain* and *range* both MOF instances of *OWLClass*, while a MOF instance of *OWLDatatypeProperty* has *domain* a MOF instance of *OWLClass* and *range* a instance of another subclass of *RDFSClass* called *OWLDataRange*. An instance of *OWLDataRange* is either a class whose type is one of the rdfs:Literal types or an enumeration of literals constructed using the owl:OneOf constructor described below.

A property can be functional (*OWLFunctionalProperty*) if an instance of the domain is associated with at most one instance of the range. A property can be inverse functional (*OWLInverseFunctionalProperty*) if an instance of the range is associated with at most one instance of the domain. Object properties and datatype properties are disjoint, but either kind of property can be functional. An instance of *OWLObjectProperty* can be associated with another instance which is its inverse (*OWLinverseOf*). Two properties can be declared to be equivalent (synonymous) (*OWLequivalentProperty*).

Note that if the inverse of a functional property is defined, it is inverse functional.

In the Airline Ontology of Figure 25, the properties *operates*, *origin* and *destination* are all object properties, while *depart_time*, *arrive_time*, *name* and *airport* are all datatype properties. *Operates* is inverse functional, while the other properties are all functional.

Properties in RDF and hence OWL are directional, thought of as from the domain to the range. In the Entity-Relationship model or UML attributes are similarly directional, but relationships (UML associations) have no explicit direction. Instead relationships have roles, which often are not explicitly named. Generally, the relationship name has a conventional reading in one direction or the other. So the *operates* property might be represented by a relationship called *operates*, with the role attached to *Airline* called *operator* and that attached to *Flight* called *operated*, for example. The conventional reading in English would be that *Airline operates Flight*, so we would interpret *Airline* as the domain and *Flight* as the range.

If you are used to the ER model and the cardinality ratio notation, a functional property is many-to-one, while an inverse functional property is one-to-many. If you are used to the min-max notation of ER or to UML, in a functional property each instance of the domain is linked at most once to an instance of the range, while in an inverse functional property each instance of the range is linked at most once to an instance of the domain.

11.3.1 Properties Are Global

A feature of OWL is that properties have a default domain and range of a MOF instance of OWLClass called *Thing*, the class whose (OWL) instances are the MOF instances of *Individual*. To reduce confusion, we have in this text distinguished instances of a MOF class in Figure 31 from instances in OWL of an OWL class. An OWL class, say *Airline*, is a MOF instance of the MOF class *OWLClass*. A particular

airline, say "QANTAS", is an OWL individual (a MOF instance of the MOF class *Individual*), and an OWL instance of the OWL class *Airline*. "QANTAS" gets to be an OWL instance because there is a MOF instance of the MOF association *RDFtype* linking the individual "QANTAS" with a MOF instance of *OWLClass*, namely *Airline*. *Thing* is a particular MOF instance of *OWLClass*, and every MOF instance of *Individual* is linked to *Thing* by a MOF instance of *RDFtype*.

That last paragraph is pretty dense. It may become clearer if we look at the database represented by the MOF model, using the standard mapping of UML class diagram to a relational database scheme. The MOF association *RDFtype* is represented by a table with two columns, one containing MOF instances of *Individual* and the other MOF instances of *OWLClass*. The examples are shown in Figure 32 as a population of this database. All OWL instances of OWL classes are represented by rows in this table.

Individual	**OWLClass**
QANTAS	Airline
QANTAS	Thing

Figure 32. A population of the *RDFtype* table in a database derived from Figure 31

That a property in OWL is by default defined on all individuals is a powerful feature with many uses, including:

- An individual can have properties without being an instance of any class (other than of course *Thing*). A reasoning program can conclude from the properties of an individual that the individual is an instance of a particular class (the properties are rigid for that class). In the Airlines ontology, we may have an individual called "QF12", which is linked to the *City* "Brisbane" by the property *origin* and to the *City* "Singapore" by the property *destination*. From this the reasoner can conclude that "QF12" is an instance of the class *Flight*, and further an instance of the subclass *International*.
- An ontology can include properties that apply to any individual, regardless of which class it is an instance of. For example, every individual can have a property *dateCreated* (a datatype property with range the literal date) or *owner* (an object property with range a class of stakeholders).
- Generic properties can be defined which can be used in many classes. For example, complex classes are tied together by a *partOf* property, as we saw in Chapter 5. So the ontology may include one class *Vehicle* with a parts class *VehicleParts*, and another class *DesignDocument* with a parts class *DesignDocumentParts*, and use the same *partOf* property to link *VehicleParts* to *Vehicles*, simpler *VehicleParts* to more complex *VehicleParts*, *DesignDocumentParts* to *DesignDocuments*, and simpler *DesignDocumentParts* to more complex *DesignDocumentParts*. (The part-of property is actually fairly complicated, as we will see in Chapter 12, but it is a good example of a generic property.)

11.3.2 Mathematical Characteristics of Property

Of course it does not make semantic sense in all cases for a property to be generic. The property *mass* makes sense only for physical objects. The property *name* makes sense only for identified objects. The property *maritalStatus* makes sense only for people. These three properties have very specific possible values. There are properties which are very widely applicable (e.g. *createdBy*), but whose value is specific, in this case

always by a person. It is common for ontologies to have a number of very general classes like *PhysicalObject*, *IdentifiedObject* or *Person*. OWL allows a declaration of a range or domain for a property, so that for example the property *mass* can be restricted to instances of *PhysicalObject* by declaring its *domain* to be *PhysicalObject*, or the property *createdBy* can be restricted to have its *range* restricted to instances of *Person*. The declarations *RDFSdomain* and *RDFSrange* are used for this purpose.

Besides being functional or inverse functional, a property can have other mathematical characteristics. In the language of mathematics:

- A property can be total or partial. A *total* property has at least one value for every instance of its domain, while a *partial* property can be undefined for some instances of its domain.
- It can be a *surjection* (sometimes called *onto*). A property is a surjection if every instance of its range is the value of the property for at least one instance of its domain. (A surjection can be partial.)
- It can be an *injection* (sometimes called *one-to-one*). A property is an injection if each instance of the domain is linked to a different instance of the range. (An injection can be partial.)
- An identifying property must be a total injection, whose inverse is also an injection. An instance of the domain is identified by an instance of the range. So every instance of the domain must be linked to exactly one instance of the range, and each instance of the range to at most one instance of the domain.

  A property whose domain and range are the same can be
- *symmetric* if it reads in both directions
- *transitive* if individual A is linked to individual B by the property, and individual B is linked to individual C, then A is linked to C by the same property.

For example, in the Airlines ontology, *origin* and *destination* are both total, since every instance of *Flight* must start somewhere and end somewhere. *Origin* and *destination* are both functional, since an instance of *Flight* can start in only one city and end in only one city. But we can have instances of *City* which are not served by any instances of *Flight*, so *origin* and *destination* are not surjections. On the other hand, the property *operates* is a surjection, since every instance of *Flight* must be operated by some airline. But we can have instances of *Airline* which don't have any flights, so *operates* is partial. The property *name* identifies instances of *City*, so is total, functional and its inverse is functional.

None of the properties in the Airlines ontology have domain and range the same, so none of them are either symmetric or transitive. We can see examples of these in an ontology of family relation properties on the class *Person*. *Sibling* is symmetric, since if Robert is a sibling of Joan, Joan is a sibling of Robert. But *brother* is not, since although Robert is a brother of Joan, Joan is not a brother of Robert, but a sister. *Ancestor* is transitive, since if Raymond is an ancestor of Robert and James is an ancestor of Raymond, James is an ancestor of Robert. But *parent* is not. If Raymond is a parent of Robert, and James is a parent of Raymond, James is not a parent of Robert, but a grandparent.

OWL allows properties to be declared transitive or symmetric, in the same way it allows properties to be declared functional or inverse functional. An injective property is both functional and inverse functional. The other mathematical characteristics total, partial and surjection are declared less directly.

11.3.3 Local Cardinality Restrictions

Total or partial properties and properties being surjective or not have to do with the number of instances of the property containing particular instances of the domain or particular instances of the range. If the property is total, then each instance of the domain must be linked to at least one instance of the range by the property. If the property is surjective, then each instance of the range must be linked to at least one instance of the domain by the property.

In ER modeling, a total property is a *mandatory* participation in that role, as is a surjective property. In ER modeling using the min/max notation, or in UML, the minimum cardinality is 1 for a total or surjective property and 0 for a partial property or a property that is not surjective.

We have seen above that the min/max notation allows specification of a property as functional or inverse functional (max cardinality 1). The min/max notation allows a much richer specification of cardinalities (multiplicities in UML) than this. The maximum cardinality can be indefinite (* in UML), which says that any number of instances (greater than or equal to the min cardinality) can participate in the property. In the Airline ontology, a particular instance of *Airline* can be linked to any number of instances of *Flight* by the *operates* property.

Further, the min or max cardinality can be a definite number greater than 1. For example, if *Person* is the domain of a *parent* property, then we could specify the min and max cardinality to be 2 for the domain (every person has exactly two parents). If a student is allowed to sit an examination a second time (say if they were ill enough in the first sitting for it to have affected their performance), then we could set the max cardinality of the range of the property *grade* to be 2. The min cardinality of the range would be 0, since the student may have sat the examination only once, or perhaps not at all.

OWL uses a min/max cardinality restriction, but it is different from ER or UML.

Recall that a property in OWL is declared by default to apply to any individual. The specializations of property (functional, inverse functional, symmetric, transitive) are global declarations, so apply to any instance. The optional declarations *RDFSdomain* and *RDFSrange* are also global. For example if the property mass is declared to have its domain *PhysicalObject*, then it can be applied only to instances of that class (or of course its subclasses).

But a property often behaves differently for different specific subclasses. We may have a class *Student*, with subclasses *Enrolled*, *Active* and *Graduated*. *Enrolled* students are accepted into the institution but have not yet signed up for any courses in the current semester. *Active* students have signed up for courses in the current semester. *Graduated* students are students who have completed their study. Assume that the three subclasses are disjoint. The property *gradePointAverage* would be total on the subclass *Graduated* (every graduated student must have a grade point average), but partial on the subclasses *Enrolled* or *Active* (some of these students may not yet have completed any courses).

Further assume that every student must have declared one or possibly two majors. The property *major* is total for *Student*. Now assume there are two subclasses of *Student*, *SingleMajor* and *DoubleMajor*. An instance of *SingleMajor* is linked to exactly one instance of *Major*, while an instance of *DoubleMajor* is linked to exactly two. So the property *major* has a maximum cardinality 1 on the domain *SingleMajor* but a maximum cardinality 2 on the domain *DoubleMajor*.

The same sort of variation can take place on the range of a property. Assume a company leases an office building. The company has employees and the building has a collection of rooms. Consider a property *assignedTo* whose domain is *Employee* and whose range is *Room*. A room can be vacant, so the minimum cardinality of the range of *assignedTo* is 0. Assume the class *Room* has two subclasses, *Vacant* and *Allocated*. The maximum cardinality of the range of *assignedTo* on the subclass *Allocated* is 1, but on the subclass *Vacant* the maximum cardinality is 0.

11.3.4 OWL Cardinality Restrictions on Property Domains

OWL defines restrictions in a very different way from ER or UML. As you can see from Figure 31, an *OWLRestriction* is a special kind of *OWLClass*, which is specific to a particular *OWLProperty* via the MOF association *OWLonProperty*. Every restriction class is a subclass of the domain of its property. We are always going to apply a restriction to some subclass S of the domain of the property. So the general procedure is to define a restriction class then declare S to be a subclass of the restriction class.

There are several flavours of *OWLRestriction.* The flavours of restriction shown in Figure 31 are represented as MOF associations whose target is a literal. The flavours shown in Figure 31 are the cardinality restrictions *OWLminCardinality*, *OWLmaxCardinality* and *OWLcardinality*.

So for example if we have a property P we can have a restriction on property P where the minimum cardinality is 1. This restriction is the subclass of the domain of P whose instances are associated with at least one instance of the range of P. In mathematical terms, this restriction is the subset of the domain of P on which P is total. If we want to say that P is total when applied to a particular class, we declare that class a subclass of the restriction.

For example, suppose we want to declare the property *gradePointAverage* to be total on the subclass *Graduated* of *Student*. The restriction *OWLminCardinality* = 1 on *gradePointAverage* is the subclass of *Student* which have a grade point average. We declare *Graduated* to be a subclass of the restriction class. Similarly, the requirement that all students must have nominated a major can be expressed as a restriction *OWLminCardinality* = 1 on the property *major*, and a declaration that the domain of *major*, namely *Student*, is a subclass of the restriction class.

To get the subclass *SingleMajor* of *Student*, we create the restriction *OWLmaxCardinality* = 1 on property *major*. This restriction class consists of the instances of *Student* who have one major. We declare *SingleMajor* to be a subclass of the restriction class. The minimum cardinality restriction on *major* when applied to its domain *Student* insures that each student has at least one major. In this case, the restriction class involving maximum cardinality is identical with the named subclass, but the same is not true of the restriction *minCardinality* = 1 on *gradePointAverage*. There may be students other than *Graduated* who have a grade point average.

The rules of OWL permit only one restriction in a restriction class. If we wanted to declare that a property was both functional and total when applied to a class, we would have to make two restriction classes, one for *minCardinality* = 1 and one for *maxCardinality* = 1. OWL provides a shorthand way to do this with a declaration *OWLcardinality*, which says that the minimum and maximum cardinalities are the same and are equal to the indicated literal. So if we allowed students to declare majors late in their studies, the property *major* would no longer be total on its domain. We could declare a restriction class *OWLcardinality* = 1 on *major*, and declare *SingleMajor*

to be a subclass of that. In this case *SingleMajor* would be the subclass of *Student* who have declared exactly one major.

A combined cardinality restriction is particularly convenient for *DoubleMajor*, since the minimum cardinality is strengthened to 2 so we need both a minimum and maximum cardinality restriction even if *major* is total on *Student*. Instead, we could declare a restriction class *OWLcardinality* = 2 on *major*, and declare *DoubleMajor* to be a subclass.

Restriction classes are not necessarily named, because they are often used only once in the definition of a property. The class *OWLminCardinality* = 1 on *major* is used when the property *major* is defined on its domain *Student*. The restriction automatically applies to all subclasses of *Student*, and by the definition of domain no individual not an instance of *Student* can have a value for the property *major*. Similarly, the *OWLmaxCardinality* and *OWLcardinality* restrictions on *major* need be declared only once.

However, it is sometimes convenient to name restriction classes, and OWL provides a mechanism to do so. There is a declaration *OWLequivalentClass* which states that two classes are equivalent. In the case of single and double major students, we could declare that *SingleMajor* is an *OWLequivalentClass* to the restriction class *OWLmaxCardinality* = 1 on *major*, and similarly for *DoubleMajor*. This is stronger that declaring that *SingleMajor* is a subclass of the restriction class, since it excludes the possibility that there are instances of the restriction class which are not instances of *SingleMajor*.

We have noted that there may be instances of the restriction class *OWLminCardinality* = 1 on *gradePointAverage* which are not instances of *Graduated*. Any student who has completed at least one course will be an instance of the restriction class. So in this case it might be useful to name the restriction class, using a declaration like *StudentsWithGradePointAverage OWLequivalentClass* (*OWLminCardinality* = 1 on *gradePointAverage)*.

11.3.5 OWL Restrictions Involving a Property Range

OWL has mechanisms to declare restrictions involving the range of a property. These are the declarations *OWLhasValue*, *OWLsomeValuesFrom* and *OWLallValuesFrom* (not shown in Figure 31). But these restriction classes, like all restriction classes, define subclasses of the domain of the property.

OWLhasValue designates the subclass of the domain of a property containing the instances linked to a particular instance of *RDFSResource*. For example, the restriction *OWLhasValue* "Colomb" on the generic property *owned* designates the subclass of *Thing* owned by the person Colomb. The restriction *OWLhasValue* "Computer Science" on the property *major* designates the subclass of the domain of *major* which is linked to the individual "Computer Science" by the property. Remember that the restriction classes are anonymous, so if we wanted a named class *ColombsStuff*, we would have to declare it equivalent to the restriction *OWLhasValue* "Colomb" on *owned*, or if we wanted a class *ComputerScienceStudent* we would have to declare it equivalent to the restriction *OWLhasValue* "Computer Science" on the property *major*.

OWLsomeValuesFrom designates the subclass of the domain of a property containing the instances one of whose links is an instance of a designated class. Suppose we have a class *ScienceMajors* consisting of the subclass of *Majors* which are one of the sciences, including Mathematics, Physics, Chemistry, Geology, Botany,

Zoology and so on. The restriction *OWLsomeValuesFrom ScienceMajors* on *major* designates those instances of the domain of *major*, the class *Student*, one of whose majors is a science. Again, the restriction class is anonymous, so if we want to name it we must declare a name using *OWLequivalentClass*.

OWLallValuesFrom designates the subclass of the domain of a property containing the instances all of whose links are instances of a designated class. The restriction *OWLallValuesFrom ScienceMajors* on *major* designates those instances of the domain of *major*, the class *Student*, each of whose majors is a science. The *OWLsomeValuesFrom ScienceMajors* on *major* restriction includes students with single science majors and double majors like Mathematics and Philosophy or Mathematics and Physics. The restriction *OWLallValuesFrom ScienceMajors* on *major* also includes students with single science majors and double majors like Mathematics and Physics, but excludes double majors like Mathematics and Philosophy.

If we want several restrictions on a property to apply to a subclass of the domain of a property, we define each restriction separately, then declare the subclass to be a subclass of each of the restriction classes.

We can use *OWLsomeValuesFrom* to get some cardinality restrictions on the range of a property when applied to a particular class. A cardinality restriction applies only to the domain of a property, but if we declare an inverse property the domain of the property is the range of the inverse.

In particular, we can declare a property to be surjective on a subclass of its domain. Of course, if a property is surjective on a subclass of its domain, it is surjective on the domain. Assume we have a property P whose domain is D and whose range is R. For example, take the property *assignedTo* whose domain is *Employee* and whose range is *Room*. We want P to be surjective on some subclass D' of D; that is every instance of R is linked to an instance of D' by P. For example, suppose we have a subclass *KeyEmployee* of *Employee*. We want each *KeyEmployee* to be assigned to a room such that each room has at least one *KeyEmployee* assigned.

To do this in OWL requires several declarations, following the diagram in Figure 33. We declare PI to be the inverse of P. In our example, we could declare *assigned* with domain *Room* and range *Employee* as the inverse of *assignedTo*. We then declare a restriction Res = *someValuesFrom* D' on PI. In our example, we declare the restriction *someValuesFrom KeyEmployee* on *assigned*. That is, given an instance r of Res there is an instance d' of D' such that P(d') = r. In the example, for each instance of *Room* there is a least one instance of *KeyEmployee assignedTo* the instance of *Room*. Then we declare R a subclass of Res. In the example, we declare *Room* a subclass of *Restriction*. *Room* thus satisfies the restriction predicate, so *assignedTo* is surjective when applied to *KeyEmployee*.

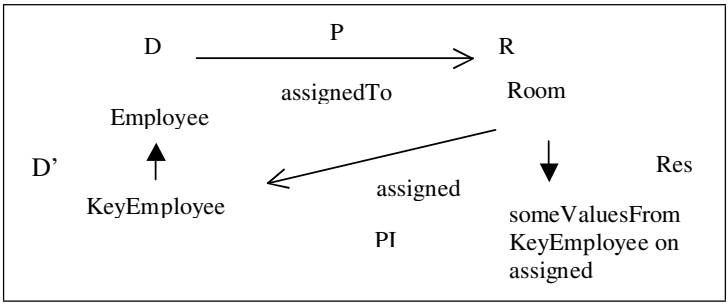


Figure 33. Diagram of how to declare a surjective property in OWL

Note that the behaviour of P is unconstrained on D – D'. In the example, the restriction does not specify whether the property *assignedTo* has values for instances of *Employee* which are not also instances of *KeyEmployee*.

So even though OWL does not directly support cardinality restrictions on the range of properties when applied to particular subsets of their domain, we can get the effect of a restriction minimum cardinality = 1 on the range. However, it appears to be much more difficult to get other cardinality restrictions on the range.

11.3.6 Identifiers, Single and Composite

We have already established that for a property I whose range is N to be an identifier for a subclass S of its domain it must be total, functional and its inverse also functional. Following the method of the previous section, in OWL we declare S to be a subclass of the restriction class on I *maxCardinality* = 1. We then declare IN as *inverseOf* I, define a restriction class on IN *maxCardinality* = *1*, and declare N a subclass of that restriction class.

Note that we have not declared that I is an identifier for S. We have only declared the constraints necessary for I to qualify as an identifier. OWL does not have the concept of identifier. But see Chapter 12, which suggests a way to augment OWL to include identifiers.

We often encounter objects in a conceptualization which have composite identifiers. For example, the class *enrolled* of Figure 28 has a composite identifier consisting of an instance of *Student* paired with an instance of *Course*. Unfortunately, it does not appear to be possible in OWL to define the constraints necessary for composite identifiers.

11.4 Names

A common assumption in computing applications is that within a namespace the same name always refers to the same object, and that different names always refer to different objects (the *unique names assumption*). As a consequence, given a set of names, one can count the names and infer that the names refer to that number of objects.

Names in OWL do not by default satisfy the unique name assumption. The same name always refers to the same object, but a given object may be referred to by several different names. Therefore counting a set of names does not warrant the inference that the set refers to that number of objects. Names, however, are conceptually constants, not variables.

OWL provides features to discipline names. The unique name assumption can be declared to apply to a set of names (*allDifferent*). One name can be declared to refer to the same object as another (*sameIndividualAs*). One name can be declared to refer to something different from that referred to by any of a set of names (*differentFrom*).

Classes and properties are by default different, but two classes or two properties can be declared to be equivalent (*equivalentClass*, *equivalentProperty*).

11.4.1 Closed World Assumption

Names in OWL not satisfying the unique names assumption is a consequence of a deeper principle: that OWL does not support the closed world assumption.

Database systems generally permit negative queries. For example, if a university student records systems has a table *Student* containing a column *Name*, then we can make a query that tells us that a particular person is not a student. In SQL such a query could be

```
SELECT true FROM Student WHERE                                    (41)
    NOT EXISTS SELECT Name FROM Student WHERE Name = Colomb
```

What this says it that the predicate "Colomb is not a student" is true by virtue of there being no student of that name in the *Student* table. Contrast that with an inference drawn by a predicate calculus theorem prover from the formula

```
ForAll Name (notstudent(Name) or                                  (42)
    Exists X (student(X) and name(X, Name)))
```

In the first case, the conclusion is drawn from the failure of the database to find the information. In the second case, the theorem prover can not conclude that there is no student with the given name. The truth of an existentially quantified formula is not disproved by the failure to find any concrete instances. For a somewhat macabre example, consider the predicate

```
ForAll Name (immortal(Name) or                                    (43)
    Exists D (date(D) and personDied(Name, D)))
```

We can't conclude that a presently living person is immortal only on the evidence that there is no date on which that person died. We are pretty sure that such a date will come eventually.

Consider another alternative to (41). Instead of whether a person is a student, the query is about whether the person has a mobile phone. Assume that a student is asked to provide a mobile phone number when they update their student details, but that the University has no way of verifying whether the information entered is correct. The query

```
SELECT true FROM Student WHERE                                    (44)
    NOT EXISTS SELECT Name FROM Student
        WHERE Name = Colomb AND MobilePhoneNo NOT NULL
```

would not be taken as a warrant to conclude that the student Colomb does not own a mobile phone.

The difference between (41) and (44) is that in the former case, presence of the student in the Student table is considered by the university as definitive evidence of the institutional fact "Person Colomb is a student". Therefore if the person is not listed in the student table, they are not students no matter whether they attend classes or learn things. In the case of (44), the query is simply an indication that we don't know whether the person has a mobile telephone.

What makes (41) a valid inference is the assumption that the contents of the Student table are definitive as to whether a person is a student. This is called the closed world assumption. It often applies in information systems because information systems are generally constructed to keep the records of an organization that makes speech acts and therefore creates institutional facts.

OWL is a Web language, built on RDF. As we saw in Chapter 10, it is always possible to bring in more information. The world is open, not closed. This is why the reasoning systems built on OWL require definite rules to conclude negations. For

example, just because an object does not appear in the Periodic Table does not mean it is not an element, only that it might be an undiscovered element.

This characteristic of OWL makes the interpretation of cardinality constraints somewhat strange. For example, consider a property *P* with domain *D* and range *R* declared to be total on *D* using *cardinality on P = 1*. If we have *d* an instance of *D*, we might expect that we would have a corresponding instance *r* of *R* such that *d* is linked to *r* by *P*. Suppose we can't find such an *r*. In a database system, the integrity constraint would signal a violation. In OWL, it doesn't. OWL assumes that there is an appropriate r somewhere, but we just don't know about it. Similarly, the cardinality constraint leads to expect that there is only one *r* linked to an instance of *d*. Suppose we find two, *r* and *r'*. In a database system, an integrity constraint violation would be signaled. But not in OWL. The reasoner would conclude that *r* and *r'* were the same (*r sameAs r'*). It would take a contrary declaration *r differentFrom r'* to constitute a violation.

There are of course cases where the ontology is definitive. The possible game states in Tic-Tac-Toe can be enumerated, so we can conclude that if a state is not a winning state it must be either a drawn state or a losing state.

11.5 Class Descriptions

So far we have declared classes and subclasses (in the terminology of Chapter 6), and classes which are subclasses of several others (called multiple inheritance in the object-oriented world). Given an instance of a class like this we can say something about it. For example, given that a person is an instance of *KeyEmployee* we can conclude that the person is also an instance of its superclass *Employee*. Given that a person is an instance of the class *PresidentOfUnitedStates* we can conclude that the person is an instance of the class *NativeBornUSCitizen*, and also an instance of the class *PersonOver35YearsOld*, because we can represent the class *PresidentOfUnitedStates* as a subclass of both of the other two classes.

This sort of class description is called necessary conditions. But we can't reason the other way. Given that a person is an instance of *Employee* we can't conclude that the person is an instance of *KeyEmployee*. Of course, we wouldn't expect to since we would expect that there was something that distinguishes instances of *KeyEmployee* from other instances of *Employee*. But we also can't conclude from the fact that a person is an instance both of *NativeBornUSCitizen* and *PersonOver35YearsOld* that that person is an instance of *PresidentOfUnitedStates*.

OWL has a number of ways of giving a class a description which is both necessary and sufficient, called *complete* in OWL. Note the *isComplete* attribute of the MOF class *OWLClass* in the metamodel of Figure 31.

11.5.1 OneOf

One necessary and sufficient class description is to enumerate the class instances. For example, the class *DayOfTheWeek* consists in English of the names {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} and no others. This kind of definition is called an *enumeration*, and is expressed in OWL by declaring the class as *OneOf* a set of identifiers.

11.5.2 Intersection

Class intersection is a stronger description than multiple inheritance. The class *ComputerScienceGraduate* can be described as the intersection of the classes *Graduate* and *ComputerScienceMajor*, both of which are subclasses of *Student*. Unlike the situation with the President of the United States, given that we know that a person is both an instance of *Graduate* and an instance of *ComputerScienceMajor* we can conclude that the person is an instance of *ComputerScienceGraduate*. This is an example of a defined class in the terminology of Chapter 6. OWL has the construct *intersectionOf* to make these descriptions.

11.5.3 Other Boolean Combinations

Besides intersectionOf, OWL allows classes to be described as the union of two classes (*unionOf*) and a class to be described as the complement of a class (*complementOf*). Union is pretty straightforward but complement is a little tricky.

For example, we could define a class *NotStudent* as *complementOf* (*Student*). The question is what constitutes an instance of *NotStudent*? We need to know what the universe is to know this. In OWL, every individual is an instance of the class *Thing*, so the class *NotStudent* is every instance of *Thing* which is not an instance of *Student*. But this is still pretty indeterminate.

It is usual to use *complementOf* only in conjunction with *intersectionOf*. So we might define *ActiveStudent* as *intersectionOf*(*Student*, *complementOf*(*Graduate*)).

Note that besides the built-in class *Thing*, OWL has also built in the empty class *Nothing*, defined as *complementOf*(*Thing*).

11.5.4 DisjointWith

OWL allows the description that two classes are disjoint, using the descriptor *disjointWith*. For this to make semantic sense, the two classes should be subclasses of the same superclass, so the fact that the two subclasses are disjoint is meaningful. Of course, since all classes in OWL are subclasses of *Thing*, it is possible to declare any two classes to be disjoint. We could make the description *Human disjointWith Brick*, for example.

In the Tic-Tac-Toe example in Figure 8 there are five subclasses of *Row*, namely *FilledRow*, *Possible*, *Xs*, *Os* and *Mixed*. Although they are grouped into two collections by the graphical notation, the notation used does not imply that the subclasses are disjoint. Semantically, *FilledRow* and *Possible* are disjoint, as are *Xs*, *Os* and *Mixed*. But any instance of *FilledRow* could be also an instance of *Xs*, or *Os*, or *Mixed*, and so also for any instance of *Possible*. We would need to augment the diagram with *disjointWith* descriptions to capture the full semantics.

Note that *disjointWith* and *complementOf* can be used to represent some aspects of the ontology to which the closed world assumption should apply.

11.6 Defined Subclasses for the Airlines Ontology

The airline ontology of Figure 25 has a subtype structure. *City* has the subclasses *USCity*, *EUCity* and *OtherCity*. *Flight* has two, *Domestic* and *International*, while *Airline* has three, *USAirline*, *EUAirline* and *InternationalAirline*. In the RDF representation, all these were modeled as declared subclasses.

But in the specification of the ontology, *Domestic* was described as a flight in which both origin and destination were either US cities or EU cities and *International* as the rest. *USAirline* was described as an airline all of whose flights were between US cities, *EUAirline* as an airline all of whose flights were between EU cities, and *InternationalAirline* as all others.

One of the deficiencies of RDFS discussed at the end of Chapter 10 is that there is no mechanism to define these subclasses. Representing these as defined subclasses is higher in clarity than representing them as declared subclasses. It is also higher in coherence, as given a change to the extent of *Flight* we can infer changes to the extents of its subclasses, and also infer changes to the extents of the subclasses of *Airline*. OWL permits us to do this, which is one of the reasons to prefer OWL over RDFS as an ontology representation language.

We show how we can define the subclasses of *Flight* and *Airline* using the facilities of OWL. To be concrete, we will employ the XML syntax.

The first task is to define the subset of flights whose origin is a US city. We do this using the restriction *allValuesFrom*.

```
<owl:Restriction>                                                    (45)
    <owl:onProperty rdf:Resource="air:origin"/>
        <owl:allValuesFrom rdf:Resource="air:USCity"/>
</owl:Restriction>
```

Restrictions can be anonymous, but in this case we want to increase the extendibility of the ontology as well as the clarity and coherence, so we will follow Debenham's advice and name every semantically meaningful structure for possible reuse. We do this using *equivalentClass*.

```
<owl:Class rdf:ID="USOriginFlights">                                 (46)
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:Resource="air:origin"/>
                <owl:allValuesFrom rdf:Resource="air:USCity"/>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

Similarly, we need definitions of flights whose destination is a US city

```
<owl:Class rdf:ID="USDestFlights">                                   (47)
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:Resource="air:destination"/>
                <owl:allValuesFrom rdf:Resource="air:USCity"/>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

whose origin is an EU city

```
<owl:Class rdf:ID="EUOriginFlights">                                    (48)
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:Resource="air:origin"/>
                <owl:allValuesFrom rdf:Resource="air:EUCity"/>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

and whose destination is an EU city

```
<owl:Class rdf:ID="EUDestFlights">                                      (49)
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:Resource="air:destination"/>
                <owl:allValuesFrom rdf:Resource="air:EUCity"/>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

With these elements we can complete the definitions using Boolean combinations. First, the flights whose origin and destination are both US cities. (Note that *intersectionOf* and *unionOf* both apply to an arbitrary number of classes. The relevant classes are packaged using the RDF *parseType* "Collection".)

```
<owl:Class rdf:ID="USFlight">                                           (50)
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#USOriginFlight">
        <owl:Class rdf:about="#USDestFlight">
    </owl: intersectionOf>
</owl:Class>
```

and whose origin and destination are both EU cities.

```
<owl:Class rdf:ID="EUFlight">                                           (51)
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#EUOriginFlight">
        <owl:Class rdf:about="#EUDestFlight">
    </owl: intersectionOf >
</owl:Class>
```

We can now define *Domestic* using *unionOf*

```
<owl:Class rdf:ID="Domestic">                                           (52)
    <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#USFlight">
        <owl:Class rdf:about="#EUFlight">
    </owl: intersectionOf>
</owl:Class>
```

and since international flights are flights which are not domestic, we complete the subclasses of *Flight* using *intersectionOf* and *complementOf*

```
<owl:Class rdf:ID="International">                                      (53)
    <owl:intersectionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Flight">
        <owl:complementOf rdf:resource="#Domestic">
    </owl: intersectionOf >
</owl:Class>
```

We define the subclasses of *Airline* in a similar way. First, *USAirline*

```
<owl:Class rdf:ID="USAirline">                                          (54)
    <owl:equivalentClass>
```

```
        <owl:Restriction>
            <owl:onProperty rdf:Resource="air:operates"/>
                <owl:allValuesFrom rdf:Resource="air:USFlight"/>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

Notice that we have re-used one of the subclasses we created which were not called for in the specification of Figure 25, showing how this approach improves extendibility. Now *EUAirline*

```
<owl:Class rdf:ID="EUAirline">                                    (55)
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:Resource="air:operates"/>
                <owl:allValuesFrom rdf:Resource="air:EUFlight"/>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

and finally, *InternationalAirline*, which are airlines other than US or EU airlines, using either *intersectionOf*, *complementOf* and *unionOf* like (53) or using the fact that an international airline operates at least one international flight. For the latter, we can use *someValuesFrom*

```
<owl:Class rdf:ID="InternationalAirline">                         (56)
    <owl:equivalentClass>
        <owl:Restriction>
            <owl:onProperty rdf:Resource="air:operates"/>
                <owl:someValuesFrom rdf:Resource="air:International"/>
        </owl:Restriction>
    </owl:equivalentClass>
</owl:Class>
```

11.6.1 Limitations of OWL Descriptions

To someone used to SQL, the range of descriptors used to construct class descriptions in OWL is very limited. For example, the class *PersonOver35YearsOld* can't be defined in OWL, it can only be declared. A class *Person35YearsOld* could be defined using *hasValue* on a property representing age, but there is not a way to represent the subset of a domain on which the value of a property is greater than or less than a nominated value.

Further, consider a class *Rectangle* which is the domain of two datatype properties *length* and *width* both with range *real*. An obvious subclass is *Square*, defined as the instances of *Rectangle* for which the values of *length* and *width* are equal. But predicates comparing the values of two properties cannot be represented in OWL.

The reason for this is that OWL is descended from a class of logical system called description logic, where it is important to be able to check whether a class description is consistent or satisfiable (it is possible to have instances). For example, the intersection of *Square* and *Circle* is not satisfiable. In description logics it is also important to be able to check whether one class description subsumes another. These sorts of checks are not computationally tractable for the full range of predicates expressible in for example SQL, so there are very tight constraints on what predicates can be represented. This is a very big area, and to explore it further would take us too far afield.

11.7 Ontology As an Engineered Object

One of the limitations we saw on RDF and RDFS was that it was hard to define an ontology as a definite engineered object with a creator, versions, and clear boundaries. OWL enables ontologies to be definitively described.

OWL allows an RDF description to be bounded (sharing a base URI) and named (using the header *Ontology*). An ontology can have properties:

- *versionInfo*. A literal identifying a version
- *priorVersion*. A ontology which is a prior version
- *incompatibleWith*. Definitions in one ontology are incompatible with definitions in another. The two cannot be merged.
- *backwardCompatibleWith*. Definitions in one ontology are compatible with definitions in another, generally a prior version.

These properties are built-in to OWL, but it is possible for a given developer to define other ontology properties using the declaration *ontologyProperty*.

Further, in RDF one namespace can link locally to another, so that there is no clear boundary. OWL allows one ontology to be imported into another via an ontology-valued ontology property *imports*. In this way it can be clear which descriptions are part of a given ontology and which are not.

11.8 Flavours of OWL

OWL is specified in three flavours of progressively greater expressivity: OWL Lite, OWL DL and OWL Full. That is to say everything that can be represented in OWL Lite can be represented in OWL DL, and everything that can be represented in OWL DL can be represented in OWL Full.

The biggest distinction is between OWL Full and OWL DL. In OWL Full, classes and properties are individuals, so we can represent classes of classes, classes of properties, and classes can have properties. In OWL DL, the three MOF subclasses of *RDFSResource* in Figure 31 are disjoint, so we can represent none of these things. Furthermore, in OWL Full, literals are individuals while in OWL DL they are not. So in OWL Full there is no real distinction between *ObjectProperty* and *DatatypeProperty*, while in OWL DL there is.

The reason for these two flavours is that it is expected that some ontologies will be used in a way that allows them to be reasoned over by description logic inference engines (DL stands for Description Logic). To make reasoning computationally tractable, expressivity is limited.

OWL Lite further restricts expressivity to make simpler sorts of reasoning even easier. OWL Lite does not support the features *oneOf*, *hasValue*, *disjointWith*, *unionOf* or *complementOf*. Further, *intersectionOf* is allowed only between named classes and restrictions, and the ranges of *maxCardinality*, *minCardinality* and hence *cardinality* are limited to 0 or 1. The last restriction means that OWL Lite can express properties being total or partial with respect to a given class, but cannot express for example the restriction supporting *DoubleMajor*.

11.8.1 Some Things That Can Be Expressed in OWL Full

Having classes and properties being also individuals allows many things to be represented that are very useful.

Class having property. We can define properties like *numberOfInstances* or *dateLastUpdated* with domain *Class*. Each instance of *Class* will be linked to a value for each of these class properties, which can be updated by the repository manager (see Chapter 15).

Class having classes as instances. Consider the simple student records system from Chapter 2, where Faculties create courses, assign lecturers to them, and admit students to degree programs. Students enroll in courses. Lecturers assign marks and advise the Faculties on grades, which the Faculty formally awards.

There are many players in that system, divided into three kinds: faculties, students and lecturers. The objects classes in the ontology include students (as enrolled in degree programs), courses, enrolments and grades. Responsibility for creating instances of these classes is distributed among the players. Students (as enrolled in degree programs) are created by a cooperation between the Faculties and students (as social agents). Courses are created by Faculties. Enrolments are created by students (as enrolled in degree program). Grades are created by faculties on the advice of lecturers, and so on.

Players are semi-autonomous. They each manage their local affairs constrained by their commitments to the common ontology. Each student (as social agent) can maintain an application functioning as an avatar representing them (as enrolled in a degree program), which keeps track of the courses in which they are enrolled, due dates for assignments, marks for assignments, grades for courses completed, and so on. Each lecturer maintains applications managing the class list for each of their courses. Each course knows about the students enrolled, and has local structure – some have only grades while others have a complex structure of individual and group assignments, with marks for each aggregating into a grade for each enrolled student at the end. To a faculty there are collections of students (as enrolled in degree programs), courses, and enrolments with eventually grades.

A conventional view of this system, as would be commonly represented in a UML class diagram, is from the point of view of the University. *Students* and *Courses* are classes, with a many-to-many association representing *enrolment*. The association is reified so it can have attributes such as a *grade*. Each student is seen as a view giving a record of study, while each lecturer is seen as a view giving a class list. From an ontology perspective this way of looking at things is problematic because it does not leave much room for local autonomy in the way the students or lecturers manage their affairs.

From an ontology point of view it makes more sense to think of a collection of student avatar classes, each of which has an association with enrolment in the Faculty, and a collection of classlist avatar classes, each of which also has an association with enrolment. The student avatars can see courses through enrolment by transitive relationship, as can the classlist avatars see students.

This way of looking at the situation gives a large number of student avatar and classlist avatar classes. These classes are in one-to-one correspondence with the populations of the students (enrolled in degree programs) and course classes maintained by the faculty. We can therefore think of the student and classlist avatar

classes as instances of the faculties' student (enrolled in degree program) and course classes.

Classes of properties. Consider an application where we are trying to assemble a consistent set of biographical data for a collection of famous people. We will have a class Person which will have identifiers of all the people of interest. Person will be the domain of a number of properties: dateOfBirth, placeOfBirth, mothersName, fathersName, whetherMarried, and so on. A research team is engaged to find values for all these properties and update them when values are found.

So even though we intend finally to have all the properties be total, during the process there will be many instances of Person which are not linked to values by some of the properties. We might like to be able to identify the subclass of Person for which all properties have a value. (It is clear that allowing classes of properties can make it easy to state this, but how to operationalize the representation would take us deeper into OWL than is warranted here. It involves the fact that OWL uses special properties to make declarations. So it is possible to define a restriction class on *rdfs:domain*, for example.)

11.8.2 Strangeness of OWL Full

The additional expressivity of OWL Full comes at a cost of significant strangeness.

Most modeling languages, and OWL in particular, are based ultimately on the theory of sets. This theory was developed in a formal way in the late 19th century. In the early 20th century a number of people, including Bertrand Russell, discovered serious weaknesses in the theory. A particular weakness is Russell's paradox.

The earlier theory of sets, now called naïve set theory, defined a set as a collection of objects which satisfy some defining predicate. In particular, we can define the set consisting of objects which are sets. This is perfectly reasonable and is commonly done. For example, the MOF class *OWLClass* in Figure 31 is exactly that. It is also reasonable for the set of objects which are sets to be itself a set. Again, in OWL the set owl:Class is an instance of owl:Class, so an instance of itself.

If we have a set of sets, we can define a subset, namely the set of sets which contain themselves. This subset would contain for example owl:Class and owl:Thing, but would not contain for example owl:Nothing. If we have a set and a subset, then we can define the complement of the subset in the set, in particular the set of sets which do not contain themselves. Call this last the set N.

The question is, is N an instance of itself? It can't be, by definition, since N is the set of sets which don't contain themselves. But it must be, by definition, since it is indeed a set which doesn't contain itself. This conclusion is disastrous, since it says that naïve set theory is inconsistent and therefore we can't trust automated reasoners on it.

We can come very close to expressing Russell's paradox in OWL Full. As we have seen we can express a set of sets, a set of sets can contain itself, and we can have the complement of a set. The only thing that saves us it that the predicate language is weak, so we can't define the set of sets that contain themselves. These would be the RDF triples

X rdf:type X (57)

which we can't represent in the same way we can't represent *Square* as the subclass of *Rectangle* for which *length* = *width*. This lack of expressivity is serious

enough that it will very likely be changed in later versions of OWL, or by extensions to OWL such as OWL Rule Language currently under development by the W3C.

OWL is therefore naïve set theory, with all the expressivity but all the pitfalls. An ontology engineer will need to take good care to avoid paradox.

UML, ER and SQL all prevent expression of Russell's paradox by not allowing the definition of a set containing itself. For example, in SQL a table or an attribute can't be the value of an attribute. These systems are all based on post-Russell set theory which avoid the paradoxes by limiting expressivity.

That OWL is subject to paradox is not necessarily a disaster. We know that in most Turing complete programming languages like Java or C it is impossible to develop an algorithm which will determine whether an arbitrary program will go into an infinite loop. Programmers learn to be careful, and computer systems often protect themselves against mistakes by terminating a program that runs too long, even if it might eventually terminate. OWL Full reasoners will need to similarly protect themselves.

11.9 Key Concepts

OWL extends RDFS. OWL includes **Object** and **datatype properties** and a Universal class **Thing**. **Restrictions** are subsets of property domains. Names do not satisfy the **unique names assumption**. Class descriptions can be **enumeration** or **boolean combination** of classes. There are **Ontology properties**. OWL comes in flavours: **OWL Full**, **OWL DL** and **OWL Lite**.

11.10 Exercise

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following, as represented in RDFS in the last exercise.

a. Elaborate a significant portion of the ontology in OWL using the OWL facilities from the Chapter.

b. Describe a concept you can't represent in OWL (make a plausible extension if necessary).

c. How would you use the facilities of OWL Full in this ontology (make a plausible extension if necessary)?

11.11 Further Reading

There is a large amount of material on the World-Wide Web Consortium web site http://www.w3.org/TR/#Recommendations including in particular [18] and [19]. A primer on ontology building in OWL is [20]. A freely available ontology editor supporting OWL is Protégé [21]. The abstract syntax used comes from the Ontology Development Metamodel [22], [23].

12 Advanced Issues

Up until this point we have been discussing aspects of representations of ontologies that apply to pretty well any application. There are very many applications, though, and as we have seen in Chapter 9 they fall into broad classes which have more specific requirements. In this Chapter we will discuss some of these.

12.1 How OWL Relates to Desired Capabilities for Ontology Representation

In the first part of this text we discussed a number of things we are going to need in order to represent conceptualizations supporting interoperating information systems. These include

- Ability to represent individuals
- Player/role/act/fact taxonomy
- Ability to represent complex objects
- Identifying, unifying, essential and rigid metaproperties
- Bulk classes
- Subsumption of classes and properties
- Interaction of metaproperties with subsumption
- Formal upper ontologies

We have presented in detail the ontology representation language OWL in Chapter 11. OWL has native support for only some of these desiderata, namely representation of individuals and subsumption with a limited capability for defined subclasses. On the other hand, OWL has some features not considered earlier in the text, coming from its Semantic Web history:

- Universal Resource Identifiers
- Namespaces with a global scope

OWL is extremely general and lacks many desirable capabilities. We have seen in Chapter 9 that the range of application of ontologies is very great, and there are large classes of ontologies with specific structural requirements not available in the most general representation languages. These more specific application classes are often large enough to support specializations of the general representation languages. And in fact OWL has a facility for defining modules which can be separately published and imported into the representation language when developing a particular ontology. This facility is called the ability to import an ontology.

In this Chapter we will first discuss an important aspect of the OWL metamodel, then examine a number of widely usable facilities, showing how they can be imported into OWL. We will use the term *package* (derived from UML) for a published ontology which can be imported.

12.2 Capabilities of Ontology Platforms

Since an ontology must support interoperating software agents, it will generally be organized into several metalevels of decreasing rigidity. The most rigid metalevel is the set of structural primitives from which the ontology is constructed, generally represented as an ontology language or *metamodel*. The example we have seen of this level is OWL, with structural primitives *class*, *property*, *individual*, *subclass* and so on. Below this metalevel is the schema for the particular community of agents, represented as instances of structural primitives, and below that is the collection of individuals with which the community is concerned. The lower two levels are strictly separated in some metamodels (e.g. UML) and are mixed in others (e.g. OWL Full).

The metamodel is the most stable part of the ontology, with the least ontological commitment. Because of this the same metamodel can be used for many ontologies. It can therefore justify the development of software platforms for ontology development and use. These platforms can include techniques to draw inferences from the structural features of the metamodel, so they can be applied to any particular ontology using those features.

In brief, if we need to add capability to a language like OWL, we can do it in two ways:

- Change the language by changing the metamodel. In this case every application has access to these features.
- Package the capability into a standard module with its own global namespace, and import it as a component of the ontology we are developing. In this case, only applications which have imported the package can use the facilities. But some packages, like XML Schema Literals, become extremely widely used so become de facto parts of the language.

  We will illustrate both of these ways below.

12.3 Avoiding Attributes

First, in our representation language based on OWL introduced in Chapter 4 we have only one way to relate classes of objects to each other, the *property*. We promised a justification for this.

Both Entity-Relationship and UML modeling have two different ways to relate the identifiers of object instances to other objects: the relationship (association) and the attribute. (Object-Role modeling has only one way, the fact type). A relationship is a relation among object identifiers. It is possible for a relationship to be functional (if it is many-to-one or one-to-one).

An attribute is formally a function from an object identifier to a value set, although UML and some versions of ERA modeling permit multi-valued attributes, so the attribute is a relation between object identifiers and members of the value set. (From here we will treat attributes as functional, for simplicity.) Therefore relationships and attributes are formally very similar, so the question arises as to why ERA and UML modeling have the two mechanisms and our representation language based on OWL has only one.

The fundamental distinction is in the semantic importance placed on the relation. An attribute is a light-weight relationship. If we look at the ER model in Figure 16, we see that there are several relationships involving the entity *Product*, some derived from

the relationship between *Order* and *Product* and others derived from the relationship between *Acknowledge* and *Product*. In general, in an ER or UML model, all the connections between one entity or class and another are represented in the model in a structural way. We do not need to know anything about the semantics of the application of Figure 16 to see that there are several connections involving *Product*.

Compare the model in Figure 18, where there are two attributes called *e-mail address*. The model shows a connection between the supertype *Student* and *e-mail address* and a connection between the subtype *E-student* and (a different) *e-mail address*. Formally, the two connections are different. We know that they are in fact the same because we know the application's semantic interpretation of the attribute name. The sameness of the connection is not represented in the structure of the model.

Attributes are widely used, so this kind of situation is very common. An invoice entity may have attributes *total-list-price*, *discount-rate-percent*, *total-pre-tax-price*, *sales-tax-rate-percent*, *sales-tax-amount*, and *total-amount-due*. If these connections were modeled as relationships, there would be two additional entities *Value in AUD* and *Percent*, with four different relationships *total-list-price*, *total-pre-tax-price*, *sales-tax-amount*, and *total-amount-due* between the invoice identifier and *Value in AUD*; and two different relationships *discount-rate-percent* and *sales-tax-rate-percent* between the invoice identifier and *Percent*. These connections are not shown as relationships because they are considered less significant to the application, and their presence in the model would clutter it, obscuring the connections seen as more significant.

Ultimately, of course, the value sets of attributes must be specified. It is common, though, for this specification to be the minimum necessary for the database implementation of the system modeled. Therefore, the value sets of the six attributes of *Invoice* described above might all be *real number*. A problem with such a minimum specification is that the database manager can allow predicates that do not make semantic sense, such as *discount-rate-percent* > *total-list-price*, where semantically the comparison between a percentage rate and a value in dollars is not legitimate. To overcome this kind of problem, the SQL:1999 standard has the concept of *distinct type*, a semantically meaningful value set. Using distinct types, the four values would be declared to be of type *Value in AUD*, while the two percentages would be declared to be of type *Percent*. The two distinct types are join-incompatible, so that the illegitimate predicate is not allowed. Distinct types are related to the domain facility in SQL'92 which is not widely used. A possible reason for this is that people within a given organization understand its domain well enough that they do not make illegitimate queries, so don't need to be prevented from doing so.

However, an ontology is a data representation designed to be shared among several organizations, possibly very many. It is therefore outside the understanding of any individual organization. The agreements on the domains of all attributes must be made explicitly, and documented in a way that all organizations can easily understand. Declared, semantically meaningful value sets are essential in an ontology.

From the modeling point of view, a domain or distinct type is the same kind of thing as an entity type or a class. It is unnecessary to have two different representations for the same concept in the same system. Therefore, if the ontology is represented in an EER system, attributes should not be used, only relationships. Similarly, if the ontology is represented in UML, attributes should not be used, only associations. The W3C modeling language OWL has only one way of representing relationships, the *property*,

which is probably the way of the future. This is why our representation language is based on OWL.

12.4 Bulk Classes

Bulk classes or types have been mentioned briefly in Chapter 5. The advice to use a single method for representing links between classes (e.g. properties) brings bulk types into focus, so they require more extensive discussion.

Formally, a class is a set, whose members are instances of the class. The standard idea of an instance is an object which can be identified and which can be distinguished from other objects. The class is then a set of identifiable individuals. Think of people, customers, suppliers, products, messages, invoices, sporting matches, web sites.

The requirement to be able to distinguish one object from another can be a problem for the sorts of classes often used as value sets for attributes, and therefore as domains. In the physical world, we can distinguish one person from another, and the two persons remain distinguished. We can count the number of people in a room and this number does not change unless some identifiable people enter or leave. From Chapter 5, there is a unifying relation with which we can tell where one person ends and another starts. The same is not true for say clouds. Looking at the sky at a particular time one might be able to distinguish a single cloud, but if we follow this cloud it will often merge with other clouds into a larger cloud mass. This larger cloud mass may later split, but we can not say that the new cloud emerging from the cloud mass is the *same* cloud as earlier merged into the cloud mass. There is no unifying relation to reliably identify the cloud's boundary.

Many things in the physical world have this sort of indefiniteness. We can pour a cup of water into a jug with more water in it, then refill the cup from the jug. The cup has the same amount of water as before, but we would never say it was the *same* water. A class without a unifying relation is called *bulk*. In contrast, a class consisting of identifiable individuals is called *countable*.

Bulk classes are very common in information systems. Consider a system managing inventory for a petrol station. A record is kept of the number of litres of petrol in the storage tank, of the number of litres pumped out of the tank in each sale, and of the number of litres put into the tank in each delivery from the refinery. In each case we know how many litres there are, but we don't know which litres, and can't track the same litre of petrol from the refinery to the customer's tank. The domain in each case is *petrol, unleaded, in litres*, and the entity or class is bulk. Things which have a dimension are generally bulk. A room may have a length and width in metres, but it doesn't make sense to say that the metres are the same.

Value is generally bulk. Just as the petrol in a delivery can't be distinguished in the storage tank and can't be related to the petrol in the customer's tank, neither can the money paid by the customer be distinguished in the money paid to the refinery for the petrol purchased by the station. The domain *Value in AUD* from the previous section is bulk.

Some classes consist of individuals, but those individuals are anonymous, for example apples, pens or sheets of paper, which come in quantities (cases, dozens, reams). Here the technical problem is not the lack of a unifying relation – we can tell one sheet of paper from another if they are both on the desk, and they will still be different sheets the next day. The technical problem is lack of an identifying relation. If

someone comes in and exchanges the two sheets of paper, we can't tell. One pen is like any other. One apple will be subtly different from another, but we rarely pay sufficient attention to notice the difference. A class lacking an identifying relation is also bulk.

One can identify something as being some of a bulk class, but can identify below that only if there is an association with a countable class of *containers*, either physical or metaphorical. We can identify the water delivered to account 54129 for the period 1 January to 31 March, 2003; or the amount of money owed in respect of that delivery; or the wheat in truck VH9302 at 10:00 AM on 23 June, 2003; or the pens in stock in the Southbank warehouse at the close of business on 30 June, 2002.

This ontological distinction between countable and bulk classes has consequences. Consider a relation

R(StudentID, Program, Advisor:Lecturer, BeerDrunk2002:Litres, PencilsUsed2002:units, FeesIncurred2002:Dollars) (58)

Consider the projection $\Pi_{Advisor}R$. The result is a subset of the set of lecturers. The further operation Lecturer/ $\Pi_{Advisor}R$ is the subset of lecturers which are not advisors. *Lecturer* is a countable class, so the projection identifies a subset and the difference identifies another subset.

Now consider the projection $\Pi_{BeerDrunk2002}R$. Let us say that some students have drunk 100 litres, some 105 litres, some 10, some 0. The projection will be the set {105, 100, 10, 0}. There is no useful class *Litres* from which we could take the difference Litres / $\Pi_{BeerDrunk2002}R$. We can re-interpret the value set to be *Beer (litres)*, with the attribute function *Drunk2002*: *Student -> Beer (litres)*, but that does not help. There is no set *Beer (litres)* from which to take the difference Beer / $\Pi_{Drunk2002}R$.

We can look at the problem in another way. Two different students with the same value of *Advisor* are associated with the same value of *Lecturer*. This is interpreted as that the lecturers are the same. Two different students with the same value of *Drunk2002* are associated with the same amount of beer, but not the same beer. More strikingly, two different students with the same value of *BeerDrunk2002* are associated with the same number of litres, but not the same litres, whatever that would mean.

A consequence of the distinction between countable and bulk classes is that the projection operator is problematic with bulk classes. For this and similar reasons our agents will want to know whether a particular class is bulk or countable, since it can perform some operations on one but not the other.

Another consequence of the distinction is that boolean operations intersection, union and difference are very problematic with bulk classes. So long as two countable classes are both subclasses of a universal class (as they are in OWL), the boolean operations apply. An individual is in the intersection of two classes if it is in both of the classes and in the union if it is in one of the classes. These definitions don't apply to bulk classes, since they have no individuals. We might have two bulk classes whose content adds to a third, say total profit in USD = total profit from US Operations + total profit from International Operations, but it is not clear that this can be interpreted as a union of the two classes.

The distinction between countable and bulk classes can be incorporated into the ontology representation language as an imported ontology in two different ways, as in Figure 34. The version on the left imports the distinction as a *metaclass taxonomy*,

which in effect changes the ontology representation language.[19] The version on the right imports the distinction as a set of instances, all using the standard ontology representation language. When creating a particular ontology instance like the examples in Chapter 4 the classes are all instances of the metaclasses in the language definition. In a standard OWL implementation, the class instances are all instances of the MOF class *OWLClass*. All the classes in the examples are OWL classes, instances of the MOF class. If the metaclass taxonomy package of Figure 34 were imported into the OWL MOF class system, it would then be possible to designate class instances as either instances of *CountableClass* or *BulkClass*. We can think of these as being flavours of classes, which could be distinguished by a modeling tool by different colours for example.

The package on the right consists of three distinguished instances of the MOF class *OWLClass*. These instances and any other particular classes in an ontology are all instances of the same MOF class. The ontology definition language does not distinguish them. But a particular class could be designated as a MOF instance of the rdfs:subClassOf relationship with any of the three distinguished instances. The modeling tool would treat them all the same, so would not be able to colour them differently, for example. The distinguished instances approach is not so nice to work with as the metaclass taxonomy, but is much easier to implement.

Nearly all the classes in the Z39.50 or Tic-Tac-Toe example would be designated countable classes, while most of the classes in the dimension example would be designated bulk classes.

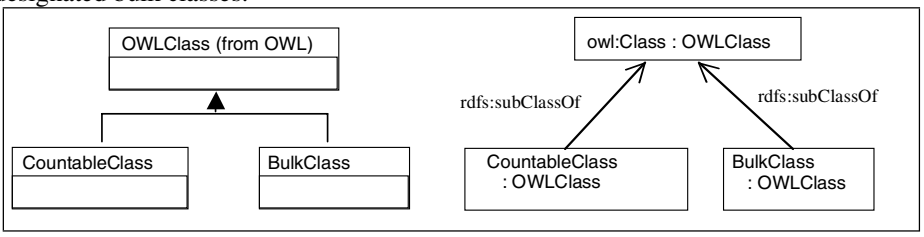


Figure 34. Countable/Bulk package extending OWL as metaclasses (left) or as distinguished instances (right)

This same approaches can be used with other taxonomies of metaclasses, for example the taxonomy of endurants and perdurants proposed in the Dolce system described in Chapter 7.

12.5 Concept Versus Representational Classes

The need for interoperation raises the problem that there may be several different representations of the same concept, which need to be tied together. This problem has already arisen in the discussion of ontologies versus models in Chapter 1, where we noted that individuals may be represented differently in different systems, so that it is

[19] Somewhat ambiguous terminology. A metaclass is what we referred to in earlier chapters as a MOF class. Metaclass is more general than MOF Class, since other metalanguages than MOF exist. A metaproperty, however, is a property of a property. We will use meta-association for a generalization of MOF association.

necessary for each system to record the correspondence between its internal representation and the standard representation specified by the ontology.

For example, I am a person. Most people call me Bob. My mother calls me Robert. My University employer knows me by a payroll number. And since I was until recently a student, also by a student number. The Australian Taxation Office knows me by tax file number. Queensland Transport knows me by my driver's licence number.

All these are representations of the same thing. For two to interoperate they must make a correspondence among their representations. A simple way to model this is to make a distinction between *concept classes* and *representation classes*. So we would make *person* a concept class and the various identifiers representation classes. The applications maintaining the correspondences among representations would use the concept class as a sort of container for the representation classes.

This distinction is made for example in the conceptual modeling system Object-Role Modeling, where what we call a concept class is called a non-lexical object type and what we call a representation class is called a lexical object type. However, in modeling particular information systems it is uncommon to make a distinction between conceptual and representation since there is generally only one representation for any concept. Implementations of Entity-Relationship-Attribute modeling in fact do not generally support the distinction. UML does, with the mechanism of *abstract class*, but this tends to be used more for methods or to organize taxonomies than for data representation. A blank node in RDF can function as a concept class.

Players in B2B e-commerce exchanges need to establish correspondences among their various internal product identifiers. The ontology would have a concept class *productIdentifier*, which would have one-to-one relationships with the several representation classes used by the different players. This collection of one-to-one relationships would be implemented by a table of corresponding product identifiers which could either be stored by the exchange or fragmented in various ways and stored by the players.

As we have seen in Chapter 4, a shared world will often contain instances, both of classes and properties. It is therefore necessary to include individuals in the model of the ontology.

Representation classes obviously can have instances. Concept classes in principle cannot have instances since they are independent of particular representations, although for convenience one might allow a natural language representation not intended to be used other than as a description. Alternatively, a concept class might have as instances object identifiers (OID) such as are used in object-oriented programming languages, or blank node identifiers as in RDF.

Individual classes can have instances. Bulk classes are not sets, since their contents do not carry both unity and identity, but they can have quasi-instances, generally the target of an association with an instance of a countable class. They can also have a fixed set of possible quasi-instances, say a series of standard volumes for containers in a soft drink application.

Since classes, instances and quasi-instances are potentially all parts of an ontology model, they can all appear in the vocabulary of predicates in the constraint/ assertion/ rule languages used in the ontology.

The facility to distinguish concept and representation classes can be added to OWL in the same way as the countable/bulk distinction using a package as in Figure 35. The difference is that a representation class must represent some concept class, so there is a

meta-association as well as metaclasses. This meta-association is an OWL DL property.

As in Figure 34, the concept/representation class structure is shown as a metamodel package and as a structure involving distinguished instances. The latter includes a property *represents* whose domain and range are both owl:Class, similarly to *rdfs:subClassOf*. This is possible only in OWL Full in which classes can be individuals.

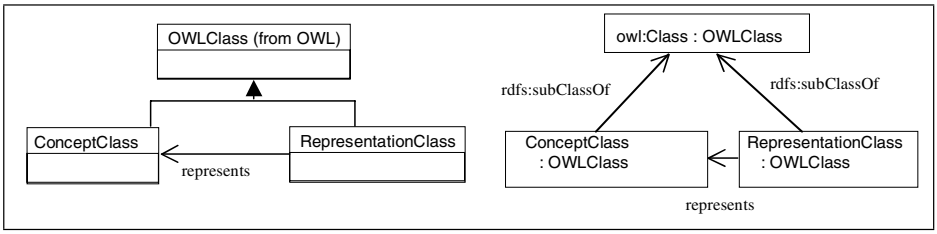


Figure 35. Concept/representation class package extending OWL as a metaclass structure (left) and as a distinguished instance structure (right)

12.6 Dimension

We have discussed some aspects of dimension in Chapter 4. Dimensions are a good application for the distinction between concept and representation class. In the earlier discussion we made the distinction between dimensions like distance, volume or value and units of measure, like metres, litres or Australian Dollars.

Any property relating to the same dimension is comparable to any other. We can compare two speeds. We can compare the speed of movement of the Australian continent relative to the Asian continent (measured in centimetres per century) with the speed cosmic rays (measured in fractions of the speed of light). We can also represent any speed in any unit of measure. The speed of light in centimetres per century is about 10^{20}.

Each dimension in a system is represented by any of a collection of units. It does not make sense to consider the units as subclasses of the dimension, since each dimension necessarily is represented by all of the units. On the other hand, a concrete dimension is almost always represented by a specific unit. The unit can be represented as part of the range of a property instance. The values USD134; 6 pounds, 5 shillings, 3 pence; EUR100 are all valid instances of the bulk class ‘money’. The currencies and values can be extracted by parsing using the formation rules. But usually the unit is factored out, so that all instances of a property have the same unit.

In this case we consider the various unit names as designating representation classes, as distinguished from the concept class which is the dimension within its system. Associated with a concept class for the dimension is a set of representation classes for the units.

So a water supply utility would have a class “product” which is the range of a property “consumed” whose domain is “customer”. Product would inherit from the bulk class “water” and also from the representation class “kiloliters” associated with the SI system class part “volume”. Similarly, there would be a property “owing” between “customer” and “amount”. The latter class would inherit from the concept

class “money” and from the representation class “USD”. An automated reasoning system would navigate through the class/subclass, set/instance and whole/part structure of the ontology.

The relationship among the dimensions and among the units within a system and between systems must be represented as a series of formulas. This is similar to the need to represent assertions, integrity constraints, subclass definition predicates and derived class rules in conceptual modeling languages. A particular ontology would represent these formulas in an appropriate textual language.

The dimension issue is made more complicated by the fact that a specification that a bottle has a capacity measured in litres is not sufficient for interoperation. A program needs to know not only that the class is in litres, but how the litres are represented as numbers (integers, reals, magnitudes in powers of 10, including range and precision). Further, it needs to know how the numbers are implemented (integers as 16 bits least significant bit first, reals as IEEE double float, etc). To cater for these aspects (dimension concept, dimension unit representation, unit number representation, number implementation), it makes sense to use a class with multiple inheritance (faceted) approach.

For example, we could first develop our ontology using a class system appropriate to the application, including a bare class *Capacity*, represented as a subclass of *volume*. Later we could increase our ontological commitment by importing the SI dimension system including volume represented by litres, and assert that *Capacity* is a subclass of *litre*, with the capability to navigate to *volume* and to *SIsystem*. Still later we could increase commitment again by importing a number system and assert that litre is a subclass of the COBOL number with five digits, two digits after the decimal (designated in the COBOL system as ‘999.99’).

Figure 36 shows the result. The representation system is our ontology representation system from Chapter 4, based on OWL. *SIsystem* is represented as a whole with parts (the *partOf* property) *Volume* and *Distance*. *Volume* is shown as a concept class with representations *litre* and *ml*. The two representations are shown interconnected by a bidirectional property *SIconvert*. *SIconvert* designates a collection of methods which given an instance represented in some unit will convert it to another representation. The domain of the property is known when the individual to be converted is known, but the range must be indicated in the property name, so two members of the collection would be *SIconvertToLiter* and *SIconvertToMl*. The various COBOL numbers are shown as representations, with their own conversion properties.

This formulation assumes that the ontology has already imported the *bulk/countable* and *concept/representation* packages. This is why both COBOL numbers and SI System both use the property *representation*. But each of the two more specific package has its own *convert* properties, which are therefore named differently.

Dimension is involved in what sorts of things one can say about a class. A question as to whether one can say 2.5 litres or 2.5 people resolves into one of two questions. First, is 2.5 a valid value for an attribute (or range of a property)? This is settled when the ontological commitment reaches the level of a number system. Second, does it make sense to say that a derived property of a class can be 2.5? Here it partly depends on whether our class is countable or bulk. If countable, the class is a set with a finite number of members, so the standard aggregation operations apply, including count and average of a property. If the class is bulk (litre, ‘999.99’) then aggregation operations do not apply at all since there are no individuals to aggregate.

The collection of properties supporting a dimension system can be packaged with the class system. General facilities such as addition or equality are optional anyway, since for examples reals do not support equality, dates do not support addition and GIF objects do not support greater than. So it is consistent for a subclass to forbid an optional method.

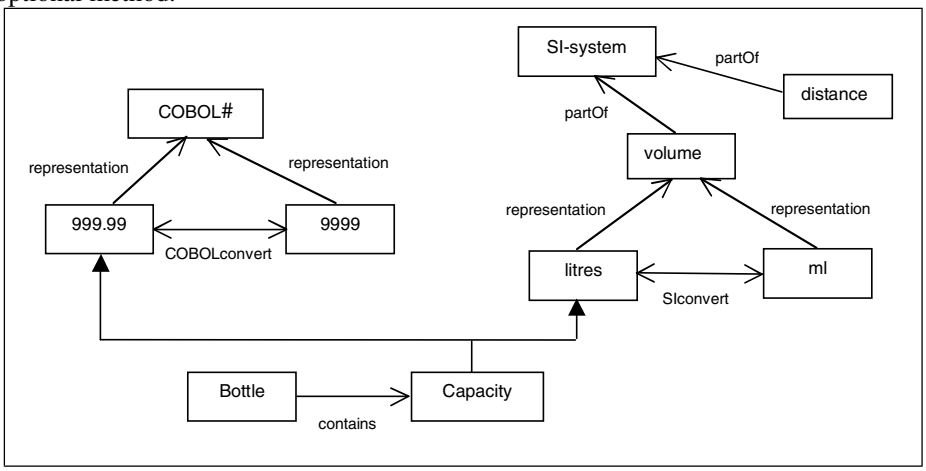


Figure 36. An ontology built from a collection of imported packages

12.7 Representing Mereological Structures

The objects shared are often represented as complex structures consisting of wholes organized into or composed from parts. The ontology must therefore be able to represent these part-whole or mereological structures. One way to do this is to model the mereological relationships needed for a particular application using non-mereological structural features of the chosen metamodel. The disadvantage of doing it this way is that the mereological modeling structures are not structural features of the metamodel, so there is much less scope for the applicability of application-independent software. This is an example of encoding bias. So to reduce encoding bias we would like to have mereological structural features in the metamodel itself. This section suggests a way to do so.

In the following, we first note that there is an enormous variety of the part-of relationship. Consider

- An engine is part of a car, a piston is part of an engine, so a piston is part of a car. Part-of here is transitive.
- A player is part of a team, a team is part of a league, but a player is not part of a league. Part of here is intransitive. (The *partOf* property in Figure 36 is intransitive.)
- An American presidential administration cannot exist without a President (that is why there are succession rules), but it can exist without a Vice-President. Both President and Vice-President are parts of the administration.
- A President cannot exist unless as part of an administration, but an ex-President can. Both President and ex-President are objects which can be parts, but the President depends on being a part while the ex-President doesn't.

- An engine can be part of only one car, but a University academic can be part of many committees.
- When a module is changed in a software product a new instance of the whole is created (new version), while a player on a team can change with the team staying the same.
- A hole is part of a donut, but is not an object.
- The US has 48 parts (states) which are geographically contiguous, but also two (Alaska and Hawaii) which are not.

Many researchers have studied the part-whole relationship in a wide variety of applications. It turns out that there is an enormous number of structurally different such relationships needed to model different situations. For example, Winston, Chaffin and Herrmann (1987) develop a taxonomy based on the three boolean dimensions (shown with examples of *true* and *false* values).

- Functional (role with respect to whole) handle of cup/ tree in forest
- Homeomerous (similar to each other and the whole) grain – salt/ tree - forest
- Separable (can be disconnected from the whole of which they are a part) tree-forest/ gin martini

and five relations:

- component/ integral object (handle/cup),
- member/collection, portion/mass (grain/salt),
- stuff/object (gin/martini),
- feature/activity (paying/shopping),
- place/area.

To minimize ontological commitment, we don't want to adopt a *material mereotopologial taxonomy*. That is, we don't want a system of part-whole relationships which described applications like *stuff/object*, because this commits us to having *stuff* and *objects* in our application. A formal approach is better, giving a language to express a given material flavour without necessarily committing to which flavours will be needed. That is, we want a flavour of part-whole relationship which can be used to represent the *stuff/object* relationship because it has the right formal properties.

But there are so many possible flavours that there is a very large number of possible formal structures to represent them. Our approach is to model the part-of relationship using the OWL structural feature *property*, with a five-dimensional taxonomy of flavours provided by cardinality and other standard OWL property constraints giving a total of 32 flavours. Although this is not nearly enough, the dimensions all support reasoning mechanisms which can be built into generic tools. We suggest a naming mechanism for the flavours based on the taxonomic dimensions, then show how other structural features of OWL can be used to model more specific flavours of meretopological structures.

We model the part-of relationship as a property whose domain is the part and whose range is the whole.

- One formal distinction is whether the property is functional, inverse functional or non-functional. A functional part-whole relationship says that an object may be a part of at most one whole (an engine of a car), an inverse functional relationship says that possibly several wholes have at most one of a part (in a parliamentary government, several ministries may be held by one minister), while a non-functional part-whole relationship says that an object may be a part of many wholes (an academic staff member and committees).

- A non-functional part-of relationship can be transitive or intransitive. An engine is part of a car and a piston is part of an engine, so a piston is part of a car. But piston is now part of two wholes, engine and car, so the relationship is not functional. A transitive property cannot be functional. A non-functional intransitive example is player/team, team/league. A player is not part of the league by virtue of being a member of possibly teams.
- A whole may depend on a part (a US presidential administration depends on the President) or may not (a car doesn't depend on a hubcap).
- A part may depend on a whole (a competitor in a race depends on the race) or may not (a keyboard can exist apart from a computer).
- The whole may or may not depend on its specific collection of parts (*essential whole*) (the software product/ team distinction).

These formal distinctions give rise to 32 combinations, each of which gives rise to a flavour of the part-whole relationship. If the representation language being used to model our ontology had standard names for these flavours of part-whole relationship, it would be easier for agents to make sense of the structural organization of complex objects.

OWL has the facility that properties are by default defined with domain and range both *Thing*. So it is possible to declare properties which are interpreted as particular flavours of the part-whole relationship. OWL has also the facility to import ontologies. If someone were to publish an ontology of part-whole relationships in the same sort of way that the Dublin Core metadata ontology has been published, then particular ontologies could import the part-whole ontology and the terminology widely propagated.

One naming convention is to designate flavours by values of the metaproperties above, as in Table 7. For example, a part-of relationship which was inverse functional with dependent whole and independent part would be named *PartOfiwdpi*.

More specific flavours could be declared, if necessary, by specializing one of the standard properties. For example, a US presidential administration is identified by the name of the person occupying the President part, so is a specialization of the inverse functional part-of relationship with dependent whole and dependent part *PartOfiwdpd*. We could either declare the material relationship directly as a subproperty

presidentPart is subproperty of *PartOfiwdwnpd* with whole identified by the range instance

or we could declare a formal subproperty, say

PartOfiwdpdIDp is subproperty of *PartOfiwdwnpd* with whole identified by the range instance and declare that *presidentPart* is an equivalent property to *PartOfiwdwnpdIDp*. The latter course would be a good option for an ontology of governments where various jurisdictions have different designations of executive. American states have governors, parliamentary governments have prime ministers, and so on.

Table 7. Flavours of PartOf

Metaproperty	Designator
Cardinality	
Functional	f
Inverse Functional	i
Non-Functional Transitive	nt
Non-Functional Intransitive	ni
Whole	
Independent Whole	wi
Dependent Whole	wd
Essential Whole	we
Inessential Whole	wn
Part	
Independent Part	pi
Dependent Part	pd

12.8 N-ary Associations

A key aspect of the OntoClean methodology of Chapter 6 is the concept of a metaproperty. Recall that a property has the metaproperty *essential* with respect to a class if being an instance of that class determines the value of the property. Besides *essential*, the metaproperties include *rigid*, *identity* and *unity*. A property with respect to a class can *necessarily*, *necessarily not* or *not necessarily* have a given metaproperty. A natural way to model metaproperties is as quaternary associations.

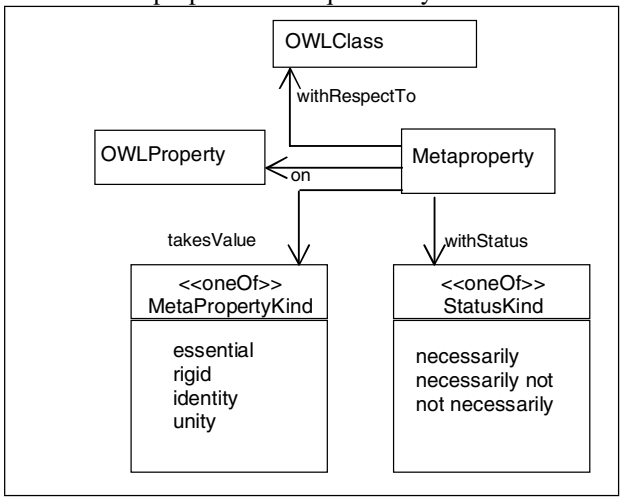


Figure 37. Metaproperty package for OWL

OWL only provides binary properties. But an n-ary association can be represented as a class with n binary properties, as we have seen in Figure 29. A possible package to model metaproperties in Figure 37 extends the OWL metamodel. The metaproperty is modeled as a subclass of *OWLClass*. Note the <<*oneOf*>> annotations, which designate classes whose extents are determined by the OWL enumeration restriction *oneOf*.

12.9 Extent-Descriptive Metaclasses

There is often information *about* a world which is not properly part of the ontology. Examples are the size of populations of class instances or the specificity of attributes. (Specificity of an attribute is related to the frequency of occurrence of an attribute value within a class population. An identifier within a class is maximally specific, a necessary property of the class is minimally specific.) Other examples are the likelihood that a class instance will appear in an optional association or have an optional attribute, how often a class has instances added or deleted, likely values for an attribute within a range of valid values, default values, and so on.

This sort of information is not necessary for the structural integrity of the world, but describes the extents of the classes. We can represent these descriptions using systems of metaclasses like the countable/bulk system. We will call metaclasses of this kind *extent-descriptive* metaclasses.

Extent-descriptive metaclasses can be important for several reasons.

- We will sometimes want to use an ontology as a model of a world for which we what to develop implementations, either conventionally or automatically (*Model-Driven Architecture*). An application which commits to the ontology will construct an implementation of it. *Extent descriptions* of the classes in the ontology is used to inform design decisions. A class with a small population can be cached. A class whose population is never updated can be stored on CD-ROM. A rarely occurring optional association can be implemented in a separate relational table, while a rarely missing optional association may be better implemented as a foreign key attribute with null values allowed. When generating software for a within-enterprise communication application, there may be different design decisions made depending on likely values of network latency between organizational nodes.
- An application committed to an ontology will sometimes want to construct complex queries by assembling information from several other systems committed to the same ontology. The paradigm for this activity is a complex SQL query. In a single relational database, the database manager will decompose the query into its elements then automatically compose a query plan based on the kind of information we are describing, which is conventionally stored in the database manager's system catalog. Where the data is stored in multiple autonomous applications, there is no central system catalog. Extent descriptions can function as at least a partial substitute.
- We may be concerned about the authoritativeness of an implementation of a world. Does the implementation store all the instances of a class, or only a fortuitous selection? Is the implementation the repository of the creators of the institutional facts stored in it, or are we seeing a copy? How often is the information updated?

There is an enormous variety of extent descriptions. Each world or community of users will wish to design their own ontology of concepts to be used in extent-descriptive metaclasses. People could package such ontologies which could then be imported, like the other packages described in this Chapter.

12.10 Discussion

In this Chapter, after making the case for having a single way to represent associations among classes, we have looked at some advanced ontology description features which would not typically be implemented in an ontology description language. Along with the discussion of the feature, we have shown how packages implementing the feature can be published by third parties and imported into specific ontologies.

12.11 Key Concepts

Widely-used structural features include: **countable/ bulk** classes, **concept/representation** classes, **dimension systems**, **mereological structures**, **metaproperties** and **extent-descriptive metaclasses**. All can be modelled using packages in OWL.

12.12 Exercise

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a. Describe examples of bulk classes in the ontology.
b. Find situations where the concept/ representation class distinction is useful. (Make plausible extensions if necessary).
c. Find deep part/whole structures (with more than one level of part), and describe them using the formal taxonomy of the Chapter. (The races, etc of an event might be a good candidate.)
d. Find examples of the metaproperties +unity, +identity +rigid, +essential. Model them using the n-ary association method from the Chapter.
e. Find situations in this ontology where various kinds of extent-descriptive metaclasses might be useful, and describe how in each case. Model them.

12.13 Further Reading

A standard reference on mereology is [24], especially part I. A content-based taxonomy of the part-whole relation is described in [25].

13 Predicates

A predicate is a statement which can be true or false. We have used such statements in various places so far, but never focused on them. In this Chapter we will discuss predicates and their uses, employing the syntax of Common Logic.

13.1 Predicates and Their Uses

A predicate is a statement which can be true or false. Examples of predicates are:

"My surname is Colomb" (59)
"Regina is a student at the University of Queensland" (60)
"Every student has exactly one student identification number" (61)
"All students enrolled in INFS3101 or in INFS7100 are on my classlist" (62)
"If X is a distance in metres, then X/1000 is the same distance in kilometres" (63)

Predicates are intended to be used in reasoning. Given some predicates, we can often conclude others by a process of logic or formal reasoning. The classic example is: given the predicates "All men are mortal" and "Socrates is a man", we can conclude a third predicate "Socrates is mortal." Over the last hundred and fifty years or so there has been a movement towards automated reasoning, to the point that for us as information technologists reasoning is carried on mainly by computer programs called inference engines. In order to automate reasoning we need to replace the natural language used in examples (59) – (63) with formal languages something like programming languages.

There are two main formal logical languages, the propositional calculus (PC) and the first-order predicate calculus (FOPC). The *propositional calculus* is mainly about combinations of uninterpreted predicates. For example, from

If a thing is a kookaburra, then the thing is a bird (p -> q)
If a thing is a bird, then the thing is an animal. (q -> r)

we can conclude

if a thing is a kookaburra, then the thing is an animal (p -> r)

Further, given a true statement

a thing is a kookaburra (p)

we can conclude

the thing is a bird (q)
the thing is an animal (r)

PC often replaces the predicates with symbols like p, q, r; and if … then with a symbol "implies" (->); as is shown to the right of the statements in the kookaburra example above. PC supports the reasoning system Boolean algebra, named after George Boole, a 19th Century mathematician, along with other reasoning systems. Predicates in the propositional calculus are often called *propositions*.

The first-order predicate calculus allows the construction of predicates from simpler parts, so can look inside predicates. It is a much richer language than PC. In fact PC is equivalent to a subset of FOPC. FOPC is the formal language we will mainly consider in this Chapter.

We will need some terminology, which relates to some of the standard uses for predicates. Predicates can

- make statements about definite individuals, like (59) and (60). These are called facts.
- define classes whose instances are facts, like (62). These are called definitions.
- define conditions which must be satisfied for the ontology to be valid, like (61). These are called integrity constraints.
- define rules for deriving instances of the range of a property given instances of the domain, like (63).
- given a definition, find all the facts that satisfy that definition, called a result. These are called queries. Queries are of course the main business of database systems.

These uses are related. A query links a definition with a set of facts. An integrity constraint can be looked at as a query which must have an empty result. Collections of predicates are made to do useful work by programs called *inference engines*. An inference engine does, among other things

- given a fact, determine which classes it is an instance of using the definitions
- given a definition, determine the facts which satisfy it (compute the result of a query)
- calculate instances of the range of a property defined by a rule given instances of the domain
- check that a collection of predicates will continue to satisfy integrity constraints after proposed changes
- determine whether or not a definition is consistent
- determine whether one definition subsumes another

Database systems not only compute the results of queries, they allow the result classes to have properties using the aggregation mechanisms of for example SQL. We can find the number of instances in the result using the SQL facility COUNT or can calculate the total of a quantity property of the instances in the result using the SQL facility SUM. Further, we can derive properties using rules expressed in the SQL facility GROUP BY. If, for example, we have a property *enrolledIn* whose domain is *Student* and whose range is *Course*, and a property *studyingFor* whose domain is *Student* and whose range is the class of degree programs in the University, *DegreePrograms*, we can define a property *totalEnrolment* whose domain is instances of *DegreePrograms* and whose range is a quantity in integers, giving the number of course enrolments totaled by degree program.

Aggregations pose special problems compared to general queries. First, the formal system must permit classes to have properties. Many do not, but OWL does and as we will see so does Common Logic. Second, an aggregation depends on the result of a query being finite. This is not guaranteed in FOPC systems. It is possible to define classes which have an indefinite number of instances, for example the natural numbers: zero is a natural number, and if N is a natural number, then so is N + 1. Aggregation is also a problem in OWL, since OWL does not support the unique names assumption. That a class contains N distinct names does not allow us to infer that it refers to N distinct objects, since it may be discovered later that two names refer to the same object. Aggregation works unproblematically in SQL systems because the result of any query always finite, and SQL systems default the unique names assumption. Stronger

systems which provide aggregation must depend on the programmer's judgment to avoid infinite queries or possible synonyms.

There are many different notations for FOPC. Here we will use the notation *Common Logic* (CL), under development as a standard by the International Standards Organization (ISO) as a replacement for an earlier standard Knowledge Interchange Format (KIF).

13.2 Abstract Syntax for CL

As with OWL in Chapter 10, we will introduce CL in the form of a MOF metamodel. The metamodel of Figure 38 provides what is called an abstract syntax, which shows what parts there are and how they are put together. We will then show the concrete syntax, an implementation of the abstract syntax in an XML environment. There are a number of aspects of CL which do not appear in Figure 38. Further, we do not discuss some aspects of Figure 38, in particular functions and sequence variables, as they are not needed for representation of OWL. Our treatment of CL is far from complete.

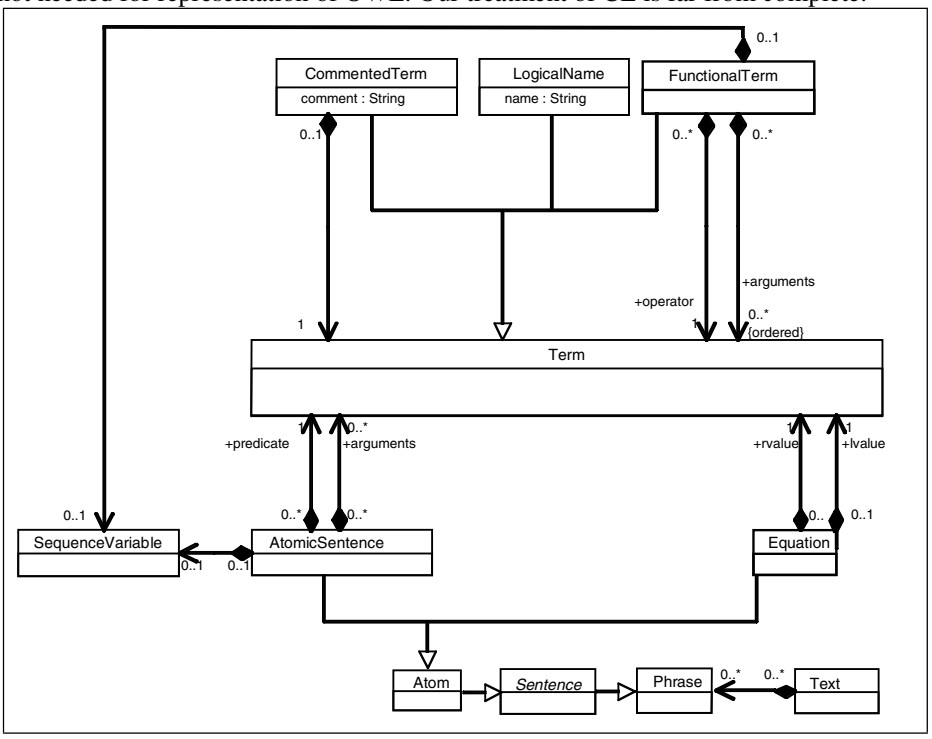


Figure 38. Fragment of MOF metamodel for Common Logic

CL is a very rich system. Anything that can be expressed in OWL can be expressed in CL. So we will explain CL as far as possible in terms of the equivalent structures in OWL.

The basic construct in CL is an *atom*. An example of an atom is the declaration that *air:City* is an *rdfs:Class*

(rdfs:Class air:City) (64)

An atom is composed of terms. As we can see from Figure 38, the syntax for *Term* is quite varied. The simplest term is a *LogicalName*. In the case of (64), the term is constructed from *AtomicSentence*. *AtomicSentence* has two parts, a *predicate* and zero or more *arguments*. The name *rdfs:Class* is the *predicate*. There is one instance of *Term* linked by *arguments*, namely *air:City*. The whole atom is enclosed in parentheses.

The same structure is used to declare that *air:LosAngeles* is an instance of *air:City*

(air:City air:LosAngeles) (65)

An atom with one *argument* is called a *type predicate*, declaring that the *argument* is an instance of the class given by the *predicate*.

A property is declared in a type predicate. For example, that *air:operates* is a property is represented as

(rdf:Property air:operates) (66)

and an instance of a property is represented by an atom with the property name as the *predicate* and two *arguments* denoting the domain and range instances, e.g.

(air:operates air:QANTAS air:QF1) (67)

represents that *air:QANTAS* is an instance of the domain of *air:operates* and *air:QF1* is an instance of its range.

Assuming a suitable interpretation of *me*, the example fact (59) could be represented in CL as

(surname me Colomb) (68)

In (68) we have represented *Colomb* as a term. But we could represent "Colomb" as a (string) literal using a term made from the *FunctionalTerm* metaclass as

(xsd:string 'Colomb') (69)

or more directly as

'Colomb' (70)

since a string enclosed in single quotes is an instance of *LogicalName*.

Example (60) could be represented as a property *student* whose domain is *Person* and whose range is *University*, as the binary fact

(student Regina TheUniversityOfQueensland) (71)

assuming

(Person Regina) (72)
(University TheUniversityOfQueensland)

In fact, the class and property declarations of predicates (64) and (66) are not necessary. The semantics of CL includes that in a type predicate the term in *predicate* names a class and that in the *arguments* the term names an instance of that class. Similarly, the term in *predicate* in a binary *AtomicSentence* names the property, while the terms in *arguments* name the instances of its domain and range.

As with OWL, the instances of *arguments* in a binary *AtomicSentence* can be anything. This faithfully models the fact that the default domain and range of a property are both *Thing*. The description of (71) has that the property has domain and range restricted. CL can represent that restriction using a more complex structure than *atom*, a *sentence*.

An atom can be a sentence, so that (64) through (72) are all sentences. But a sentence can be universally or existentially quantified. For example, that there is at least one instance of *air:City* is declared by

(exists x (air:City x)) (73)

Sentences can be constructed from sentences connected by the boolean operations *and*, *or*, *implies*, *iff* (equivalence) and *not*. We can declare that every international airline is an airline by

```
(forall x (implies (air:International x) (air:Airline x)))                    (74)
```

Using quantified Boolean sentences we can declare predicates like (62)

```
(forall x (iff                                                                (75)
    (or (enrolled x INFS3101) (enrolled x INFS7100))
    (on-my-classlist x)
))
```

This last is an example of a class that in OWL would be defined as the union of two classes, as

```
<owl:Class rdf:ID="OnMyClasslist">                                            (76)
    <owl:unionOf rdf:parseType="Collection">
        <owl:Restriction>
            <owl:onProperty rdf:Resource="#enrolled"/>
            <owl:hasValue rdf:Resource="#INFS3101" />
        </owl:Restriction>
        <owl:Restriction>
            <owl:onProperty rdf:Resource="#enrolled"/>
            <owl:hasValue rdf:Resource="#INFS7100" />
        </owl:Restriction>
    </owl:unionOf>
</owl:Class>
```

In the discussion of (71) we said that the domain of student is *Person* and the range is *University*. We can use the machinery of quantified Boolean sentences to make that declaration, by adding to (71) and (72) the following

```
(forall (s u) (implies                                                        (77)
    (student s u)
    (and (Person s) (University u))
))
```

OWL Lite cardinality restrictions can also be represented as quantified Boolean sentences, for example integrity constraint (61) can be represented as

```
(forall student (implies                                                      (78)
    (exists ID (studentID student ID))                                        (78a)
    (and                                                                      (78b)
        (forall i (studentID student i ))
        (= i ID)
    )
))
```

(That minimum cardinality = 1 is expressed by (78a) and that maximum cardinality = 1 by (78b).)

There might be a temptation to use *and* as the connector in (78), since we want both the antecedent (78a) and the consequent (78b) to be *true*. But what about the situation where the class *studentID* is empty, so there are no instances? The formula (78a) is *false*. If the connector in (78) is *and*, then the whole formula is also *false*. An integrity constraint is a formula which is meant to be *true* for all valid states of the system. It is part of what it means for a possible state to be a valid state. That *studentID* is empty is a valid state. If we use the connector *implies*, then if the formula (78a) is *false*, the whole formula remains *true*.

Finally, complex OWL class definitions can be represented in CL. We illustrate definitions involving *allValuesFrom* by the equivalent of (46), the definition of *USOriginFlights* as flights whose origin is a US city.

```
(forall f (iff                                                    (79)
    (USOriginFlights f)
    (forall c
        (implies (air:origin f c) (USCity c)))
))
```

Definitions involving *intersectionOf*, like (50) that a *USFlight* is the intersection of *USOriginFlights* and *USDestFlights*, are represented

```
(forall f (iff                                                    (80)
    (USFlight f)
    (and (USOriginFlights f) (USDestFlights f))
))
```

Definitions involving *unionOf*, like (52) that *Domestic* is a US flight or an EU flight are represented

```
(forall f (iff                                                    (81)
    (Domestic f)
    (or (USOriginFlights f) (USDestFlights f))
))
```

Definitions involving *complementOf*, like (53) that an international flight is a flight other than a domestic flight, are represented

```
(forall f (iff                                                    (82)
    (International f)
    (and (Flight f) (not (Domestic f)))
))
```

and finally, definitions involving *someValuesFrom*, like (56) that an international airline operates at least one international flight, are represented

```
(forall a (iff                                                    (83)
    (InternationalAirline a)
    (implies
        (and (Airline a)
            (exists f (and
                (operates a f)
                (International f)
            ))
    ))
))
```

13.3 CL Beyond OWL

We can express things in CL that we can't in OWL. In Chapter 11 we discussed the definition of a square as a subclass of rectangle where the length was equal to the width.

```
(forall r (iff                                                    (84)
    (square r)
    (and (rectangle r) (length r l) (width r w) (= l w))
))
```

A CL reasoner can conclude from (84) that *rectangle* subsumes *square*.

```
(forall s (implies (square s) (rectangle s)))                     (85)
```

13.3.1 Defined Subclasses for Tic-Tac-Toe

In Chapter 4, the Tic-Tac-Toe ontology of Figure 8 has many subclasses, all of which were represented as declared. We can use CL to define many of them.

We wish to use a nominally second-order feature of CL, that we can quantify over property names. We can only do this if there is a declared finite population over which to quantify, so the semantics of CL remain first-order. The predicates using this facility can't be represented in OWL.

We declare a class of properties.

```
(cellsOfRow first)                                                    (86)
(cellsOfRow second)
(cellsOfRow third)
```

The subclasses of *Cell*: *X*, *O*, *Filled* and *Vacant*.

```
(forall c (iff (X c) (and (Cell c) (value c 'X'))))                  (87)
(forall c (iff (O c) (and (Cell c) (value c 'O'))))
(forall c (iff (Filled c) (or (X c) (O c))))
(forall c (iff (Vacant c)
    (and (Cell c) not (Filled c))
))
```

Subclasses of *Row*: *Xs*, *Os* and *Mixed*. Subclass *Xs* is rows which could potentially have all cells filled with 'X'. The key mechanism here is at lines 2-6, which say that none of the properties in *cellsOfRow* for that instance of row link to an instance of the subclass *O* of *Cell*. This is an example of universal quantification over predicates (in CL) or properties (in OWL), something that can't be expressed in OWL.

```
(forall r (iff (Xs r)                                                (88)
    (forall (cr c)
        (and (cellsOfRow cr)
            (Rows r)
            (implies (cr r c) (not O c))
        ))
))
```

subclass *Os* is rows which could potentially have all cells filled with 'O'. Also uses universal quantification over properties.

```
(forall r (iff (Os r)                                                (89)
    (forall (cr c)
        (and (cellsOfRow cr)
            (Rows r)
            (implies (cr r c) (not X c))
        ))
))
```

subclass *Mixed* is rows which contain both 'X' and 'O'. Here the quantification over properties is existential rather than universal. It says that the instance of *Row* is linked to a cell in *X* by one of the properties and to a cell in *O* by another.

```
(forall r (iff (Mixed r)                                             (90)
    (forall (cr c)
        (implies (Rows r)
            (and
                (exists (cr c) (and (cellsOfRow cr) (cr r c) (X c)))
                (exists (cr c) (and (cellsOfRow cr) (cr r c) (O c)))
            )))
))
```

Subclasses of *Row*: *FilledRow* and *Possible*. Subclass *FilledRow* is a row with no vacant cells. Here we use a negative existential quantification over properties. There are no cells in the row which are linked to *Vacant*.

```
(forall r (iff                                                    (91)
    (FilledRow r)
    (and (Rows r)
        (not (exists (cr c) (and
            (cellsOfRow cr)
            (cr r c)
            (Vacant c))
        )))
))
```

Subclass *Possible* is a row with a vacant cell all of whose filled cells are the same.

```
(forall r (iff                                                    (92)
    (Possible r)
    (and (Rows r)
        (not (Filled r))
        (or (Xs r) (Os r))
    )
))
```

Subclasses of *State*. Subclass *Won* has subclass *WonX*, where there is a row containing all 'X'.

```
(iff                                                              (93)
    WonX
    (exists r (and
        (Rows r)
        (Filled r)
        (Xs r)
    ))
)
```

and subclass *WonO*, where there is a row containing all 'O'.

```
(iff                                                              (94)
    WonO
    (exists r (and
        (Rows r)
        (Filled r)
        (Os r)
    ))
)
```

Notice that the subclasses of *State* are all propositions. This is a consequence of the Tic-Tac-Toe ontology of Figure 8 not having mechanisms to have more than one game.

Subclass *Won* is either *WonX* or *WonO*

```
(iff                                                              (95)
    Won
    (or
        WonX
        WonO
    )
)
```

subclass *Initial* has all cells vacant

```
(iff                                                              (96)
```

```
    Initial
    (forall c
        (implies (Cell c) (Vacant c))
    )
)
```

subclass *Blocked* has all rows in *Mixed*

```
(iff                                                    (97)
    Blocked
    (forall r
        (implies (Row r) (Mixed r))
    )
)
```

and finally subclass *Winnable* is not *Blocked*.

```
(iff                                                    (98)
    Winnable
    (not Blocked)
)
```

13.3.2 Commuting Diagram Constraints

Consider the model fragment in Figure 39, derived from Figure 16. Each instance of *PurchaseTransaction* is associated with an instance of *Product* by the property *for*. Each instance of *PurchaseTransaction* is also associated with an instance of *Activity* through which the business *completes* its part of the transaction. Each instance of *Activity* is *inRespectOf* some instance of *Product*. The business would expect that the instance of *Product* associated with the instance of *PurchaseTransaction* by the property *for* is the same instance as associated with the instance of *PurchaseTransaction* by *completes* followed by *inRespectOf*. A failure of this expectation would be for example the shipment of the wrong product.

This sort of integrity constraint, where an instance of one class is associated with an instance of another by two different paths, and the two paths must associate the same instances, is called a *commuting diagram constraint*. We can express these constraints in CL. For example, the commuting diagram constraint in Figure 39 is represented as

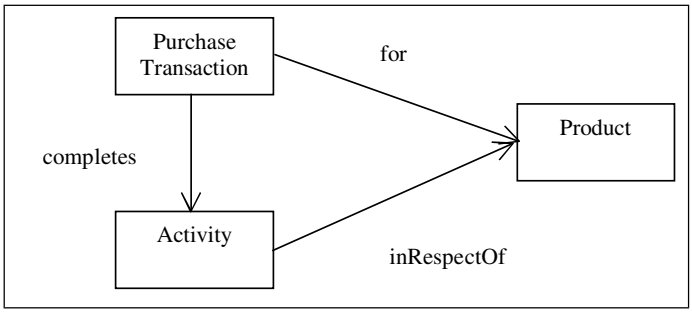


Figure 39. Commuting diagram fragment of Figure 16

```
(forall (t a p1 p2) (implies                            (99)
    (and
        (PurchaseTransaction t)
        (and (Product p1) (for t p1))
        (and (Activity a) (completes t a) (Product p2) (inRespectOf a p2))
```

```
    )
    (= p1 p2)
))
```

These sorts of constraints can be more elaborate. Consider the Tic-Tac-Toe ontology in Figure 8. We want to say that every cell is important in a game. We operationalize this by saying that every cell in a game is a cell in a row which a state of the game is evidenced by. We can get from *Game* to an instance of *Cell* via two paths. One is directly via *has*. The other is indirect, via *hasState*, *evidencedBy* and one of *first*, *second* or *third*. The commuting diagram constraint is represented as

```
(forall (g c) (implies                                                    (100)
    (and (Game g) (Cell c) (has g c))
    (exists (s r cr)
        (and (State s) (hasState g s) (Row r) (evidencedBy s r) (CellsOfRow cr)
            (cr r c))
    )
))
```

13.3.3 Representing Dimensioned Quantities

We have commented on all of the original list of natural language predicates (59) – (63) except for the last one. We lave left (63) until this point because it required not only arithmetic, which CL can represent but OWL can't, but also dimensions. There are many ways of representing dimensioned quantities in CL. The one used here is consistent with the discussion of dimension in earlier Chapters, and is sufficient to illustrate more of the capabilities of CL. We refer in particular to the system of dimensions described in Figure 36.

The model in Figure 36 has a material class *Capacity* which is a subclass of, among others, a dimensioned quantity class *litres*. *Litres* is a unit of measure class representing the dimension class *volume*, shown as a part of the *SI-system*. We can package the entire dimension system aspect of *Capacity* in a single CL structure

```
(dimensioned-quantity Capacity ((SI-system, volume), litres) (number 2.5))   (101)
```

which represents an instance of *Capacity* which is 2.5 *litres*, representing *volume* in the *SI-System*. This structure represents a dimensioned quantity uniformly as an *AtomicSentence* with *arguments* representing the material class; the dimension system, dimension and units of measure; and finally the value as a number.

Representation of arithmetic in CL is under discussion. One of many ways to do it is to use equality as a request to the inference engine to perform a computation in a structure like

```
(forall (x y) ((exists z) (= z (+ x y))))                                 (102)
```

and similarly for the other arithmetical operations '-'. '*', '/', and exponentiation, 'exp'.

If we generalize the structure (101) in the scheme

```
(dimensioned-quantity, materialClass, ((System, dimension), unit) value)   (103)
```

we can represent rule (63) as the sentence

```
(forall mc (implies (materialClass mc)                                    (104)
    (= (dimensioned-quantity mc ((SI-system distance) metres) (* 1000 x))
        (dimensioned-quantity, mc, ((SI-system, distance) kilometres) x)
    )
))
```

This leads to sentences allowing the construction of derived dimensions. For example, we can define area using the sentence

```
(forall (mc1 mc2 units value1 value2) ((exists mc3)                    (105)
    (= (*
            (dimensioned-quantity mc1 ((SI-system distance) units) value1)
            (dimensioned-quantity mc2 ((SI-system distance) units) value2))
        (dimensioned-quantity mc3 ((SI-system area) (exp units 2))
            (* value1 value2))
    )
))
```

We can construct an example using the rectangle structure of (84). We need a material class as the range of the properties *length* and *width*, say *distanceValue*. Applying (103) will construct a new material class which can be the range of a property expressing the area. Call it *areaValue*. So if we adopt *metres* as the unit, the instantiation of (103) is

```
(forall (value1 value2) (=                                             (106)
    (* (dimensioned-quantity distanceValue
            ((SI-system distance) metres) value1)
        (dimensioned-quantity distanceValue
            ((SI-system distance) metres) value2))
    (dimensioned-quantity areaValue
            ((SI-system area) (exp metres 2)) (* value1 value2))
))
```

To develop this sketch into a full dimension ontology would take us much further into the technicalities of CL than is required for this text. CL is still under active development at the time of writing. Once it stabilizes, people will undoubtedly publish dimension systems as packages which can be imported into ontologies.

13.4 Connecting OWL and CL

OWL and CL are different systems, with different metamodels. But OWL lacks a strong predicate mechanism, which CL has. It would make ontology development much easier if the modeling features of OWL could be supplemented by the definition of predicates in CL. We could then develop ontologies like the Tic-Tac-Toe or dimension systems in the combined system.

OWL and CL may be different, but they are closely related. In fact, one of the motivations for the development of CL was that CL could serve as a semantics for OWL. It is possible to translate all of OWL Full into CL.

To see how the connection might be made, note that a predicate is a boolean functional property. So suppose we have a class *boolean* containing exactly the individuals *true* and *false*, and a functional property called *isSquare* with domain *rectangle* and range *boolean*, whose specification is to associate exactly those instances of *rectangle* with *true* for which the *length* property equals the *width* property. We don't at this point care how *isSquare* is implemented. It will be useful for the class *boolean* to have two subclasses *isTrue* and *isFalse* whose instances are respectively exactly the individuals *true* and *false*.

Given *isSquare*, we can define *square* as a subclass of *rectangle* restricted to those instances for which *isSquare* has all values from *isTrue*. A user of this ontology is free to implement *isSquare* any way they like. In particular, they could implement it in CL.

In this way we can use CL as a predicate language for OWL with CL being opaque to OWL.

If we were working entirely in CL we could define the predicate *isSquare* with a sentence like (84). To implement the OWL property *isSquare* in CL, there needs to be a way of referring in CL to things in an OWL model. One way to do this is to provide a mapping between the relevant metamodel elements of RDF/OWL to CL

We can map a class to a term and its properties to roles in its role set. If we represent the mapping of an RDF/OWL model element to SCL with *map*, then the implementation of *isSquare* would be something like

```
(forall r (iff                                                              (107)
    (map(isSquare) r)
    (and (map(rectangle) r) (map(length) r l) (map(width) r w) (= l w))
))
```

Where those instances for which the implication holds are represented in the RDF/OWL model as having *isSquare* associated with *true* and those for which the implication doesn't hold associated with *false*.

This way an ontology expressed in RDF/OWL could use CL as a predicate language without compromising either metamodel. At the time of writing, CL and the details of its relationship with RDF and OWL are under development, so there is not as yet a concrete syntax permitting a translation. That sketched in (107) is certainly inadequate. What is probably needed is some sort of procedure call. Suitable facilities are likely to be available in the near future.

13.5 If OWL Full Is Strange, CL Is Even Stranger

At the end of Chapter 11 we showed that OWL Full permits self-referential statements, so is based on naïve set theory. It is nearly possible to express serious paradoxes in OWL Full, in particular Russell's paradox concerning a set N which is defined as the set of sets which do not contain themselves. Is N a member of itself? The definition both includes N and excludes N.

That discussion showed that what saves OWL Full from Russell's paradox is the weakness of OWL Full's predicate language. But if we succeed in connecting OWL and CL, so that we can implement OWL predicates in CL, then we allow Russell's paradox.

A CL definition of N is

```
(forall s (iff  (N s) (not (s s))))                                         (108)
```

Like the ability to quantify over properties used in (88) – (91), the expression in (108) relies on CL allowing a class name to appear as either the predicate term or argument in an atom, and also on its allowing a variable to appear as the predicate term.

Russell's paradox is that if we accept (109) the predicate (108) forces us to accept (110) and vice versa. The predicates (109) and (110) are clearly inconsistent. It is impossible for N to be both a member of itself and not a member of itself.

```
(N N)                                                                       (109)
(not(N N))                                                                  (110)
```

So, as discussed at the end of Chapter 11, we must be careful using the power of CL.

13.6 Key Concepts

Common logic is a language to express **predicates**. It is possible to map OWL Full to CL. But CL is richer than OWL Full, so can be used to extend OWL. But we need to be careful, since CL open to paradox.

13.7 Exercise

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following. (Make plausible extensions if necessary).

a. Define subclasses using CL.
b. Define some integrity constraints using CL.
c. Formulate some queries as class definitions in CL

13.8 Further Reading

Common Logic is a standard of the International Standards Association. There is a large amount of material at http://cl.tamu.edu/, especially the current version of the developing standard [26]. More information about commuting diagram constraints and their uses can be found in [27]. A detailed analysis of dimension ontologies is in [28]. The abstract syntax used comes from the Ontology Development Metamodel [22], [23].

14 Topic Maps

Topic Maps is another semantic web concept representation language. It is something like RDF, but very different from OWL and Common Logic.

14.1 Topics

RDF arises from the problem of typing hyperlinks, so is based on the fundamental construct *property*. OWL is derived from artificial intelligence and information systems languages, so is based on the fundamental construct *class*. Common Logic is a syntax for the first order predicate calculus, so has the fundamental construct (logical) *predicate*. Topic Maps arise from the problem of constructing electronic indexes for collections of publications, so is based on the fundamental construct *topic*.

Properties, classes and predicates are all set-level concepts. An instance of each is a structure which itself can have many instances. In contrast, a topic is an individual-level concept. The paradigmatic topic is an entry in the index at the back of a book. The terms in a book's index represent the concepts that the publishers of the book think the book is about. In fact, looking at a book's index is a good way to find out what the book is about.

Index terms are generally at the level of specificity of the book's text. If a book's index includes the term "object property", we would expect that the book would use that term or a closely related term in its text. Another way to find out what a book is about is to look at the table of contents. The chapter and section headings say what the book is about at a more general level. Many bibliographic databases use a system of keywords to describe what their articles are about. These controlled vocabulary keywords are usually intermediate in generality between chapter headings and index entries.

For concreteness, in this Chapter we will use the Wikipedia entry for "Semantic Web"[20] as a source of examples of topics and other Topic Maps constructs. In the first paragraph of the entry, the following concepts are mentioned: semantics, World Wide Web, Tim Berners-Lee, World Wide Web Consortium and markup languages. In a Topic Maps representation, these would all be represented as topics.

In Topic Maps, a topic will always be *about* something. What the topic is about is called its *subject*. A subject can be anything at all. It can be a concrete object in the world (e.g. "Tim Berners-Lee"), a complex abstract object in the world (e.g. "Semantic Web", "World-Wide Web", "World-Wide Web Consortium"), an abstract concept (e.g. "semantics", "the number pi"), a fictional concrete object (e.g. "Sherlock Holmes", "Santa Claus"), or even an impossibility (e.g. "a square circle", "faster-than-light travel"). Since computer systems in general and the world-wide web in particular are in the world, some subjects are objects that exist in computer systems (e.g. "The Wikipedia entry for Semantic Web").

[20] http://en.wikipedia.org/wiki/Semantic_Web

The subject of a topic will sometimes be something existing in a computer system whose IRI can be dereferenced by systems software to access the subject, like "The Wikipedia Entry for Semantic Web" <http://en.wikipedia.org/wiki/ SemanticWeb>. In this case, the IRI will be stored in the *subject locator* attribute of the topic. In other cases, a subject of a topic in a topic map is identified by a text string (in Unicode) called a *subject indicator*. The subject indicator will be stored in the *subject identifier* attribute of the topic. A subject identifier for the "World Wide Web Consortium" could be <http://w3c.org>. That IRI can be dereferenced, but what would be reached is not the World Wide Web Consortium, but its web site. Subject indicators are a sort of name, and the name must be unambiguous in the context of the World-Wide Web. It will generally conform in syntax to IRI or some similar scheme.

A topic may have any combination of subject identifiers and subject locators, but they must be synonymous. A topic is identified by its collection of subject identifiers and locators.

Subject identifiers and locators are identifiers of main use to the computer systems. For humans, a topic will generally have a *name*. The strings "semantics", "World Wide Web", "Tim Berners-Lee", "World Wide Web Consortium" and "markup languages" are all topic names. Often, a subject will have several synonymous names. The World-Wide Web Consortium is often referred to as "W3C". A topic may have more than one name, or no names.

A topic is one kind of object in a topic map, and a topic name is another kind. As with RDFS/OWL and Common Logic, we show the abstract syntax of Topic Maps as a MOF metamodel, in Figures 40, 41 and 42. The first, Figure 40, describes the central constructs, *Topic* and *Association*, with their parts. The attributes of *Topic*, *subjectIdentifier* and *subjectLocator*, are shown in Figure 41.

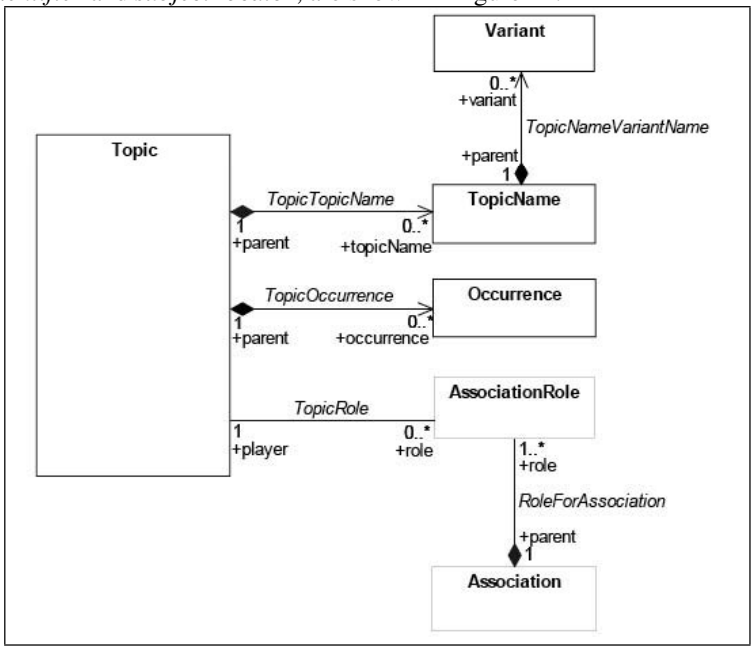


Figure 40. The Topic class

Immediately, we can see the concepts we have already discussed: topic and topic name, and can see that a topic is associated with zero or more names, but a name is associated with exactly one topic.

14.1.1 Occurrences

Simple use of a word or phrase designating a topic isn't generally what someone looking for information wants. They generally are looking for some substantial information about the topic. In a book, each index entry will include page numbers on which the concepts named will be defined, or commented on, or used in context, or shown examples of. Each of the topics in the first paragraph of the Wikipedia entry above is linked to a Wikipedia entry on that topic.

In a computing environment, what we want from a topic is an *information resource*, something that we can access via our terminal, that is about the topic. If we have the Wikipedia entry for the Semantic Web on our web browser, we can click on the entry "Tim Berners-Lee", and be taken to a Wikipedia entry about him.

In Topic Maps, the link between a topic and an information resource is an *occurrence*. In Figure 41, *Occurrence* is shown to have two attributes: *value* and *datatype*, both strings. If the datatype is "IRI", then the value is an IRI that can be dereferenced to access the information resource. If the resource were the Wikipedia entry for Tim Berners-Lee, then the value of *datatype* would be "IRI" and the value of *value* would be "http://en.wikipedia.org/wiki/Tim_Berners-Lee".

On the other hand, the occurrence can contain the information resource. If the resource is Tim Berners-Lee's birthdate, the value of *value* would be "1955-06-08" and the value of *datatype* would be "xsd:date". If the resource were a photo of Tim Berners-Lee, then the value of *value* would be a bitstring in say JPEG format, and the value of *datatype* would be "http://www.w3.org/Graphics/JPEG/jfif.txt".

An occurrence is like a hyperlink in that it allows the user to access an information resource pertaining to the topic. But a topic can be associated with many occurrences, as we can see from the multiplicities on the property *occurrence* in Figure 40. So a topic is more like a query to a search engine than a hyperlink anchor. At the time of writing, Google gives 783 responses to the query "Tim Berners-Lee". However, unlike a search engine, all the occurrences associated with a topic are actually about that topic. In a search engine there can be false matches. A topic map is a constructed artifact, so every occurrence is judged to be relevant to its topic by the designers of the map.

Recall from Chapter 10 that we can consider a property in RDF to be a typed hyperlink, with the type given by the predicate name. Occurrences in Topic Maps are also typed. The type of an occurrence is represented by another topic, as we can see from Figure 42. The metaclass *Occurrence* is a subclass of *TypeAble*, and *TypeAble* has a property *type* with lower multiplicity 1. So the three examples of occurrence above would all be typed. The Wikipedia entry might have type [WikipediaEntry], the date of birth might have type [DateOfBirth], and the photo might have type [Portrait].

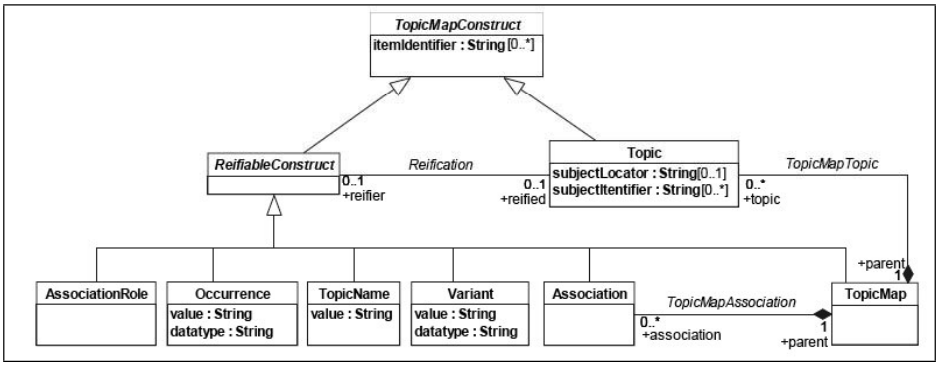


Figure 41. Topic Maps constructs

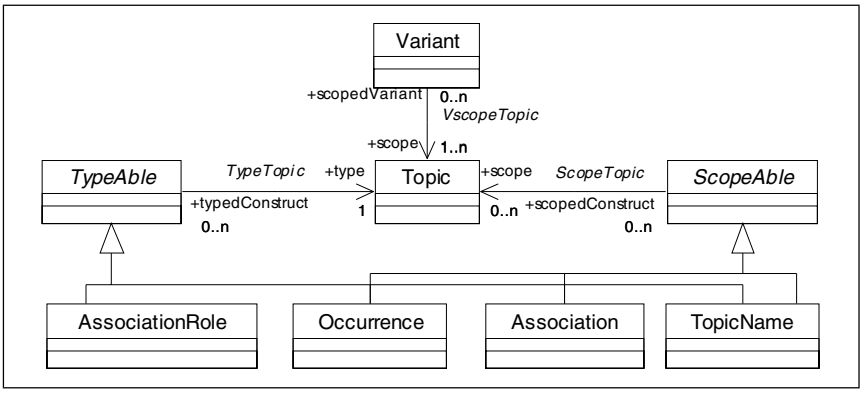


Figure 42. Type and Scope

14.2 Linear Topic Maps Notation

Our examples are getting more complex, so we need to express them in a formal notation. Like RDF, Topic Maps has a number of notations. The official notation is XTM which is based on XML, but a more human-readable notation is the non-standard Linear Topic Maps notation (LTM) developed by a company called *Ontopia*.

The topic we have referred to as "Tim Berners-Lee" would be identified by the URI "http://www.w3.org/People/Berners-Lee/card#i". So although the URI can be dereferenced as an FOAF (Friend of a Friend) card, the resource dereferenced is not Tim Berners-Lee, only some information about him. But no other person has that URI, so it uniquely identifies Tim Berners-Lee.

Using a URI to identify a topic tends to defeat the purpose of LTM, to be a human-readable notation. But a subject can have multiple subject indicators, and a topic map is a constructed object with a definite boundary. Note that in Figure 41 a topic has direct attributes *subjectLocator* and *subjectIdentifier*, but also inherits an attribute *itemIdentifier* shared by other Topic Maps constructs. The subject locator and subject identifier are constrained to be universal in scope, but an item identifier can be local to a topic map. This gives more freedom to make human-understandable identifiers. LTM takes advantage of this by representing a topic by an item identifier in square brackets.

So we could assign an item identifier "TBL" to "Tim Berners-Lee", to the topic would be

[TBL] (111)

In LTM, item identifiers are conceived of as fragment identifiers on a URL base which is the URL of the file containing the LTM statements. But the item identifiers are not subject identifiers. They identify the topic, not what the topic is about. We can think of them as a kind of abbreviated name for the topic intended to be read by the ontology designer.

Topics do have properly designated names, so (111) would become

[TBL = "Tim Berners-Lee"] (112)

A user would see the name in preference to the identifier.

We can include the URI as a subject identifier using the symbol "@"

[TBL = "Tim Berners-Lee" (113)
@ http://www.w3.org/People/Berners-Lee/card#i]

If the subject has a subject locator, we can represent it using the symbol "%". A topic whose subject is the World-Wide Consortium web site could look like

[W3CWebSite = "World-Wide Consortium web site" %http://w3c.org] (114)

Occurrences are defined using curly brackets. First appears the topic identifier, then the identifier of the occurrence's type topic, then the URI of the information resource, for example

{TBL, WikipediaEntry, "http://en.wikipedia.org/wiki/Tim_Berners-Lee"} (115)

Occurrences which contain their value are represented with the value in pairs of square brackets, for example

{TBL, DateOfBirth, [["1955-06-08"^^xsd:date]]} (116)

Occurrences whose value is non-textual can be handled in the same sort of way as in web pages, by relative URLs. A photo of Tim-Berners-Lee is stored in a file, so an occurrence might be

{TBL, Portrait, "tblportrait.jpg"} (117)

We will introduce additional features of LTM syntax as we describe the remaining constructs of Topic Maps, below.

14.3 Associations

Up to this point we have defined topics, which refer to resources, and occurrences, which link a topic to an information resource which is about the topic in some way defined by the occurrence's type. But topics have been seen as isolated objects.

Consider the collection of example topics derived from the Wikipedia entry for Semantic Web, that we have seen above. Information resources about [TBL = "Tim Berners-Lee"] may or may not mention that he is the inventor of the World-Wide Web and the director of the World-Wide Web Consortium. Similarly, information resources about [WWW = "World-Wide Web"] or [W3C = "World-Wide Web Consortium] may or may not mention relationships between the two or with [TBL].

Topic Maps allows us to represent relationships among topics directly, using the construct *Association*. In Topic Maps, an association links a number of topics, and topics are individual-level constructs. The fact that several topics are linked in an association in itself says nothing about the nature of the association. In contrast, the UML construct with the name *Association* is a class-level construct. A UML

association specifies a set of links among instances of classes. The association has a name, as do the two ends. OWL properties also have names. So in UML and in OWL an association/property does indicate the nature of the association.

However, as you can see from Figure 42, an instance of *Association* must be linked to an instance of *Topic* which is the association's type. So associations are typed in the same way as occurrences. An association is a more general construct than an occurrence. An occurrence is directed: the value is about the topic. An association is not directed.

Before we complete the discussion about associations, we will first show the LTM notation for associations. An association among a number of topics is shown by listing the topics after the association's type, for example

```
inventorOf(WWW, TBL)                                    (118)
```

This says that [WWW = "World-Wide Web"] and [TBL = "Tim Berners-Lee"] are linked in an association of type [inventorOf]. It doesn't say which is the inventor and which is the thing invented. To specify this, we need the additional construct *AssociationRole*. Association roles in effect order the topics linked in an association. Like *Association*, *AssociationRole* is an individual-level construct, but also like *Association* and *Occurrence*, association roles are typed (Figure 42). The type of the association role gives the role that the topic plays in the association. So if we represent the concept *inventor* by the topic [inventor] and the concept *thing invented* by the topic [thingInvented], we can make the statement "Tim Berners-Lee is the inventor of the World-Wide Web" using the association

```
inventorOf(WWW:thingInvented, TBL:inventor)             (119)
```

where the type of an association role is shown by the topic following the colon.

Serialization of an association is arbitrary. The statement (119) is equivalent to

```
inventorOf(TBL:inventor, WWW:thingInvented)             (120)
```

We know from conceptual modeling languages (Appendix D) that it is possible to have relationships among any number of objects. If there are N objects, the relationship is said to be N-ary. In practice, though, most relationships are binary. Indeed OWL allows only binary properties (as relationships are called in OWL).

In Topic Maps, associations can be N-ary. For example, we have the association among [TBL], [WWW] and [W3C] that expresses the statement "Tim Berners-Lee guides the development of his invention, the World-Wide Web, via the organization World-Wide Web Consortium"

```
guidesDevelopmentOfBy(                                  (121)
    TBL:inventor, WWW:thingInvented, W3C:guidingOrganization)
```

Association (121) is built using the types [guidesDevelopmentOfBy], [inventor], [thingInvented] and [guidingOrganization]. As with any type system, the type topics can be used to construct analogous associations among other topics. For example, we have other inventors who have led organizations developing their inventions: Thomas Edison formed Edison Electric Light Company to develop the light bulb, among other inventions, and Edwin Land formed Polaroid Corporation to develop the instant camera. So if we have the topics [ThomasEdison], [EdisonElectricLightCompany], [lightBulb], [EdwinLand], [PolaroidCorp] and [instantCamera], we can state two other instances of the association template of (121)

```
guidesDevelopmentOfBy(                                          (122)
    ThomasEdison:inventor, lightBulb:thingInvented
    EdisonElectricLightCompany:guidingOrganization)
guidesDevelopmentOfBy(
    EdwinLand:inventor, instantCamera:thingInvented,
    PolaroidCorp:guidingOrganization)
```

All associations are typed, as are all association roles, but keep in mind that the type declarations are at the individual level, not at a class level. An association of a given type can have instances whose roles are of different types. Even the arity of the association instances can be different. For example, some inventions have more than one inventor. Consider the airplane, credited to Wilbur and Orville Wright. So (120) is compatible with

```
inventorOf(WilburWright: inventor, OrvilleWright: inventor,     (123)
    airplane: thingInvented)
```

Some inventors invent more than one thing. Besides the light bulb, Thomas Edison is known for inventing the phonograph, among many other things. So (120) and (123) are both compatible with

```
inventorOf(ThomasEdison: inventor, lightBulb: thingInvented,    (124)
    phonograph: thingInvented)
```

Note that (124) is not necessarily good practice in Topic Maps. Many would argue that the statement should be separated into two. The point here is simply that Topic Maps syntax permits a structure like this.

In the same spirit, we know that some things are invented by companies rather than individuals. So Microsoft is credited with inventing both Windows and Excel, among other things. (120), (123) and (124) are all compatible with

```
inventorOf(Microsoft: inventingCompany, Windows: thingInvented,  (125)
    Excel: thingInvented)
```

The only limit on the variety of association role types and arities of association instances is the ability of the designers and users of Topic Maps to make sense of it all.

This is of course quite different from RDFS, OWL, and the conceptual modeling languages of Appendix D. In those systems, relationships, properties, etc are all typed at the class level, so all instances have the same structure. Common Logic has more of the flexibility of Topic Maps.

Finally, the LTM allows the definition of topics inside association declarations. Instead of (114) and (120) the two can be combined

```
inventorOf([TBL = "Tim Berners-Lee"                             (126)
    @ http://www.w3.org/People/Berners-Lee/card#i]:inventor,
    WWW:thingInvented)
```

The topic [TBL] can now be used elsewhere without further definition.

14.4 Scope

Topic Maps can contain large amounts of information about their topics. Different people are interested in different information, even different information about the same topics. There are often large groups of people who share broad interests, though. A good example of people who share broad interests is speakers of a given natural language – they generally relate best to information expressed in the natural language they speak. Professional disciplines are another example.

Topic Maps allow some constructs to be assigned a *scope*. Scope can be used for example to identify occurrences by the natural language their information objects are expressed in. From Figure 42, we can see that a scope is a topic which can be associated with an occurrence, association or topic name. So if we define a series of topics [en] for English, [ar] for Arabic, [zh] for Chinese, [fr] for French, [it] for Italian, etc., using the ISO 639 codes for topic subject indicators, we can tag occurrences and names by language.

Topic names are scoped in LTM by placing the scope topic after the name separated by '/', as in

```
[TBL = "Tim Berners-Lee" / en]                                    (127)
```

Occurrences are scoped by placing the scope topic after the occurrence, separated by '/', as in

```
{TBL, Wikipedia Entry, "http://en.wikipedia.org/wiki/Tim_Berners-Lee"} / en   (128)
{TBL, Wikipedia Entry, "http://it.wikipedia.org/wiki/Tim_Berners-Lee"} / it
{TBL, Wikipedia Entry, "http://fr.wikipedia.org/wiki/Tim_Berners-Lee"} / fr
{TBL, Wikipedia Entry, "http://zh.wikipedia.org/wiki/蒂姆·伯纳斯 - 李"} / zh
```

Associations can be scoped. One reason for doing so is that different communities often will have different links among topics. In particular, during the Cold War there was often a difference of opinion as to the inventor of important technologies between American and Soviet sources. The Soviets claimed that Konstantin Tsiolkovsky invented the rocket, while the Americans credited Robert Goddard with that invention. In LTM, associations are scoped in the same way that occurrences are. Defining the appropriate types, we could have

```
inventorOf(Rocket:thingInvented, KonstantinTsiolkovsky:inventor)/Soviet   (129)
inventorOf(Rocket:thingInvented, RobertGoddard:inventor) /American
```

Note that association roles cannot be scoped. An association is either present in a scope or not present.

A Topic Maps construct peculiar to scope is that of variant name. Notice in Figure 42 that although *scope* is optional for the other constructs, it is mandatory for *Variant*. In Figure 40 we see that *Variant* is a part of *TopicName*, so an instance of *Variant* is considered as an alternative name for a topic which is valid in a certain scope. Variant names are synonymous. LTM allows variant names to be represented in parentheses after the name, with the scope represented in the usual way, for example

```
[TBL = "Tim Berners-Lee" ("蒂姆·伯纳斯 - 李" / zh)]                (130)
```

Topic Maps includes a built-in variant scope, *sort name*, whose subject identifier is the IRI "http://psi.topicmaps.org/iso13250/model/sort". A sort name is a variant name which will force the name into the desired sequence when sorted. So the name "Tim Berners-Lee" would have a sort name "Berners-Lee, Tim". LTM allows the representation of a sort name variant as a string following the (base) name separated by a semicolon. The sort name variant precedes any other variant.

```
[TBL = "Tim Berners-Lee" ; "Berners-Lee, Tim" ("蒂姆·伯纳斯 - 李" / zh)]   (131)
```

An association, occurrence, topic name or variant can have several scopes, as you can see from the upper multiplicity on *scope* in Figure 42.

14.5 The Topic Map

Topic Maps objects are grouped together in topic maps, in the same sort of way that objects in an OWL ontology are grouped together in ontologies, objects in Common Logic models are grouped into modules, or objects in UML models are grouped into packages. So like these, and unlike RDFS graphs, a topic map is an engineered object, with a creator, date of creation, versions etc.

In Figure 41, you can see that every instance of *Topic* or *Association* is associated with exactly one instance of *TopicMap*. The diamond notation indicates that an instance of *TopicMap* has parts, which are instances of *Topic* and *Association*. In Figure 40, you can see that every instance of *TopicName*, or *Occurrence* is associated with exactly one instance of *Topic*, and that every instance of *Variant* is associated with exactly one instance of *TopicName*. Further, every instance of *AssociationRole* is associated with exactly one instance of *Association* as a part of the association instance. Therefore every object in a Topic Maps model is associated with an instance of *TopicMap*.

There can be several topic maps in the same model. An association in one topic map can have an association role associated with a topic in another topic map. Topics which are types and scopes of objects in a topic map can be parts of other topic maps.

In Figure 41, there is an abstract superclass *TopicMapConstruct* which supplies an attribute *itemIdentifier*. This attribute is used to provide a topic map-specific local identifier for all Topic Maps constructs. This identifier can be used in the repository storing a model, or in an XML serialization of a Topic Maps model being transported from one application to another.

Note that it is possible for an instance of *Topic* to have neither a *subjectIdentifier* nor a *subjectLocator* (an anonymous topic, something like a blank node in RDF). In this case the topic instance must have a value of *itemIdentifier*. Otherwise the *itemIdentifier* attribute is optional.

14.6 Reification

The only kind of object in a topic map that can be the subject of an occurrence or participate in an association is a topic. But the other objects in a topic map are objects in the world. In particular, in an engineered object like a topic map, there are all sorts of information that one would like to record about topic maps, occurrences, associations and so on.

For the topic map itself, we would like to be able to record who created it, when, which version it is, and the other kinds of information that are recorded in OWL by ontology properties. For associations, occurrences and the other parts of a topic, we would like to record who has authorization to change the object, what version it relates to, date created, and a wide variety of other things, the sorts of things recorded in OWL by annotation properties or in UML by comments.

Topic Maps provides a mechanism for this. It is possible to associate a topic with any other kind of object in a topic map by a relationship called *reification*, as shown in Figure 41. The topic reifying a topic map can be the subject of the maker, version, etc as either occurrences or associations. The topic reifying any construct can be the subject of an occurrence or association giving authorization, date created, or any kind of comment or annotation. Topics used for this purpose are often anonymous, since the

object they are about is recorded by the reifier/reified association, not by subject identifiers or locators.

Note that it is not possible to reify a topic.

A topic map is reified in LTM using a directive, called the topic map directive. If we had a file containing a topic map about for example the semantic web, we could reify it with the statement

```
#TOPICMAP ~semanticweb                                    (132)
```

This creates a topic [semanticweb], which could then be used to make the occurrences

```
{semanticweb, creator, [["Robert Colomb"]]}               (133)
{semanticweb, created, [[2006-08-14]]}
```

In this case we have used topics whose subject identifiers are IRIs from the Dublin Core metadata set.

```
[creator, @"http://purl.org/dc/terms/creator"]            (134)
[created, @"http://purl.org/dc/terms/created"]
```

Other kinds of Topic Maps objects can be reified in LTM using the "~" notation as in

```
inventorOf(TBL:inventor, WWW:thingInvented) ~ tbl-top     (135)
```

reifying the association (120). The resulting topic [tbl-top] can be used as the subject of occurrences or associations e.g.

```
{tbl-top, created, [[2006-08-14]]}                        (136)
```

14.7 Topic As a Class

A topic is an individual-level construct, but we have seen that topics can be used to designate classes whose instances are occurrences, associations, association roles or topic names. If we are to use topics to designate classes, then it is very convenient if we can have a subclass structure. Topic Maps provides a mechanism to do this.

Two topics can be placed in a subclass/superclass relationship using an association with type a topic with the special IRI "http://psi.topicmaps.org/iso13250/model/supertype-subtype". The association has two roles, of type respectively "http://psi.topicmaps.org/iso13250/model/supertype" and "http://psi.topicmaps.org/iso13250/model/subtype".

If we wanted to declare the topic [inventorOf] as a subclass of the Dublin Core class [creator], we would express this in LTM as

```
subtype-supertype(inventorOf : subtype, creator : supertype)   (137)
[subtype-supertype @http://psi.topicmaps.org/iso13250/model/supertype-subtype]
[subtype @http://psi.topicmaps.org/iso13250/model/subtype]
[supertype @http://psi.topicmaps.org/iso13250/model/supertype]
```

Further, it is often convenient to group topics in set-instance relationships with other topics. This is particularly true if we wished to use Topic Maps to implement information system oriented ontologies such as the airlines ontology of Chapter 10. Our OWL-based representation language distinguishes between class names and the names of individuals used as instances of classes, while Topic Maps has only the one construct, *Topic*. Topics can be declared in a set-instance relationship with each other in a way similar to the subtype-supertype relationship, using another set of association and association role types. The association type is "http://psi.topicmaps.org/iso13250/

model/type-instance", while the association role types are "http://psi.topicmaps.org/ iso13250/ model/type" and "http://psi.topicmaps. org/iso13250/model/instance".

If we define the elements of the airline ontology as topics, we can declare Sydney and Los Angeles as instances of the class City by

```
type-instance(LosAngeles : instance, City : type)                    (138)
type-instance(Sydney : instance, City : type)
[type-instance @http://psi.topicmaps.org/iso13250/model/type-instance]
[type @http://psi.topicmaps.org/iso13250/model/type]
[instance @http://psi.topicmaps.org/iso13250/model/instance]
```

Topics declared as types whose instances are other topics can have subtype-supertype relationships declared just as topics used as types for other kinds of Topic Maps object.

Both the subtype-supertype and type-instance associations can be scoped, just like any other.

We see that broader (more general) term/ narrower (more specific) term relationships among topics can be represented using associations. Besides the built-in subtype/supertype and type/instance relationships, an application can define other general relationships such as whole/part.

14.8 Merging Topic Maps

We often want to create a single topic map by combining several others. In Topic Maps, this operation is called *merging*. It is similar to the operation in OWL of importing an ontology, and to the operation in UML of merging packages. The Topic Maps data model prescribes the merge operation.

Topics which are found to be synonymous are merged into a single topic. Two topics are synonymous if the set of subject identifiers and locators of the two are not disjoint. In this case a new topic is created whose subject identifiers and locators are the union of the subject identifiers and locators of the two topics. The new topic will participate in all of the occurrences, associations, etc of each of the original topics. Note that a topic used to reify another Topic Maps object can be synonymous only with another topic reifying the same object.

Merging of two topic maps is accomplished in LTM using a directive "#MERGEMAP" with argument in "" the file name containing the file to be merged with the file containing the declaration. The result of the directive is that the external topic map is merged with the local topic map. Two topics in different maps with the same item identifier will not be merged unless one of their subject identifiers or locators is the same. LTM has also a #INCLUDE directive that works like #MERGEMAP. The difference is that it assumes that the topics in the included file are in the same namespace as the including file. (The included file is unchanged.)

Assume for example we are constructing a topic map, and use LTM topic item identifiers "creator", "created" and so on. These item identifiers are the same as the item identifiers in the published Dublin Core topic map. If we merge the Dublin Core topic map using the #MERGEMAP directive, the Dublin Core topics will be separate from the local topics. If we use the #INCLUDE directive, the Dublin Core topics with item identifiers "creator", "created" etc. would be merged with our topics with the same item identifiers. The effect would be to add the Dublin Core subject identifier IRIs "http://purl.org/dc/terms/creator", "http://purl.org/dc/terms/created", etc. to our topics.

Once topics have been merged, their topic names, variants, occurrences and associations can be merged. Two such objects are equivalent if their own attributes are the same and all their associated topics are the same after merging. This last step may allow topics reifying objects to be merged, if their reified objects are merged.

14.9 Key Concepts

A **topic map** consists of **topics** and **associations**. Topics have **names** and **occurrences**, and the names may have **variants**. Associations are linked to topics using **association roles**. Topic names, occurrences, associations and association roles are all **typed**. Topic names, occurrences and associations may also be **scoped**. A variant is always scoped. Topics can be used to **reify** other topic map objects. Topic maps may be **merged**. One concrete syntax for topic maps is **LTM**.

14.10 Exercises

1. Construct a topic map from the wikipedia entry for the Athens Olympics, as cited in the exercise for Chapter 2. Use instances of all topic map constructs.
2. Construct a topic map equivalent of a substantial portion of the OWL ontology from the Chapter 11 exercise. Provide representations of all the different OWL constructs.
3. Comment on the differences between the OWL and topic maps representations.

14.11 Further Reading

The Topic Maps data model is being standardized by the ISO [29]. A primer on topic maps is Empolis [30]. The LTM has been published in [31]. The abstract syntax used comes from the Ontology Development Metamodel [22], [23].

15 Using an Ontology : the Ontology Server

We turn now to the software needed to support ontology development and use: the ontology server.

15.1 What Is an Ontology Server?

An ontology is a complex information object. Even the examples from Chapter 4 are complex enough that they would not be built in a single session, would benefit from graphical representations, and are difficult to check for completeness and consistency. Ontologies used in practice can contain millions of concepts in complex relationships. We have touched for example on the SNOMED ontology in Chapter 4.

When we want to manage complex information objects, we generally turn to information systems technology. There are many tens of millions of information systems in use around the world doing everything from helping a business manage its billing to keeping track of the design and parts inventory of large aircraft. An information system intended to manage an ontology is called an *ontology server*.

Information systems are generally built around a database core, since the complex information object being managed needs to be stored and desired parts of it retrieved. The database core of an ontology server is called an *ontology repository*.

Ontology server technology is at the time of writing quite immature. There are a number of research projects. We will refer to that at Stanford University (Ontolingua, Protégé), Free University of Brussels (Dogma) and the University of Karlsruhe (KAON). Because the field is immature there is not a body of routinely used comprehensive tools based on well-established principles. This Chapter therefore is necessarily somewhat speculative in nature. However, ontology servers are closely related to Computer-Aided Software Engineering (CASE) tools, which are a relatively mature technology.

CASE tools are generally used to support the design of a system. Database systems such as Oracle or DB2 are supported by tools which assist in data modeling, often using some variant of the Entity-Relationship system. They will assist in the construction of SQL Data Description Language (DDL) statements based on the conceptual model. They also assist in the construction of SQL Data Manipulation Language (DML) statements using forms, a Query-By Example (QBE) interface, or perhaps a natural language query interface.

Unlike CASE tools, an ontology server is used both during design phases and during the execution of the ontology-based applications.

- At design time the server assists the ontology engineers in the design and construction of the ontology. The three research prototypes are mostly confined to services at design time.

- At commit time, a player wishing to join an exchange needs to commit to the ontology, integrating part of their local conceptual model with at least part of the ontology. The ontology server can assist with this task.
- At run time, an ontology server can perform tasks like mediating the exchange of messages.

We will look at these requirements in more detail.

15.2 Design Time Requirements

Some of the use of ontology servers occurs at the time of design or modification of an ontology. This aspect is the closest to CASE, and is therefore best understood. Besides the information systems design tools associated with database products noted above, we have the suite of specifications developed by the Object Management Group (OMG) centered on UML, the Protégé ontology editor developed by Stanford University, and numerous other tools.

15.2.1 Editing Tools

A very basic requirement is a tool to enable ontology engineers to enter, modify, and browse a developing ontology. UML and ER tools such as Rational Rose use visual editors as interface to the model stored in a proprietary repository. Protégé uses a highly-structured forms-oriented interface, also with a proprietary repository. These tools need to be able to manage packages or other modules, to allow an ontology engineer to find components and view them in appropriate contexts, and to include documentation. Ontolingua/Protégé, Dogma and KAON all perform this function.

15.2.2 Certify an Ontology

Besides allowing the ontology to be created and modified, the ontology server should be able to certify that the ontology satisfies the syntactic constraints of the modeling language used. In some cases this is done by an analysis after the ontology has been entered, using analogs to compliers and syntax checkers. In other cases, especially with visual editors or forms-based editors, the tool does not permit a syntactically incorrect entry.

Where the syntactic rules are complex and operate over a large portion of the model it is often convenient for the server to permit an incorrect structure, while noting the errors present. This is sometimes called *soft integrity constraint*. It permits progressive entry and refinement of a complex structure. But the server is capable of certifying that an entered ontology is syntactically correct. Ontolingua/Protégé, Dogma and KAON all perform this function.

15.2.3 Manage Imported Ontology Modules

An ontology is a representation of the world in which a community of agents operates. There are many widely-used aspects of the world common to many applications. In Chapter 4 we have looked at several of these, including the Z39.50 system of information retrieval, the Standard Industrial Classification system, SNOMED, the

Periodic Table of the Elements and systems of engineering dimensions. In Chapter 10 we looked at the Dublin Core metadata ontology and standardized systems of literals like xsd. There are many more such ontologies published, and we can imagine that in the future there will be very many more. We would expect that national postal services would publish ontologies of postal districts, possibly through an international body such as the Universal Postal Union, and that the International Olympic Committee would publish ontologies of sports and sporting events, possibly collating them from the various international sporting federations.

So we would expect that as ontology development matures an ontology project would be able to assemble much of its content from standard modules, adding application-specific content. In terms of Chapter 9, if we were building an ontology to support run-time interoperation of an exchange, that we would import and possibly specialize numerous application generation ontologies. The ontology server needs to be able to support this.

One requirement is that the imported ontology must have its details visible but will have limitations on what can be changed within it. There is also the likelihood that the publisher of an ontology component will produce new versions from time to time, so the ontology server needs to be able to support version replacement.

15.2.4 Abstract Data Types and Metaproperties

Besides the material ontology components of the previous section, an ontology development will import formal ontology components of the kind described in Chapter 7. These formal ontologies involve more or less complex data structures and programs implementing operations on them, often called abstract data types. The ontology server will have a library of built-in abstract data types, but it would be an advantage to be able to import others, like the components described in Chapter 12.

Further, design rules and constraints like the OntoClean metaproperties of Chapter 6 must be supported by server facilities.

The server will also need to support the interaction of complex material ontological structures with complex formal structures, in a way similar to the multiple representations of the SIC ontology described in Chapter 4.

15.2.5 Version Control

Version control is important in any kind of software development. Besides imported ontological components having versions, the ontology developed will go through a number of versions during the course of its life. A characteristic of ontology development making version control more interesting is that an ontology can expect to have multiple versions co-existing.

For example, suppose we have an ontology supporting an e-commerce exchange in the book publishing industry. The management of the exchange decides that there is scope for electronic books, so develops a new version of the ontology to support this aspect of the trade. But the ontology will be used by possibly very many players, many of whom will not see any value in upgrading their practices to the new electronic publishing version. They will wish to continue doing business with the old version.

The ontology server will therefore need to keep track of multiple versions.

15.2.6 Publishing

In common with CASE tools generally, an ontology server will need facilities for publishing various aspects of the ontology in human-readable forms and also in computer-readable forms, including application program interfaces (API). In addition, it will need to be able to import ontology components in possibly several different computer-readable forms, converting the resulting structures into its internal repository. Ontolingua/Protégé, Dogma and KAON all perform this function.

15.3 Commit Time Requirements

As ontology supporting run-time interoperation exists outside of any of the participating systems, of which there can be tens of thousands. In order to participate in the interoperation, a potential player needs to commit to the ontology, which involves alignment of the conceptual model of the player's information system with the ontology. Commitment to the ontology is an active process, which may require the participant to change their representations, or even to change their way of doing business. Access to the ontology is essential during the commitment process. The ontology server must provide tools and facilities to assist.

15.3.1 Browsing Services

The participant will need to access the ontology. The ontology will likely be represented in a different system from that used for the participant's own information system. For example, the ontology may be represented in OWL or Topic Maps, while the information system may be modeled in ERA or UML. It would be advantageous if the server could present the ontology to the participant in a form as close as possible to the system used for the participant's own conceptual model. At the same time, it would be an advantage if the server could help the participant develop the schemas for the messages needed to interoperate with other participants. These messages are likely to be represented in a form closer to the ontology than to the participant's system, for example OWL/S. The message schemas must be linked both to the participant's conceptual model and to the ontology.

Mediation services between the ontology's and the participant's representations can either be direct (e.g. UML versus OWL) or via intermediate models, such as UML MOF metamodels of OWL as we have seen in Chapter 11. In the latter case, it might be possible to use tools derived from the OMG's Model-Driven Architecture standards. Mediation services include provision of APIs and method libraries.

15.3.2 Find Relevant Fragments of the Ontology

Aligning a participant's information system with the ontology and ways of interoperation of an exchange is a significant cost, since it will very likely require changes both in the participant's information systems and in the way the participant does business. It is likely that many exchanges will facilitate interoperation among a variety of different players none of whom makes use of the full spectrum of possibilities offered by the ontology.

For example, an e-commerce exchange might support a wide variety of goods and services relevant to say the building industry. The ontology will support a variety of types of building materials, subcontractors in a variety of trades, information services for regulations etc., and a variety of specialized services like architects, drafting of plans, inspection services, financial advice and legal advice.

Most of these players would use only a portion of the ontology. A plumbing subcontractor would not need facilities relevant to electricians or pest inspectors, for example. Therefore only certain classes of objects in an ontology are directly relevant to a given player. Given those classes, there may be other classes on which the required classes depend. A plumbing specification might depend on the local council who needs to inspect the work, and on the general contractor overseeing the overall job.

In OWL terminology (Chapter 11) class A *depends on* class B if A is in the domain of a property whose range is B, and the property has a minimum cardinality greater than 0. Existence of an instance of A therefore requires the prior existence of an instance of B. A *coherent fragment* of an ontology is a set of classes together with the transitive closure of dependent classes.

The ontology server will need to be able to assist a participant in finding a relevant minimal coherent fragment of the ontology, using the ontology's structural constraints.

15.3.3 Subscription Services

The ontology fragments the participant accesses will be subject to change. When a new version of the ontology is published, the ontology server will need to be able to decide which participants need to be informed of the changes. One reason a participant might need to know about a change is because the fragment of the ontology to which the participant subscribes changes. A second reason might be that a player might wish to be kept informed of changes in nominated areas of the ontology even though they do not at present make use of these aspects in their interoperations. So the ontology server will need to provide publish/subscribe services, and to be able to selectively disseminate information.

15.3.4 Multiple Natural Languages

Semantic web exchanges can easily involve players who use different natural languages. Even though there needs to be agreement among the players in the definitions of the classes of objects in their shared world and the structural relationships among them, it will often be convenient for different participants to work in their own natural languages but interoperate with participants working in other languages. So the ontology server will need to make use of multi-lingual term banks, and be able to generate statements in multiple languages.

15.4 Run Time Requirements

An ontology server will have a range of functions during the day-to-day interoperations facilitated by its exchange, so it will be needed at run time.

15.4.1 Maintain Directories of Players, Roles and Objects

Players and roles are objects in the ontology. Other objects are instances of classes in the ontology. In order for a participant to interoperate they need to know who the other players are, what roles they can play, and where other objects are. The ontology server is the natural provider of directory services.

15.4.2 Validate Messages

Applications communicate by exchange of messages. Within a single system, one application (say an inventory manager) can receive a message from another (say an order processor) and execute it. The receiving application does not need to worry whether the message is syntactically correct, nor whether it comes from a valid source. Syntax of messages and the valid senders and receivers of messages are all components of the specification of the system, and their correctness is one of the elements of the acceptance tests for deployment of the application.

If, however, an application receives a message from an application running in another organization, it would be unwise to assume the correctness of the message's syntax or the validity of its source. After all, there may be tens of thousands of players in an exchange and that all of them are behaving correctly is a pretty heroic assumption. So a message must be validated before the programs satisfying its request can be executed.

Message validation in this situation is complicated by the fact that there may be many versions of the ontology simultaneously in use among the various players.

It would be a major cost saving for all players to delegate the task of message validation to the ontology server as a trusted service provider. A message would be routed through the server, checked and certified as conforming to a given version of the ontology.

15.4.3 Broker Services

Complex interoperations among a number of players often require services from third parties to complete. Examples include

- holding an auction (eBay is an obvious example)
- banking services (PayPal is an obvious example)
- acting as a repository of complex requests for tender
- hosting a virtual enterprise, like a consortium responding to a complex tender

The ontology server is well placed to provide such services.

15.4.4 Archive Services

The ontology server is the obvious choice of an application maintaining an audit trail of changes to the ontology, changes to players and roles, and keeping archives of messages.

15.5 Structure of Ontology Server

An ontology server is itself an information system. It is built around a database containing the ontology together with other information needed for it to perform its services. It will therefore need a conceptual model from which its database schemas will be designed. Since its major content will be the ontology itself, it will make sense for the ontology server's conceptual model to be expressed in one of the systems already used to model the abstract syntax of an ontology representation language. We have seen in earlier Chapters that MOF, RDFS and Common Logic can all be used for this purpose. The repository can be implemented on a number of platforms, including relational database, object-oriented database or an RDF triple store.

The repository is a database, but its design is conditioned not so much by size as with conventional information systems, but by complexity. The database will need not only to execute queries like SQL, but will need to be able to navigate classification hierarchies like Yahoo!. It will also need a reasoning capability to support some of the server's functions. This can include graph processing capabilities for navigating the ontology, the description logics capability of testing whether a subclass definition predicate is consistent (satisfiability) and whether one subclass definition predicate subsumes another, and perhaps even a full predicate calculus theorem prover.

Finally, the ontology server will need to be able to translate to and from transport representations like XML in order to be able to send and receive complex structures.

15.6 Key Concepts

An ontology server is needed at **design time**, **commit time** and **run time**. It is a significant information system in its own right, built on a database of its own called a **repository**. The server will need **reasoning capability** beyond that generally required of information systems.

Bibliography

[1] T. Berners-Lee and M. Fischetti, Weaving the Web : the original design and ultimate destiny of the World Wide Web by its inventor, HarperSanFrancisco, San Francisco, 1999.

[2] J. Paul, S. Withanachchi, R. Mockler, M. Gartenfeld, W. Bistline and D. Dologite, Enabling B2B marketplaces: the case of GE Global Exchange Services, *Annals of Cases on IT*, Idea Group, 2003, 464-487.

[3] N. Guarino and M. Musen, Applied ontology: Focusing on content, *Applied Ontology* 1(1), 2006, 1-5.

[4] J. Searle, *Construction of social reality*, Free Press, New York, 1995.

[5] A.P. Sheth and J. Larsen, Federated database systems for managing distributed, heterogeneous and autonomous databases, *ACM Computing Surveys: Special issue on heterogeneous databases*, 22(3), 1990, 183-236.

[6] R. Stamper, Management epistemology: Garbage in garbage out, in L. R. Methlie and R. H. Sprague (eds), *Knowledge representation for decision support systems*, Elsevier Science Publishers B.V., Amsterdam, 1985, 55-77.

[7] R.M. Colomb, Impact of semantic heterogeneity on federating databases, *The Computer Journal,* 40(5), 1997, 235 -244.

[8] A. Gangemi, N. Guarino, C. Masolo and A. Oltramari, Understanding top-level ontological distinctions. In A. Gómez Pérez, M. Grunninger, H. Stuckenschmidt and M. Ushold (eds.) *Proceedings of IJCAI 2001 workshop on ontologies and information sharing*, Seattle, 2001, 26-33. Available at http://www.loa cnr.it/Papers/IJCAI2001ws.pdf

[9] N. Guarino and C. Welty, Evaluating ontological decisions with OntoClean, *Communications of the ACM*, 45(2), 2002, 61-65.

[10] N. Guarino and C. Welty, A formal ontology of properties, in R. Dieng and O. Corby (eds.), *Knowledge engineering and knowledge management: Methods, models and tools. 12th international conference, EKAW2000.* Springer Verlag, Berlin, 2000, 97-112, available at http://www.loa-cnr.it/Papers/EKAW-2000.pdf

[11] R. Weber, *Ontological foundations of information systems*, Coopers and Lybrand, Melbourne, 1997.

[12] C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari and L. Schneider, *The WonderWeb library of foundational ontologies preliminary report WonderWeb deliverable D17*, Laboratory for Applied Ontology, National Research Council of Italy, 2003. Available at http://www.loa-cnr.it/Papers/ DOLCE2.1-FOL.pdf

[13] J. Dewey, *Experience and Nature*, Allen & Unwin, London, 1929.

[14] T. R. Gruber, Toward principles for the design of ontologies used for knowledge sharing, *International Journal of Human-Computer Studies*, 43(5-6), 1995, 907-928. Available at http://tomgruber.org/ writing/onto-design.htm

[15] J. K. Debenham, *Knowledge Systems Design* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[16] L. Hart, P. Emery, R. Colomb, K. Raymond, D. Chang, Y. Ye, E. Kendall and M. Dutra, Usage scenarios and goals for ontology definition metamodel, in X. Zhou, S. Su, M. Papazoglou, M. Orlowska, and K. Jeffery (eds.), *Web information systems engineering conference (WISE'04)* Brisbane, Australia. Springer LNCS 3306, 2004, 596-607.

[17] *RDF Primer*, 2004. Available at http://www.w3.org/TR/2004/REC-rdf-primer-20040210/

[18] *OWL Overview*, 2004. Available at http://www.w3.org/TR/2004/REC-owl-features-20040210/

[19] *OWL Language Guide*, 2004. Available at http://www.w3.org/TR/2004/REC-owl-guide-20040210/

[20] *Ontology 101*, 2001. Available at http://ksl.stanford.edu/people/dlm/papers/ontology-tutorial-noy-mcguinness-abstract.html

[21] *Protégé*. Available at http://protege.stanford.edu/

[22] *Ontology development metamodel final adopted specification*. Combined response to OMG/RFP ad/2003-03-40. Object Management Group, 2007.

[23] R.M. Colomb, K. Raymond, L. Hart, P. Emery, C. Welty, G.T. Xie, and E.K. Kendall, The Object Management Group ontology definition metamodel, in C. Calero, F. Ruiz and M. Piattini (eds) *Ontologies for software engineering and software technology*, Springer, Berlin, 2006, 217-247.

[24] P. Simons, *Parts: A study in ontology*, Clarendon Press, Oxford, 1987.

[25] M.E. Winston, R. Chaffin and D. Herrmann, A taxonomy of part-whole relations, *Cognitive Science* 11, 1987, 417-444.

[26] *Common Logic*, ISO 24707-25-Nov-2006, 2006. Available at http://cl.tamu.edu/ docs/cl/24707-25-Nov-2006.pdf

[27] R.M. Colomb, C.N.G. Dampney and M. Johnson, Category-theoretic fibration as an abstraction mechanism in information systems, *Acta Informatica* 38, 2001, 1-44.

[28] T. R. Gruber and G. R. Olsen, An ontology for engineering mathematics, In J. Doyle, P. Torasso and E. Sandewall, (eds.), *Fourth International Conference on Principles of Knowledge Representation and Reasoning*, Gustav Stresemann Institut, Bonn, Germany, Morgan Kaufmann, San Mateo, Calif., 1994. Available at http://ksl.stanford.edu/knowledge-sharing/papers/engmath.html

[29] *The Topic Maps data model* is being standardized by the ISO under the identifier ISO/IEC FDIS 13250-2.

[30] *The Topic Maps Handbook,* Empolis, 2003. Available at <www.empolis.com/ downloads/ empolis_TopicMaps_Whitepaper20030206.pdf>.

[31] *Linear Topic Maps,* Ontopia, 2006. Available at <www.ontopia.net/download/ ltm.htm >.

Appendix A - SIC

U.S. Department of Labor
Standard Industrial Classification Division Structure - 1987 edition

Division A: Agriculture, Forestry, and Fishing
• Major Group 01: Agricultural Production Crops
• Major Group 02: Agricultural Production Livestock and Animal Specialties
• Major Group 07: Agricultural Services
• Major Group 08: Forestry (See below for further expansion)
• Major Group 09: Fishing, Hunting, and Trapping
Division B: Mining
• Major Group 10: Metal Mining
• Major Group 12: Coal Mining
• Major Group 13: Oil and Gas Extraction
• Major Group 14: Mining and Quarrying of Nonmetallic Minerals, Except Fuels
Division C: Construction
• Major Group 15: Building Construction General Contractors and Operative Builders
• Major Group 16: Heavy Construction Other Than Building Construction Contractors
• Major Group 17: Construction Special Trade Contractors
Division D: Manufacturing
• Major Group 20: Food and Kindred Products
• Major Group 21: Tobacco Products
• Major Group 22: Textile Mill Products
• Major Group 23: Apparel and other Finished Products Made From Fabrics and Similar Materials
• Major Group 24: Lumber and Wood Products, Except Furniture
• Major Group 25: Furniture and Fixtures
• Major Group 26: Paper and Allied Products
• Major Group 27: Printing, Publishing, and Allied Industries
• Major Group 28: Chemicals and Allied Products
• Major Group 29: Petroleum Refining and Related Industries
• Major Group 30: Rubber and Miscellaneous Plastics Products
• Major Group 31: Leather and Leather Products
• Major Group 32: Stone, Clay, Glass, and Concrete Products
• Major Group 33: Primary Metal Industries
• Major Group 34: Fabricated Metal Products, Except Machinery and Transportation Equipment
• Major Group 35: Industrial and Commercial Machinery and Computer Equipment
• Major Group 36: Electronic and Other Electrical Equipment and Components, Except Computer Equipment
• Major Group 37: Transportation Equipment
• Major Group 38: Measuring, Analyzing, and Controlling Instruments; Photographic, Medical and Optical Goods; Watches and Clocks

• Major Group 39: Miscellaneous Manufacturing Industries

Division E: Transportation, Communications, Electric, Gas, and Sanitary Services

• Major Group 40: Railroad Transportation
• Major Group 41: Local and Suburban Transit and Interurban Highway Passenger Transportation
• Major Group 42: Motor Freight Transportation and Warehousing
• Major Group 43: United States Postal Service
• Major Group 44: Water Transportation
• Major Group 45: Transportation by Air
• Major Group 46: Pipelines, Except Natural Gas
• Major Group 47: Transportation Services
• Major Group 48: Communications
• Major Group 49: Electric, Gas, and Sanitary Services

Division F: Wholesale Trade

• Major Group 50: Wholesale Trade-durable Goods
• Major Group 51: Wholesale Trade-non-durable Goods

Division G: Retail Trade

• Major Group 52: Building Materials, Hardware, Garden Supply, and Mobile Home Dealers
• Major Group 53: General Merchandise Stores
• Major Group 54: Food Stores
• Major Group 55: Automotive Dealers and Gasoline Service Stations
• Major Group 56: Apparel and Accessory Stores
• Major Group 57: Home Furniture, Furnishings, and Equipment Stores
• Major Group 58: Eating and Drinking Places
• Major Group 59: Miscellaneous Retail

Division H: Finance, Insurance, and Real Estate

• Major Group 60: Depository Institutions
• Major Group 61: Non-depository Credit Institutions
• Major Group 62: Security and Commodity Brokers, Dealers, Exchanges, and Services
• Major Group 63: Insurance Carriers
• Major Group 64: Insurance Agents, Brokers, and Service
• Major Group 65: Real Estate
• Major Group 67: Holding and Other Investment Offices

Division I: Services

• Major Group 70: Hotels, Rooming Houses, Camps, and Other Lodging Places
• Major Group 72: Personal Services
• Major Group 73: Business Services (See below for further expansion)
• Major Group 75: Automotive Repair, Services, and Parking
• Major Group 76: Miscellaneous Repair Services
• Major Group 78: Motion Pictures
• Major Group 79: Amusement and Recreation Services
• Major Group 80: Health Services
• Major Group 81: Legal Services
• Major Group 82: Educational Services
• Major Group 83: Social Services
• Major Group 84: Museums, Art Galleries, and Botanical and Zoological Gardens
• Major Group 86: Membership Organizations

• Major Group 87: Engineering, Accounting, Research, Management, and Related Services
• Major Group 88: Private Households

Division J: Public Administration

• Major Group 91: Executive, Legislative, and General Government, Except Finance
• Major Group 92: Justice, Public Order, and Safety
• Major Group 93: Public Finance, Taxation, and Monetary Policy
• Major Group 94: Administration of Human Resource Programs
• Major Group 95: Administration of Environmental Quality and Housing Programs
• Major Group 96: Administration of Economic Programs
• Major Group 97: National Security and International Affairs
• Major Group 99: Nonclassifiable Establishments

SIC Major Group 08: Forestry

This major group includes establishments primarily engaged in the operation of timber tracts, tree farms, forest nurseries, and related activities such as reforestation services and the gathering of gums, barks, balsam needles, maple sap, Spanish moss, and other forest products.

Industry Group 08 1: Timber Tracts
    0811 Timber Tracts
Industry Group 083: Forest Nurseries and Gathering of Forest
    0831 Forest Nurseries and Gathering of Forest Products
Industry Group 085: Forestry Services
    0851 Forestry Services

SIC Major Group 73: Business Services

This major group includes establishments primarily engaged in rendering services, not elsewhere classified, to business establishments on a contract or fee basis, such as advertising, credit reporting, collection of claims, mailing, reproduction, stenographic, news syndicates, computer programming, photocopying, duplicating, data processing, services to buildings, and help supply services. Establishments primarily engaged in providing engineering, accounting, research, management, and related services are classified in Major Group 87. Establishments which provide specialized services closely allied to activities covered in other divisions are classified in such divisions.

Industry Group 731: Advertising
- 7311 Advertising Agencies
- 7312 Outdoor Advertising Services
- 7313 Radio, Television, and Publishers' Advertising Representatives
- 7319 Advertising, Not Elsewhere Classified

Industry Group 732: Consumer Credit Reporting Agencies, Mercantile
- 7322 Adjustment and Collection Services
- 7323 Credit Reporting Services

Industry Group 733: Mailing, Reproduction, Commercial Art and
- 7331 Direct Mail Advertising Services
- 7334 Photocopying and Duplicating Services
- 7335 Commercial Photography
- 7336 Commercial Art and Graphic Design
- 7338 Secretarial and Court Reporting Services

Industry Group 734: Services to Dwellings and Other Buildings
- 7342 Disinfecting and Pest Control Services
- 7349 Building Cleaning and Maintenance Services, Not Elsewhere

Industry Group 735: Miscellaneous Equipment Rental and Leasing
- 7352 Medical Equipment Rental and Leasing
- 7353 Heavy Construction Equipment Rental and Leasing
- 7359 Equipment Rental and Leasing, Not Elsewhere Classified

Industry Group 736: Personnel Supply Services
- 7361 Employment Agencies
- 7363 Help Supply Services

Industry Group 737: Computer Programming, Data Processing, and
- 7371 Computer Programming Services
- 7372 Prepackaged Software
- 7373 Computer Integrated Systems Design
- 7374 Computer Processing and Data Preparation and Processing Services
- 7375 Information Retrieval Services
- 7376 Computer Facilities Management Services
- 7377 Computer Rental and Leasing
- 7378 Computer Maintenance and Repair
- 7379 Computer Related Services, Not Elsewhere Classified

Industry Group 738: Miscellaneous Business Services
- 7381 Detective, Guard, and Armored Car Services
- 7382 Security Systems Services
- 7383 News Syndicates
- 7384 Photofinishing Laboratories
- 7389 Business Services, Not Elsewhere Classified

Appendix B - Airline Ontology in RDF

air:Airline	rdf:type	rdfs:Class
air:USAirline	rdf:type	rdfs:Class
air:USAirline	rdfs:subClassOf	air:Airline
air:EUAirline	rdf:type	rdfs:Class
air:EUAirline	rdfs:subClassOf	air:Airline
air:InternationalAirline	rdf:type	rdfs:Class
air:InternationalAirline	rdfs:subClassOf	air:Airline
air:QANTAS	rdf:type	air:InternationalAirline
air:Flight	rdf:type	rdfs:Class
air:International	rdf:type	rdfs:Class
air:International	rdfs:subClassOf	air:Flight
air:Domestic	rdf:type	rdfs:Class
air:Domestic	rdfs:subClassOf	air:Flight
air:QF1	rdf:type	air:International
air:operates	rdf:type	rdf:Property
air:operates	rdfs:domain	air:Airline
air:operates	rdfs:range	air:Flight
air:QANTAS	air:operates	air:QF1
air:depart_time	rdf:type	rdf:Property
air:arrive_time	rdf:type	rdf:Property
air:depart_time	rdfs:domain	air:Flight
air:depart_time	rdfs:range	xsd:time
air:arrive_time	rdfs:domain	air:Flight
air:arrive_time	rdfs:range	xsd:time
air:QF1	air:depart_time	“08:15:00+08:00”
air:QF1	air:arrive_time	“22:10:00-09:00”
air:City	rdf:type	rdfs:Class
air:USCity	rdf:type	rdfs:Class
air:USCity	rdfs:subClassOf	air:City
air:EUCity	rdf:type	rdfs:Class
air:EUCity	rdfs:subClassOf	air:City
air:OtherCity	rdf:type	rdfs:Class
air:OtherCity	rdfs:subClassOf	air:City
air:origin	rdf:type	rdf:Property
air:origin	rdfs:domain	air:Flight
air:origin	rdfs:range	air:City
air:destination	rdf:type	rdf:Property
air:destination	rdfs:domain	air:Flight
air:destination	rdfs:range	air:City
air:Sydney	rdf:type	air:OtherCity
air:LosAngeles	rdf:type	air:USCity
air:QF1	air:origin	air:Sydney
air:QF1	air:destination	air:LosAngeles
air:name	rdf:type	rdf:Property
air:name	rdfs:domain	air:City
air:name	rdfs:range	xsd:string
air:airport	rdf:type	rdf:Property
air:airport	rdfs:domain	air:City
air:Sydney	air:name	“Sydney”
air:Sydney	air:airport	iata:SYD
air:LosAngeles	air:name	“Los Angeles”
air:LosAngeles	air:airport	iata:LAX

Appendix C - Explanation of Some Technical Concepts

Equivalence Relation

Assume we have a collection of constants *a*, *b*, *c*, … and relations *R*, *S*, *T*, …. An equivalence relation *R* is a binary relation which is

- reflexive – if a is a constant, then R(a, a)
- symmetric – if R(a, b) then R(b, a)
- transitive – if R(a, b) and R(b, c) than R(a, c)

An equivalence relation *partitions* the set of constants (divides it into disjoint subsets), so that if we have the set C of constants {a, b, c, d, e} and R(a, b), R(b, c) and R(d, e), then C is partitioned by R into the subsets {a, b, c} and {d, e}. Different relations will generally give different partitions.

Lexical/Logical

Data in a database is stored in a set of tuples in a named relation. Tuples in the relation are organized by a schema, which a collection of named attributes. An individual tuple consists of a collection of values of the attributes for the named relation. A query on a relation, expressed as an SQL statement or equivalently as an existentially quantified logical formula, has a result which is a collection of values of the various attributes which appear in tuples which satisfy the selection predicate associated with the query.

Information is said to be represented logically if it can be expressed as a collection of values of attributes in the result of a query (in logic, as the bindings of variables in a predicate). Information is said to be represented lexically if it needs the names of the relations or the names of the attributes or the grouping of attributes in a relation. Lexically represented information is the form, while logically represented information is the content.

Consider the example below.

Relation name: Enrolment

Attributes	StudentID	CourseID	Grade
	123456	PHIL1200	5
	123456	INFS1200	7
	654321	COMP1500	6

Query: SELECT * from Enrolment where Grade = 6

Result:

Attributes	StudentID	CourseID	Grade
	654321	COMP1500	6

The name of the relation (*Enrolment*), the names of the attributes (*StudentID*, *CourseID*, *Grade*) and the fact that the attributes are all associated with the relation named *Enrolment* are all lexically represented information. So also is the fact that for example the constant 123456 is associated with the *StudentID* attribute in the

Enrolment relation as distinguished from the *Grade* attribute. What is represented logically is the actual values associated with the attributes in the table, so that the result of the query is represented logically.

The lexical/logical distinction is important when considering agents, since the program implementing the agent can only find out information represented logically. The lexical information is fixed as part of the program. In our example, the program fragment is the SQL query, a lexical object. It can only find out about the values of the attributes named in the query which are stored in the relation named in the query. It can't discover new relations or new attribute names, but it can discover for example associations between course code and grade.

Information represented lexically must be hard-wired into the program, which can learn only about information represented logically.

Types

Type is a very common term in the theory of computing used extensively in this book. *Type* is associated with *instance*. A type is a set of which the associated instances are members. The term occurs in programming languages, where a variable is declared to be of a type, so that any instantiation of the variable must be an instance of the type. Common types here are integers, strings, reals, and characters. Programming languages generally have the facility of declaring complex structures as types such as arrays. The instances of the type *integer* are all the positive and negative integers which can be represented with the number of bits built into the programming language definition, typically 32 or 64.

The term *type* is used in logic to limit the range of values a variable can be quantified over. The definition of a logic generally includes a set of constants. In a typed logic, the set of constants is divided into subsets often designated by unary predicates (called *type predicates*). In logical theories, the types often have semantic significance. We might have a predicate which declares that every course must have a lecturer assigned to it, say lecturer-assigned(Course, Lecturer). In an untyped logic this would be expressed as

(For all) Course (There Exists) Lecturer lecturer-assigned(Course, Lecturer)

Recall that the names of variables are arbitrary, so that this is equivalent to

(For all) X (There Exists) Y lecturer-assigned(X, Y)

Both formulas would be satisfied by the instance *lecturer-assigned(brisbane, 42)*, which might not make semantic sense.

In a typed logic, there would be two type predicates *lecturer(X)* and *course(X)*, whose instances would be respectively all and only the lecturers and courses known to the system. The predicate in question would be now

(For all) X (There Exists) Y lecturer(X) and course(Y) and lecturer-assigned(X, Y)

Types are at the core of data modeling systems. In the Entity-Relationship model, an *entity* is formally an *entity type*. In UML a type is called a *class*. In object-role modeling, the objects are organized as *object types*. The semantics of these systems is essentially that of typed logics.

Mereology

Mereology is the study of the whole-part relationship. Its basic predicate is

1. P(x, y) is true if x is a part of y
   Which is subject to the following axioms
2. P(x, x) x is a part of itself
3. P(x, y) and P(y, x) -> x = y. Two things can't be parts of each other.
4. P(x, y) and P(y, z) -> P(x, z). Parthood is transitive.
   We generally use as well the predicate PP or proper part
5. PP(x, y) -> $\exists$z(P(z, y) and not Overlap(z, x))
   where
6. Overlap(x, y) $=_{df}$ $\exists$z(P(z, x) and P(z, y))
   which implies that
7. not PP(x, x) Something cannot be a proper part of itself.
   An object is *atomic* if it has no proper parts.

The *mereological sum* of x and y ($x + y$) is the smallest thing that has both x and y as parts. If x is a part of y then x + y = y.

Appendix D - Conceptual Modeling Languages

This text assumes that the reader will be familiar with one of the information systems conceptual modeling languages. The main such languages are Entity-Relationship modeling (ERA), Unified Modeling Language (UML) and Object-Role Modeling (ORM). We summarize the main concepts from these languages for reference purposes, illustrating by representing the same specification in each. Our consideration of UML is limited to the Classes Model.

All the languages are based on the mathematical theory of sets and relations.

A conceptual model is a representation of objects in the environment of an application with which the programs will ultimately interact. These objects are collectively called the Universe of Discourse. The Universe of Discourse used to show the main features of the three modeling languages is taken from a University application, where students enroll in courses and are assigned grades. A student has an identifier and a name. A course is identified by a code, has a description, and can report the number of students enrolled. Associated with each enrolment is a staff member as instructor (there may be several instructors for a given course). A staff member is identified by an ID, and has a name.

Each of the systems is only sketched. Each system has many more facilities. For full details, the reader should consult more specialized texts.

UML Classes Model

Figure 43 shows a UML Classes model of the universe. There are three ***classes*** (*Course*, *Student* and *Staff*). Each class has attributes (*Course* has *code*, *description* and *numEnrolled*; *Student* has *ID* and *name*, *Staff* has *ID* and *name*). The attributes all take values which are strings. A class is represented by a collection of tuples of attribute values, which represents the corresponding collection of objects in the universe.

That students are enrolled in courses is represented by the ***association*** *enrolled*. A (binary) association is a collection of pairs of instances of the classes at its ends. An association can itself be a class (***association class***), so can itself have attributes. The association *enrolled* has the attribute *grade*. An association class can be an end of an association. In this case, *enrolled* is one end of the association *instructor*, whose other end is *Staff*. This association represents that a staff member is associated with a number of student enrolments.

Associations can be constrained in a number of ways, most prominently by ***multiplicities***, both upper and lower. *Enrolled* has lower multiplicity 0 and upper multiplicity n at both ends. The lower multiplicities signify that there may be a course with no students enrolled (lower multiplicity at the *Student* end), and that there may be a student enrolled in no course (lower multiplicity at the *Course* end). The upper multiplicity n signifies that a course may have an arbitrary number of students enrolled (*Student* end) and that a student may be enrolled in an arbitrary number of courses

(*Course* end). The multiplicity of 1 at the *Staff* end of *instructor* signifies that an instance of *enrolment* is associated with exactly one instance of *Staff*.

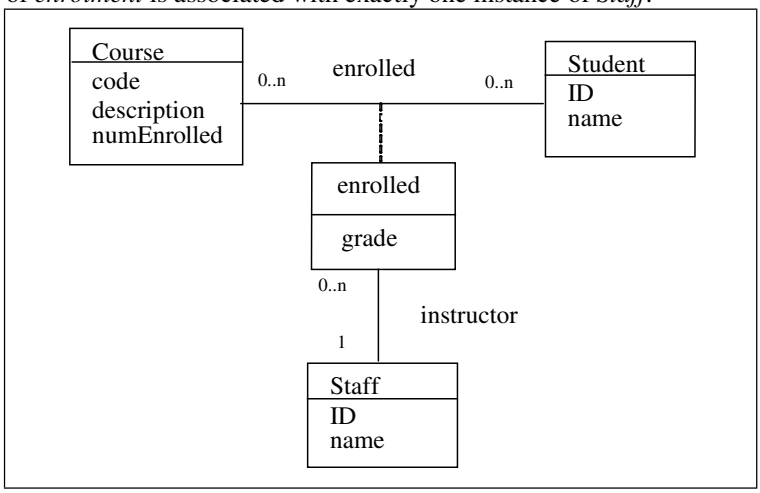


Figure 43. Example UML Classes model

Entity-Relationship Model

Figure 44 shows the same universe modeled using the Entity-Relationship method. ER is very widely used, and there are many dialects with different facilities and notations. Not all dialects will permit the model in Figure 44, and there are many different renderings of those which do.

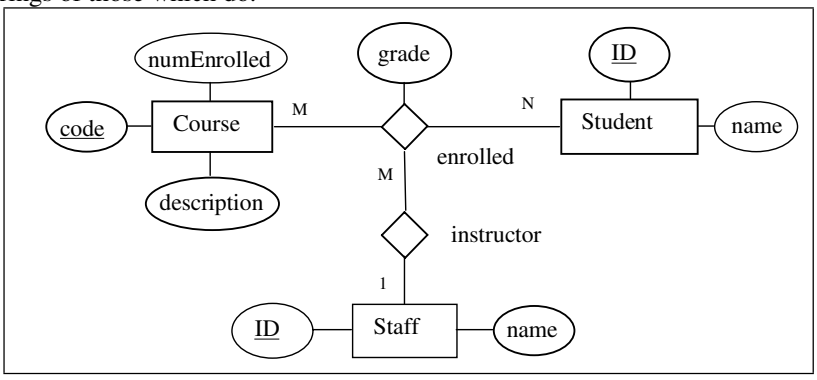


Figure 44. Example ERA diagram

The course, student and staff objects in the universe are represented by the ***entity types*** *Course*, *Student* and *Staff*. Entity types in ER are very similar to ***classes*** in UML. What are modeled as ***attributes*** in UML are modeled in a similar way in ER, by what are also called ***attributes***. The ***associations*** *enrolled* and *instructor* from the UML model are represented in the ER model by the similar construct ***relationship type***. The ends of a relationship are called ***roles***.

ER modeling has a facility called ***cardinality ratio*** that works much like the UML multiplicity facility. In Figure 44 the relationship *enrolled* has the *Course* end labeled

M and the *Student* end labeled N. M at the *Course* end signifies that an instance of *Student* can be linked to an arbitrary number of instances of *Course*. N at the *Student* end signifies that an instance of *Course* can be linked to an arbitrary number of instances of *Student*. A binary relationship labeled this way is called ***many-to-many***. In the *instructor* relationship, the *Staff* end is labeled 1. This signifies that an instance of *enrolment* (a student-course pair) is associated with at most one instance of *Staff*. A binary relationship labeled with M at one end and 1 at the other is called ***many-to-one***. The cardinality ratio corresponds very closely to UML upper multiplicity. ER has also the concept that a role can be either ***mandatory*** or ***optional***, corresponding to a minimum multiplicity of 1 or 0 respectively. Note that there are other notations in ER modeling for this kind of purpose.

In UML an association can be designated as an association class. Correspondingly, in ER a many-to-many relationship is in effect an association class. It can have attributes and in some formulations can play a role in a relationship. In Figure 44, *enrolled* has the attribute *grade* and participates in the relationship *instructor* with *Staff*.

A further facility of ER modeling illustrated in Figure 44 is the concept of ***identifier***. Each of the entities has one attribute underlined. This signifies that the attribute is never null, and that distinct instances of the entity have distinct values of that attribute. A value of the attribute therefore identifies an instance of the entity. UML does not support the concept of identifier.

Object-Role Modeling

An Object-Role Modeling view of the universe is shown in Figure 45.

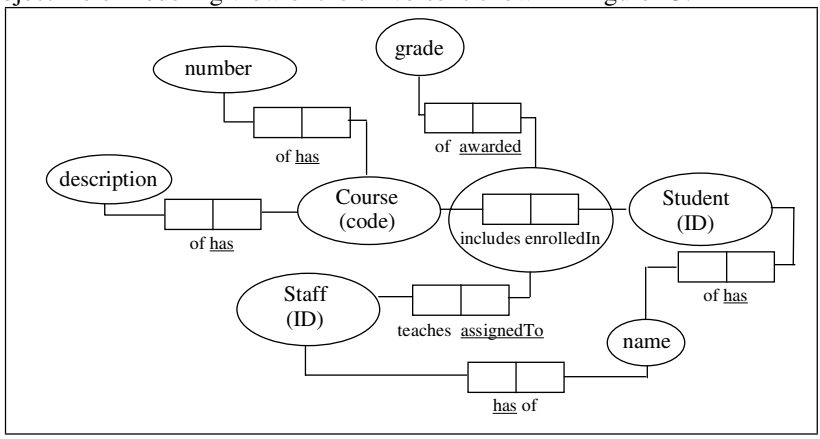


Figure 45. Example of an Object-Role Model

Course, student and staff from the universe are represented in Figure 45 as ***object types***, in ovals. The other object types: *description*, *number*, *grade* and *name*, are all what in UML or ER are value sets for attributes. ORM represents them both in a uniform way. What in UML is represented as an association and in ER as a relationship type is represented in ORM as a fact type. What UML and ER represent as attributes are represented in ORM also as fact types. All the fact types in Figure 45 are binary, but they can be unary or have arity greater than two. The two ends of a (binary) fact

type are called ***roles***, and are labeled. A fact type is intended to be expressed in natural language. For example, *Student* is *enrolledIn Course* or *Staff has name*. Note that the ends of UML associations or the roles in ER relationship types can similarly be named. However, the conventions are different. ER names roles the same as ORM, but in UML the ends are named in reverse. So the end of the association *enrolled* attached to *Course* would be called *enrolledIn*, while the end attached to *Student* would be called *includes*.

A fact type can be what is called ***reified***, so be treated as a fact type. This is shown by enclosing the fact type in an oval, as the oval enclosing *includes/enrolledIn*. A reified fact type can play a role in a further fact type, so ORM has the equivalent of an association class.

ORM includes the concept of ***identifier***. The object types *Course*, *Student* and *Staff* have respectively *code*, *ID* and *ID* in parentheses within their ovals. This is a shorthand notation for object types which play identifying roles for other object types. More generally, ORM has the concept of a ***uniqueness constraint***, indicated by an underlined role in a fact type. An instance of an object type can appear only once in a fact type where its role has a uniqueness constraint. So an instance of *Course has* at most one *description*, and instances of *Staff* and *Student have* only one *name*. Like ER, ORM has the concepts of mandatory and optional roles, not shown in Figure 45. There are many other kinds of integrity constraints that can be shown on an ORM diagram.

Appendix E - Logical Integrity Constraints

An integrity constraint is a predicate which is meant to be true for any valid population of a logical theory. A wide range of integrity constraints can be expressed in the predicate calculus, as in Chapter 13. This note is intended to clarify how the logic works to express the constraints. It will use the syntax of Common Logic from Chapter 13.

For concreteness, assume we have a simple ontology of *Course*, *Lecturer* and *School* as shown in Figure 46. There is one property *assigned* whose domain is *Course* and whose range is *Lecturer.* School is a free-standing class not connected to the others at this stage. (Assume that the ontology is not yet complete.) We will further assume that the extents of the classes are as follows

- *Course*: INFS3101, INFS3200, INFS1200
- *Lecturer*: Colomb, Orlowska, Sadiq
- *School*: Computing, Mathematics

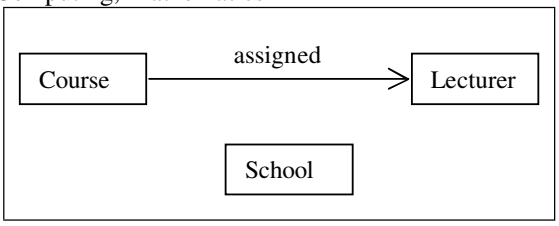


Figure 46. Simple ontology

We want the property *assigned* to be total, that is every course is assigned to someone. This integrity constraint would be expressed by (139)

```
(forall course (implies                                    (139)
    (Course course)
    (exists lecturer (assigned course lecturer))
))
```

Assume the extent of assigned is as shown in Table 8. We can see that the integrity constraint is satisfied. We will now work through the logic in detail to see how the constraint is expressed.

Table 8. Valid extent of *assigned*

Course	Lecturer
INFS3101	Colomb
INFS3200	Orlowska
INFS1200	Sadiq

Constraint (139) begins with a universally quantified variable *course*. Remember that even though the variable has an evocative name, that in the logic it is simply a placeholder which can represent any of the constants in the system. The constraint must be true no matter whether *course* is instantiated as INFS3101, INFS3200, INFS1200, Colomb, Orlowska, Sadiq, Computing or Mathematics. Let us see what happens if *course* is instantiated to Mathematics.

In this case, the second line of E1 comes into play, becoming (Course Mathematics). What this says is that one of the members of the extent of the class *Course* is the constant Mathematics. This is clearly not true, so the predicate (Course Mathematics) is *false*.

Now the connector *implies* in the first line comes into play. We know the truth table for implication. A predicate (implies p q) is *true* just in case the consequent *q* is *true* or the antecedent *p* is *false*. In our case, the antecedent is *false*, since Mathematics is not an instance of *Course*. This makes the predicate (139) *true* for *course* instantiated to Mathematics, without having to worry about the third line. The predicate (Course course) in the second line is a type predicate acting as a sort of gatekeeper for the third line. We only need to worry about the third line for instantiations of *course* for which (Course course) is *true*.

Assume now we have instantiated *course* to INFS3101. The third line of (139) says that there is some constant such that the pair <INFS3101, that constant> is in the extent of *assigned*. Concretely, the pair is one of the rows of Table 8. If we choose to instantiate the variable *lecturer* to Colomb, we find <INFS3101, Colomb> to be one of the rows in the table, so the predicate in the third line, the consequent of the implication, is also *true*. Since both the antecedent and consequent are *true*, predicate (139) is *true* for *course* instantiated to INFS3101. The predicate is similarly *true* if we instantiate *course* to INFS3200 or INFS1200. So if the predicate is true if *course* is instantiated to each constant in the extent of *Course*, and also true for every constant not in the extent of *Course*, the integrity constraint is satisfied.

Now consider an extent of *assigned* which violates the integrity constraint, shown in Table 9.

Table 9. Invalid extent of *assigned*

Course	Lecturer
INFS3200	Orlowska
INFS1200	Sadiq

Now, the constant INFS3101 is in the extent of *Course*, but for that instantiation of *course*, we fail to find a constant such that <INFS3101, that constant> is in the extent of *assigned*. This makes the existentially quantified formula in the third line of (139) *false*. As this is the consequent in the implication whose antecedent is *true*, the implication is *false*. Since the implication is not *true* for every possible instantiation of course, the integrity constraint is *false*, and that state of the extents of the predicates is invalid.

We might be tempted to express the integrity constraint as in (140), using *and* instead of *implies*. After all, we want all instantiations of *course* to satisfy both of the conjuncts. Looking at the detail of what happens as we did above, we first instantiate *course* to one of the members of the extent of *Course*, say INFS3101, and we find that the formula is indeed satisfied. But for the universally quantified formula to be *true*, we need it to be *true* for instantiations of *course* not in the extent of *Course*, for example Mathematics. In this case, the first conjunct is now *false*. It takes only one conjunct to be *false* to make the conjunction *false*, so the integrity constraint fails.

```
(forall course (and                                    (140)
    (Course course)
    (exists lecturer (assigned course lecturer))
))
```

Formulation (140) looks attractive because we have used the information technology convention of giving variables evocative names. If we had used *X* and *Y* as

variables instead of *course* and *lecturer*, as a mathematician might, we would be less attracted to the formula (140). We must remember that the evocative name evokes its intended meaning only to the human analyst, not to the computerized logical system. To the logical system, a variable gets its meaning solely from the predicates it is used in.

To reinforce the understanding of the workings of integrity constraints, we will perform a similar analysis on a related constraint, namely that the property *assigned* is functional. Each instance of the domain is associated with at most one instance of the range. A Common Logic expression of this constraint is given by (141)

```
(forall course (implies                                          (141)
    (Course course)
    (implies
        (exists lecturer (assigned course lecturer))
        (implies
            (forall lect (assigned course lect))
            (= lect lecturer)
        )
    )
))
```

This constraint is expressed by three nested implications. The first antecedent in line 2 filters out instantiations of *course* not in the extent of *Course*. The second antecedent in line 4 selects an instantiation of *lecturer* associated with an instance of *Course*. We use implication rather than conjunction because the extent of *assigned* does not need to include every instance of *Course* as in Table 9 where there is no row containing INFS3101. It could even be empty. So the antecedent might be *false*. The third antecedent in line 6 filters out instantiations of *lect* which are not associated with the instance of *course* under consideration (e.g. *lect* = Mathematics). Finally, the third consequent makes sure that the property is functional by being *false* if more than one instantiation of *lect* is found satisfying the antecedent. In this case, the formula at lines 5 – 8 is *false*. This formula is the consequent of the second antecedent, so the formula at lines 3 – 9 is *false*. This formula is in turn the consequent of the first antecedent, so the integrity constraint as a whole is *false*.

Our whole story so far presumes that the extents of the predicates are all definitive, that is satisfy the closed world assumption. If a constant *c* is not in the concrete extent of *Course*, then the predicate (Course c) is *false*. This form of negation is called negation as failure. A formula is taken to be *false* if it cannot be proved *true*. Most information systems make the closed world assumption.

But as we have seen in Chapter 11, the semantic web language OWL makes the opposite assumption to closed world, namely the open world assumption. In this case negation as failure does not apply. Integrity constraints behave quite differently in an open world. In a closed world, the concrete extents of the predicates are taken as definitive, so if an integrity constraint is not satisfied it is taken to fail. In an open world, the integrity constraint is taken as definitive. If a predicate does not have an instance in its concrete extension satisfying the constraint, one is assumed to exist but is taken to be unknown.

For example, we have seen that integrity constraint (139) might fail because the extent of *assigned* does not contain an instance of *Lecturer* associated with a particular instance of *Course*. In table 9 there is no instance of *Lecturer* associated with INFS3101. But if the integrity constraint is definitive, then the formula

```
(exists lecturer (assigned INS3101 lecturer))                    (142)
```

would be interpreted as something like "There is a lecturer assigned to INFS3101, but we don't know who they are".

This leads to the problem of how we can talk about this unknown lecturer. We need some sort of name. In the predicate calculus, it is conventional to replace existentially quantified variables with a special kind of constant, called a Skolem constant, which fails to satisfy the unique names assumption.

This is of course exactly what a name in OWL is, by default. So if we are working in OWL, we could refer to this existing but unknown lecturer with a blank node. Later, if we discovered who this lecturer is, we could declare the discovered name to be owl:sameAs the blank node.

Turning finally to constraint (141), that *assigned* is functional, in an open world. Suppose we find two instantiations of *lect* satisfying the antecedent. Instead of failing the integrity constraint, the reasoner would conclude that the two instantiations were linked in an owl:sameAs relationship. Only if the reasoner could conclude from some other source that the two were also linked in an owl:differentFrom relationship would a contradiction exist. Literals by definition satisfy the unique name assumption, so an OWL datatype property will fail to be functional if an instance of its domain is associated with more than one instance of its range.

Solutions to Exercises

Chapter 2

1. Consider the following examples of institutional facts. What are the corresponding brute facts? In what context does the brute fact count as the institutional fact (consider both framing rules and some of the background)? What is the speech act of which the institutional fact is a record, and who makes it?

- An Australian 1 dollar coin
  - Brute fact: metal disk with patterns embossed on it.
  - Context: Buying and selling small items in Australia
  - Speech act: Declaration of coin as money by Australian government.
- Your drivers licence
  - Brute fact: Piece of paper with printing on it.
  - Context: Your having passed a test of driving ability given by a government agency.
  - Speech act: Declaration of your having passed test by the agency.
- Your citizenship
  - Brute fact: Piece of paper (identity card or passport) with printing on it.
  - Context: Your having been born in the country or satisfied requirements for naturalization.
  - Speech act: Registration of your birth by a government agency or acceptance of your having satisfied naturalization requirements by a government agency.
- How you are legally able to reside here to study in this University
  - Brute fact: Piece of paper (identity card or passport) with printing on it.
  - Context: Your having been born in the country or satisfied requirements for a study visa.
  - Speech act: Registration of your birth by a government agency or acceptance of your having satisfied visa requirements by a government agency.
- A goal in soccer.
  - Brute fact: ball passes through opening into net
  - Context: ball having been validly kicked by an on-side player during one of the valid periods of a game.
  - Speech act: declaration of a goal by the designated referee of the match.

2. Consider a hypothetical Olympics, sketched below. You can add details from your general knowledge or the result of research into the Olympics.

a) Identify three distinct players who interoperate using this ontology, and three distinct roles that can be taken in the interoperation. Briefly describe what each role does and how the players interact using the ontology. Show three concrete

actions taken by players as they interoperate. Include details of players and roles involved, and the contents of the messages.

Players: a swimmer, a swimming official, LOC, IOC, FINA.

Roles: competitor (competes in races), race official (makes sure competitors compete according to the rules, declares final positions), race organizer (assigns swimmers to heats and finals, assigns lanes, organizes venues, assigns officials, keeps records of results), Olympics organizer (establishes sport organizing subcommittees, registers competitors and officials, builds venues, deals with broadcasters, issues tickets), Olympics governing body (decides city for an Olympics, decides sports to be included, decides procedures for registration of competitors and officials, decides drug testing regimes, keeps track of Olympic records), sport governing body establishes rules and procedures for specific events, keeps world records.

Concrete actions:

- *competitor* registers with *race organizer* to compete in a specific event. Provides personal and team identification and specific event
- *race organizer* assigns heat/final lane to *competitor*. Provides heat/final identifier and lane identifier.
- *race organizer* assigns *race official* to heat/final. Provides heat/final identifier.
- *race official* notifies *race organizer* of results of heat/final. Provides heat/final identifier, competitor identifiers and respective positions.

b) Describe three institutional facts created in this network, including brute fact and context. What roles are responsible for creation of each fact? Who is responsible for keeping the definitive record of the fact?

- A heat/final has been conducted. Brute fact: a group of people met at a particular place and time and behaved in certain ways. Context: heat/final had been organized by the race organizer for that place and time. Record kept by race organizer.
- A competitor competed in a heat/final. Brute fact: the person was in the group meeting at the particular time and place and behaved in a certain way. Context: person was registered with the race organizer to compete in that heat/final, was not declared by an official to have behaved inappropriately. Record kept by race organizer.
- A competitor won a gold medal in an event. Brute fact: the person was in the group meeting at the particular time and place and behaved in a certain way. Context: person was a competitor, heat/final was a final, person was not declared by an official to have behaved inappropriately. Record kept by race organizer.

c) Describe two performative and two informative speech acts performed in this system.

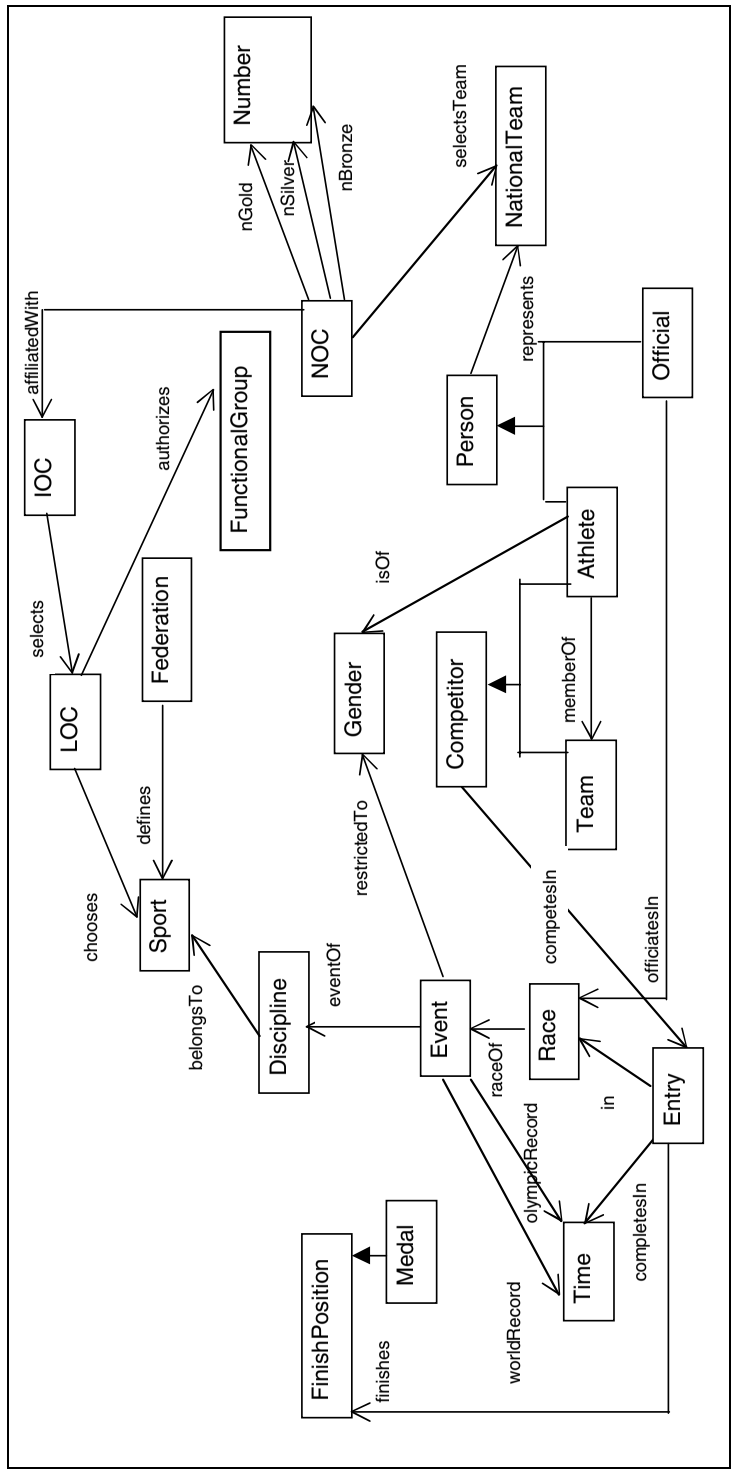
- Performative: a competitor winning a gold medal in an event. Informative: publication of the fact.
- Performative: assignment of a heat/lane to a competitor. Informative: telling the competitor to which heat/lane they have been assigned.

- Performative: IOC assigns Olympics to City, authorizing LOC to form.
- Performative: LOC organizes events for a particular sport.
- Performative: FINA devises rules for swimming events.

Chapter 4

1. Make a description of the ontology as a diagram in the representation language of Chapter 4. Include enough detail to understand the elements of the ontology involved in the interoperations. Include some instances of all classes, including all classes used in the actions of the exercise from Chapter 2.

Class	Instances	Class	Instances	Class	Instances
IOC	only one	Race	Final	Athlete	Thorpe
LOC	Beijing OC		Semifinal		Johnson
Federation	FINA		Heat3	Official	Xie
	IAAF	Medal	Gold		Lee
Sport	Aquatics		Silver	National Team	China
	Athletics		Bronze		Australia
Discipline	Swimming	Team	400mMedRelay MenAust	NOC	ChineseOC
	Track		4x400mWomen China		AustralianOC
Event	400m Free Men			Functional Group	SwimRaceOrg
	1500m Women				TrackRace Org


Number
nGold
nSilver
nBronze
selectsTeam
NationalTeam
affiliatedWith
NOC
represents
Official
IOC
authorizes
FunctionalGroup
Person
selects
Federation
isOf
Athlete
LOC
Gender
Competitor
memberOf
defines
chooses
Sport
restrictedTo
Team
competesIn
officiatesIn
belongsTo
Discipline
eventOf
Event
Race
raceOf
in
Entry
olympicRecord
Time
completesIn
worldRecord
FinishPosition
Medal
finishes

Chapter 5

1. Consider the Olympic fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise.

a. Which classes are independent and which are dependent? How are instances of the dependent classes identified?

The only truly independent classes are *IOC*, *Federation* and *Person*. All the other classes depend on them. For example, neither *LOC* nor *NOC* can exist without *IOC*.

Instances of many of the dependent classes are identified by sufficiently unique names given in the speech acts creating the instances. For example, the various *NOC* and *LOC* instances are created by the *IOC*, and the *IOC* has trademark protection worldwide on the name "Olympic Committee". The names of the sports, disciplines and events used in the Olympics are created jointly by the IOC and the sporting federations to be unique within the Olympics. For example, names of events could differ between the Olympics and purely sporting federation meets. FINA has both short course (25m pool) and long course (50m pool) events, while in the Olympics all swimming events are long course. It is also possible that both IAAF and FINA might have an event called Men's 100 metre sprint. In the Olympics, they would be differentiated. Typically the swimming event would be called "Men's 100 metre freestyle", leaving the original name for the track event.

Races within events are typically identified relative to the event. The final is the final of the Men's 100 metre sprint. An instance of *NationalTeam* is identified by the *NOC* that selects it. *Person* can have several identifiers. Within a national team, the person might be identified by name. With respect to a sporting federation, the person will have a sporting federation ID number. Within the Olympics, the person will be identified by an Olympics ID number. An instance of *Team* will be identified both by an Olympics ID number and by the combination of the name of the event and the name of the national team (Australian 4x100 metre medley relay).

The class *Entry* is an association class. Instances are identified by a combination of identifiers of *Competitor* and *Event*.

b. Describe the shared complex objects. What are their parts? What are their identifying and unifying relations? Are the unifying relations logical or lexical? Are there any objects identified indexically? Any identified metonymically? Are the independent classes used in unifying relations? (Make plausible additions if necessary.)

One complex object is an event. The event's parts include its races (*race raceOf event*). A race is also a complex object. Its parts include competitors (*competitor competesIn race*) and officials (*official officiatesIn race*). We saw identifying relations for these in the previous question.

There are different unifying relations depending on who is looking. The games organizer will have stored the races, events, competitors, officials and the properties in a database, so can identify all the parts of a race, or of an event, by

an SQL query. In this case, the unifying relation is logical. However, to a spectator the races belonging to an event are held on a scorecard, and the competitors and officials are written on pieces of paper or displayed on scoreboards. In this case, the unifying relation is lexical.

The athletes who are part of a team can similarly be identified either logically (if using a query on the database) or lexically (piece or paper or scoreboard, or by being physically present and performing appropriately - you can tell the members of a relay team simply by watching the race). The same applies to members of a national team.

IOC and *LOC* can both be identified indexically. There is only one IOC, so it can be identified simply by "the instance of the class *IOC*". There is only one *LOC* for a given Olympics, so it can be similarly identified.

It is possible to identify an instance of competitor by the instance of *Medal* associated with the instance of *Entry* (entry *finishes* gold) associated with the instance of competitor (Thorpe *competesIn* entry *in* 400mMenFreestyle). We can think of an instance of *Entry* as a complex object whose parts are instances of *Competitor*, *Race*, *Event*, *Time* and *FinishPosition*. Here we identify the entry by the parts *Medal* (gold) and *Event* (400mMenFreestyle).

In this example, the independent classes are too remote from most of the other classes to be used in unifying relations. An exception is *FunctionalGroups*. Every instance of *FunctionalGroups* is a part of the organization of the particular Olympics organized by the single instance of *LOC*. In the model as shown in our ontology, *LOC* is dependent on *IOC*. A plausible change would be to remove *IOC* from the ontology, as it plays no active part in the interactions of a particular Olympics after it selects a *LOC*. Instances of *NOC* are also affiliatedWith *IOC* typically long before the particular Olympics. So in our modified ontology, *LOC* (and *NOC*) is an independent class

c. Describe the bulk classes in the ontology, showing how they are bulk classes. What containers are they in? Do the containers give pseudo-identity?

Time and *Number* are both bulk classes because they lack unity. *FinishPosition* and its subclass *Medal* are countable classes, but their instances are bulk classes because they lack identity. (A gold medal is a gold medal, and is distinguished from a bronze medal, but the instance of a gold medal in trapshooting is different from the instance of gold medal in synchronized swimming. The two gold medal instances are distinguished only by their containers, the instance of *Entry* they are associated with via the property *finishes*.)

Number is in containers *nGold*, *nSilver*, *nBronze* identified by instances of *NOC*. *Time* is in containers *olympicRecord* and *worldRecord* identified by instances of *Event*, and *competesIn*, identified by instances of *Entry*. So the containers give pseudo-identity.

Chapter 6

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a. Describe the classes in the ontology. Give their rigid properties, indicating whether the rigid properties are lexical or logical, and their essential properties. Show a system of subclasses for each class, inventing a plausible system if necessary. Each subclass is either defined or declared. If it is defined, give the defining predicates. If it is declared, tell how objects are classified into the subclass and by which role. Show that the identifying and unifying relations are preserved in the subclass structure. Make plausible additions to the system if necessary.

Class	Rigid Properties	Log/ Lex	Essential(Not Rigid) Properties
IOC	Enumerated	Lexical	
LOC	Exists(inv)selects	Logical	
Federation	Enumerated	Lexical	
Sport	Exists(inv)defines	Logical	Exists(inv)chooses
Discipline	Exists-belongsTo	Logical	
Event	Exists-eventOf	Logical	Exists-restrictedTo
FinishPosition	Enumerated	Lexical	
Medal	Enumerated	Lexical	
Race	Exists-raceOf	Logical	
Time	Enumerated-bulk	Lexical	
Gender	Enumerated	Lexical	
Competitor	Enumerated	Lexical	
Team	Enumerated	Lexical	
Athlete	Enumerated	Lexical	Exists-represents
Person	Exists-represents	Logical	
Official	Enumerated	Lexical	Exists-represents
NationalTeam	Exists-selects	Logical	
NOC	Exists-affiliatedWith	Logical	Exists(inv)selects
Number	Enumerated-bulk	Lexical	
Functional Groups	Exists(inv)authorizes	Logical	

The properties indicated as Exists-p are mandatory (UML lower multiplicity 1). This means that every instance of the class must be linked to an instance of another class by the property p. The rigid or essential metaproperty is that such a link exists. It is true for every instance of the class. Exists(inv)p is Exists- applied to the inverse of p. The enumerated classes with lexical rigid properties simply say that an object is an instance of the class if it is declared so, perhaps by being stored in a table representing the class.

The subclass structure *FinishPosition/ Medal* is defined. *Medal* is FinishPosition < 4. *Competitor/ Team/ Athlete* and *Person/ Athlete/ Official* are all declared. Instances of *Athlete* are declared by their NOC without necessarily being registered as competing in any event. Instances of *Team* are declared by their NOC without their members being necessarily determined. Similarly, instances of

Official are declared by their NOC without their necessarily being assigned to officiate in any event.

Every instance of *Team* or *Athlete* will have a value for the identifying property of *Competitor*. Every instance of *Athlete* or *Official* will have a value for the identifying property of *Person*. Instances of *Medal* are distinguished instances of *FinishPosition* (1, 2 and 3), with an additional property *medalColor* with values respectively *gold*, *silver*, *bronze*.

The classes shown as having subclasses are all classes of simple objects. A class of complex objects is *Event*. We could define a system of subclasses using for example the property *restrictedTo* (*MensEvent*, *WomensEvent*, *MixedEvent*). The unifying relation would be unchanged. We could define another system of subclasses based on whether competitors were teams or not by a query on the composition of *raceOf* and *competesIn*, which also preserves the unifying relation.

b. Describe a property (relationship, association) involving at least one of the classes from a. This property should have a subproperty structure. Invent a plausible structure if necessary. Show a population of property instances, including at least one instance of each subproperty.

A well-understood property is the derived property *medalWon* whose domain is *Athlete* and whose range is *Medal*. The property is derived as the union of two compositions:

I: *competesIn* composed with *finishes*

T: *memberOf* composed with *competesIn* composed with *finishes*

Each composition is a subproperty

I is *individualMedalWon*

T is *medalWonAsTeamMember*

So *medalWon* is the union of two subproperties.

Chapter 7

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a. Describe examples of systems, subsystems, and environments of systems in the ontology. Find some coupled objects in the ontology. Identify hereditary and emergent properties of the systems. Make plausible additions to the system if necessary.

It is common in the Olympics to think of instances of *Discipline* as systems, composed of (subclasses of) *Event*, *Race*, *Competitor* and *Official*, linked by the properties *eventOf*, *raceOf*, *officiatesIn*, *competesIn* and *memberOf*. It is common to think of instances of *Event* as subsystems of *Disciplines*, composed of subclasses and subproperties of the events and properties comprising the *Disciplines* systems.

To see that an event E is a system in terms of the BWW formal ontology, consider the instances of *Race* linked to E by the property *raceOf*. An instance of *raceOf* is a Thing. It is created by a speech act by the race organizer. The same speech act creates an instance of *Race*. The speech act is an event in the BWW

formal ontology. So the history of E is coupled with the history of *raceOf* and therefore with the history of the associated instances of *Race*.The race organizer also creates instances of *competesIn* and *officiatesIn*, thereby coupling instances of *Competitor* and *Official* with E. Every object in the system is coupled with E, so there are no independent parts.

A similar argument shows that an instance R of *Race* is a system. The instance R is a part of E, so is a subsystem of E. Another similar argument shows that E is a subsystem of an instance D of *Discipline*.

The environment of E includes instances of *Gender*, *FinishPosition*, *Time* and *NationalTeam*, coupled by instances of *finishes*, *completesIn*, *restrictedTo* and *represents* created in speech acts by various of the agencies involved. E is also coupled to instances of *Discipline*, *Sport*, *Federation*, etc.

An individual event E has a property *eventOf*, so is linked to an instance D of *Discipline*. This becomes a hereditary property of the system E. Emergent properties of E include derived properties like *numberOfRaces*, *numberOfCompetitors*, the subset of *Athlete* competing as either individual or as member of a team, the subset of *NationalTeam* linked to the subset of *Athlete* by *represents*, and so on.

b. Describe endurants and perdurants in the ontology. What endurants participate in the perdurants? How are the histories of the endurants represented in the ontology?

All instances of classes are endurants, as are all instances of properties. The perdurants are the speech acts that create the instances of classes and properties. The speech act that creates the gold medal winner in the men's 100 metre sprint involves the endurants competitor, race, event and medal as participating. In the Olympics, the endurants are never deleted. The finishing time in the 15th position of the men's 100 metre sprint in the 1928 Olympics will remain in the records until the Olympics are forgotten.

The histories of the endurants are represented by the populations of the properties.

c. Think of an improvement in the ontology suggested by some aspect of the BWW/Dolce formal upper ontologies. Show how the improvement follows from the upper ontologies.

The ontology so far does not include the rules for the various events in the various disciplines. These rules define the contexts for the speech acts (perdurants, or BWW events) that create the endurants. Inclusion of the rules follows naturally from thinking of how the instances of the classes and properties are created, using the concepts of the formal ontologies.

Chapter 8

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a. Criticise the ontology in terms of the five principles of Gruber.

- Clarity: suggest a plausible unintended interpretation of one of the concepts. How does (or could) the ontology prevent that unintended interpretation?

We expect that an athlete competes in a race either individually or as a member of a team, but not both at the same time. Of course, an athlete may compete in one race as an individual and in another as a member of a team (think of relay teams). The ontology does not prevent the unintended interpretation of an athlete participating in a race as both an individual and as a member of a team. One way the ontology could prevent that is by declaring two subclasses of *Event*: *TeamEvent* and *IndividualEvent*. These subclasses would induce corresponding subclasses of *Race*: *TeamRace* and *IndividualRace*. Then we could introduce a constraint on *TeamRace* restricting the domain of *competesIn* to instances of *Team*, and a constraint on *IndividualRace* restricting the domain of *competesIn* to instances of *Athlete*.

The clarity of the Olympics ontology is therefore fairly low.

- Coherence: suggest a plausible inference that an agent could be expected to draw from the ontology. How does (or could) the ontology support the reasoning necessary to make the inference?

If an athlete wins a race, the athlete is awarded a gold medal. This is an inference drawn from the composition of the properties *competesIn* and *finishes*, where the instance of *Entry* linking the instance of *Athlete* to the instance of *Medal* is also linked to an instance of *Race* of type *Final*.

If a team wins a race, the medal is actually awarded to the team members, not the team. So in case the instance of *Competitor* linked to gold medal is an instance of *Team*, we need a further link to an instance of *Athlete* via the inverse of *memberOf*.

The ontology provides all the information needed to make the inferences, but not the formulas to express them. A possible language to do this is the UML language OCL (Object Constraint Language). Another is Common Logic. So to make the inferences the representation language needs to be supplemented by a method of expressing formulas and of computing with them.

Coherence is medium.

- Extendibility: suggest a plausible extension to the ontology. Show what changes would need to be made. Are any of the changes redundant? If so, show how. If not, show how the ontology design anticipated the extension.

A plausible extension is to introduce a new discipline, with its associated events and so on. Let us say the discipline is Iron Man/Woman, a type of race consisting of a cycling leg followed by an ocean swimming leg, finishing with a running leg. This discipline is structurally very similar to the *Event*/*Race* structure in the existing ontology, and could be implemented simply with new instances of *Sport*, *Federation*, *Discipline*, *Event*, *Race* and so on. None of the changes are redundant. This is because many disciplines have this structure, so the necessary classes and properties are already present.

There are of course many different kinds of discipline. Most new disciplines are structurally similar to existing disciplines. For example, Synchronized Swimming was introduced into the Olympics fairly recently, and Beach Volleyball only in 2004. The former is a judged sport, similar to gymnastics, figure skating and diving, while the latter is similar to Volleyball, Tennis, Badminton and Table

Tennis, with a tournament structure. Both would have been fairly straightforward extensions.

An ontology for the Olympics would generally be high in extendibility, if for no other reason than that there are so many disciplines already represented that a new one is likely to be similar to an existing one.

- Encoding bias: Show how one of the actions of the exercise of Chapter 2 might be implemented. Does it make sense for it to be implemented in a different way? If not, why not? If so, does the ontology make the different implementation difficult? Consider each element of the implementation of the action.

Consider the action "race organizer assigns heat/final lane to competitor".

This is actually a fairly complex process, since competitors are assigned to their first heat based partly on their world ranking (to avoid forcing strong competitors to be eliminated before the final). Assignment of lanes in semifinals and finals is determined mostly be algorithm from finishing times in earlier races. However, the ontology does not provide structural support for a complex object whose parts are subject to strong constraints. The implementation is therefore underconstrained rather than overconstrained. Essential aspects of the structure are not represented, so are left to be encoded in the particular set of programs used in a particular instance of a system using the ontology.

So there are many possible implementations, and the ontology makes implementations more difficult because they all must encode the constraints in their own way. These encodings are in effect part of the ontology.

The encoding bias can therefore be considered to be fairly high.

- Ontological commitment: There are many different systems of interoperations in which the ontology could be reused. Describe one such system and how the ontology could be adapted to the re-use.

There are many Games more or less similar to the Olympics. One such is the Commonwealth Games, held in 2006 in Melbourne, Australia. This event is smaller than the Olympics, with about 1/3 the number of sports and disciplines. It includes disciplines also included in the Olympics (like Swimming) and some not included in the Olympics (like Lawn Bowls, Netball and Rugby 7s).

Because the sports are organized by the sporting federations in both cases, the structure of disciplines in common can be pretty well carried straight over from the Olympics. Sports not represented in the Olympics can be incorporated in the way discussed in *Extendibility* above.

The high-level structures (IOC, LOC, NOC etc) of the Olympics all have equivalents in the Commonwealth Games.

The main problem comes from the fact that the Olympics is bigger than the Commonwealth Games. Within a discipline there are events in the Olympics not in the Commonwealth Games. These can be easily removed since they are represented as instances of the structural classes. However, there are also types of disciplines not represented.

Our fragment of the Olympics Ontology does not show this explicitly, but for each type of discipline, there will be a different system of structural classes

(Event, Race etc) and properties that are built into the Ontology. For the Commonwealth Games to avoid making a commitment to these unnecessary discipline types, the discipline-type-specific structural classes would have to be removed by ontology maintenance procedures. Alternatively, they could be left empty, which would lead to implementation untidiness in the form of blank fields in reports or blank reports, or unnecessary screens on menus.

The ontological commitment of the Olympics Ontology is therefore medium.

- In each quality dimension, indicate whether in your judgment the quality is high or low.

b. Propose an improvement to the ontology. Argue why this improvement is a good idea in terms of at least one of Gruber's principles.

One of the aspects of the ontology keeping encoding bias higher than it can be is the lack of a way of describing a whole/part structure whose parts are related by strong constraints. Introducing such a facility would enable all the aspects of the ontology of these situations to be explicitly represented, and therefore eliminate the redundant and possibly incompatible multiple implementations of the constraint structures. So the improvement is a good idea in that it leads to reduced encoding bias.

c. Examine the cost and benefits of the improvement, taking into account the generality of the ontology and the number of implementations one might expect it to have. On balance, is the improvement a good idea? Take a position and justify it.

The principal benefit of this improvement is that the cost of implementing an event structure is reduced. So the benefit depends on the number of implementations there are.

It turns out that the event structure is an aspect of the ontology which is the responsibility of the sporting federations. Any event organized under the auspices of the federation will re-use the event structure implementation. We can therefore expect a large number of implementations, so the benefit is fairly high.

The cost of the improvement is mainly in the design of the class libraries and constraint languages and implementations needed to allow the event organizers to use the abstract data type. This is a moderately high cost, but contemporary ontology representation languages like OWL and Common Logic have facilities to make the development easier.

On balance, the improvement is probably a good idea. The large number of expected re-uses of the structure representation should repay the development cost.

Chapter 10

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a. Represent a significant portion of the ontology in RDFS along the lines of Appendix B, using all the key concepts from the Chapter.

Remember that RDFS servers will infer some declarations from others. Below, the minimal declarations will be shown in **bold**, with the inferred declarations in normal. Includes all classes, properties and instances in the Chapter 4 solution.

selects rdfs:domain IOC
selects rdfs:range LOC
chooses rdfs:domain LOC
chooses rdfs:range Sport
defines rdfs:domain Federation
defines rdfs:range Sport
belongsTo rdfs:domain Discipline
belongsTo rdfs:range Sport
authorizes rdfs:domain LOC
authorizes rdfs:range FunctionalGroup
IOC rdf:type rdfs:Class
LOC rdf:type rdfs:Class
Sport rdf:type rdfs:Class
Federation rdf:type rdfs:Class
Discipline rdf:type rdfs:Class
FunctionalGroup rdf:type rdfs:Class
selects rdf:type rdf:Property
chooses rdf:type rdf:Property
defines rdf:type rdf:Property
belongsTo rdf:type rdf:Property
authorizes rdf:type rdf:Property
eventOf rdfs:domain Event
eventOf rdfs:range Discipline
raceOf rdfs:domain Race
raceOf rdfs:range Event
typeOfRace rdfs:domain Race
typeOfRace rdfs:range RaceType
in rdfs:domain Entry
in rdfs:range Race
completesIn rdfs:domain Entry
completesIn rdfs:range Time
worldRecord rdfs:domain Event
worldRecord rdfs:range Time
olympicRecord rdfs:domain Event
olympicRecord rdfs:range Time
finishes rdfs:domain Entry
finishes rdfs:range FinishPosition
Medal rdfs:subClassOf FinishPosition
Event rdf:type rdfs:Class
Race rdf:type rdfs:Class
RaceType rdf:type rdfs:Class
Entry rdf:type rdfs:Class
Time rdf:type rdfs:Class
FinishPosition rdf:type rdfs:Class
Medal rdf:type rdfs:Class
eventOf rdf:type rdf:Property
raceOf rdf:type rdf:Property
typeOfRace rdf:type rdf:Property
inrdf:type rdf:Property
completesIn rdf:type rdf:Property
worldRecord rdf:type rdf:Property
olympicRecord rdf:type rdf:Property
finishes rdf:type rdf:Property
restrictedTo rdfs:domain Event
restrictedTo rdfs:range Gender
isOf rdfs:domain Athlete
isOf rdfs:range Gender
competesIn rdfs:domain Competitor
competesIn rdfs:range Entry
memberOf rdfs:domain Athlete
memberOf rdfs:range Team
officiatesIn rdfs:domain Official
officiatesIn rdfs:range Race
Team rdfs:subClassOf Competitor
Athlete rdfs:subClassOf Competitor
Athlete rdfs:subClassOf Person
Official rdfs:subClassOf Person
Gender rdf:type rdfs:Class
Competitor rdf:type rdfs:Class
Team rdf:type rdfs:Class
Athlete rdf:type rdfs:Class
Person rdf:type rdfs:Class
Official rdf:type rdfs:Class
restrictedTo rdf:type rdf:Property
isOf rdf:type rdf:Property
competesIn rdf:type rdf:Property
memberOf rdf:type rdf:Property
officiatesIn rdf:type rdf:Property
represents rdfs:domain Person
represents rdfs:range NationalTeam
selectsTeam rdfs:domain NOC
selectsTeam rdfs:range NationalTeam
nGold rdfs:domain NOC
nGold rdfs:range Number
nSilver rdfs:domain NOC
nSilver rdfs:range Number
nBronze rdfs:domain NOC
nBronze rdfs:range Number
affiliatedWith rdfs:domain NOC

affiliatedWith	**rdfs:range**	**IOC**
NationalTeam	rdf:type	rdfs:Class
NOC	rdf:type	rdfs:Class
Number	rdf:type	rdfs:Class
represents	rdf:type	rdf:Property
selectsTeam	rdf:type	rdf:Property
nGold	rdf:type	rdf:Property
nSilver	rdf:type	rdf:Property
nBronze	rdf:type	rdf:Property
affiliatedWith	rdf:type	rdf:Property
BeijingOC	**rdf:type**	**LOC**
FINA	**rdf:type**	**Federation**
IAAF	**rdf:type**	**Federation**
Aquatics	**rdf:type**	**Sport**
Athletics	**rdf:type**	**Sport**
Swimming	**rdf:type**	**Discipline**
Track	**rdf:type**	**Discipline**
400mFreeMen	**rdf:type**	**Event**
1500mWomen	**rdf:type**	**Event**
Final	**rdf:type**	**RaceType**
SemiFinal	**rdf:type**	**RaceType**
Heat	**rdf:type**	**RaceType**

AQ001	**rdf:type**	**Race**
AQ034	**rdf:type**	**Race**
TR013	**rdf:type**	**Race**
Gold	**rdf:type**	**Medal**
Silver	**rdf:type**	**Medal**
Bronze	**rdf:type**	**Medal**
400mMedRelayMenAust Team		**rdf:type**
4x400mWomenChina Team		**rdf:type**
Thorpe	**rdf:type**	**Athlete**
Johnson	**rdf:type**	**Athlete**
Xie	**rdf:type**	**Official**
Lee	**rdf:type**	**Official**
China	**rdf:type**	**NationalTeam**
Australia	**rdf:type**	**NationalTeam**
ChineseOC	**rdf:type**	**NOC**
AustralianOC	**rdf:type**	**NOC**
SwimRaceOrg	**rdf:type**	**FunctionalGroup**
TrackRaceOrg	**rdf:type**	**FunctionalGroup**

Note that *IOC* is a class with one instance only. In RDFS it can appear as an individual as well as a class.

Instances of *Entry* would be blank nodes, each of which would be the subject of triples whose predicates are *in*, *completesIn* and *finishes*, and the object of a triple whose predicate is *competesIn*.

b. What concepts are you using that might be in more general namespaces? Suggest who might maintain these namespaces, if they do not already exist.

An IOC namespace would contain all the class, property and individual names, except for the instances of *Federation*. Those individuals would be in their own namespaces.

A FINA namespace would contain the individual FINA and the relevant *Sport*, *Event*, *Race* etc class and individual names, as well as the relevant property names.

An IAAF namespace would contain the individual IAAF and the relevant *Sport*, *Event*, *Race* etc class and individual names, as well as the relevant property names.

Each NOC would own a namespace containing the relevant *NationalTeam* instance and the relevant instances of *Person*.

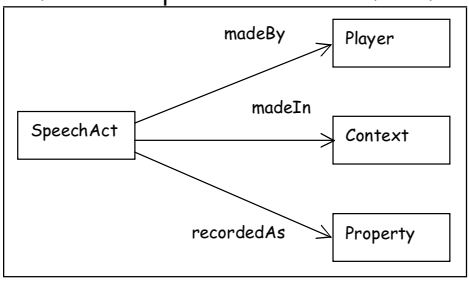
The BeijingOC would own a namespace containing the instances of *FunctionalGroup*.

Note that many of the resources are defined two or more times, sometimes with different names (see the discussion of identification in part a of Chapter 5 solution).

c. How would you represent the players and speech acts in RDF?

The players are represented as instances of classes, in particular *IOC*, *LOC*, *NOC*, *Official*, *Competitor*.

The speech acts are not directly represented in the ontology of the Chapter 4 solution. Speech acts are represented in the operation of the ontology by the insertion of instances of properties. Speech acts are also associated with players and contexts. One way to model the speech acts is as a metamodel



Instances of *Player* are the various individuals who act as players. Instances of *Property* are the various properties recording the speech acts. Instances of *Context* would be documents referred to by URIs. Instances of *SpeechAct* could be blank nodes.

For example, a speech act "selecting an LOC" would be a blank node acting as the subject of triples whose predicates are *madeBy*, *madeIn* and *recordedAs*, with objects respectively *IOC*, an appropriate context document and the property *selects*. The speech act does not need a separate URI, as it is uniquely identified by the property instance.

Chapter 11

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following, as represented in RDFS in the last Exercise

a. Elaborate a significant portion of the ontology in OWL using the OWL facilities from the Chapter.

Replace the rdfs:Class declarations of *FinishPosition* and *Number* by instances of owl:DataRange. Replace the declaration of *Medal* as rdfs:Class by an instance of owl:DataRange defined as owl:oneOf {"gold", "silver", "bronze"}.

Replace remaining occurrences of rdfs:Class by owl:Class. Note that if only the minimum declarations are made (bold in the RDFS solution), importing them into an OWL server rather than an RDFS server will automatically make the owl:Class inferences. This weak typing can be thought of as reducing encoding bias. On the other hand, weak typing can be thought of as decreasing clarity, since a weakly typed system allows misspellings to be interpreted as in this case classes, which is surely unintended.

Replace all occurrences of rdf:Property by owl:ObjectProperty, except for *finishes*, *completesIn*, *worldRecord*, *olympicRecord*, *nGold*, *nSilver*, *nBronze*, which become owl:DatatypeProperty. Note that the minimum declarations with the addition of the owl:DataRange declarations permit these declarations to be inferred by the OWL server.

Declare as owl:equivalentClass the classes shared by the IOC and the various *Federation* instances, for example.

ioc:Sport owl:equivalentClass fina:Sport
ioc:Sport owl:equivalentClass iaaf:Sport

Declare as owl:equivalentProperty the properties shared by the IOC and the various *Federation* instances, for example

ioc:belongsTo owl:equivalentProperty fina:belongsTo
ioc:belongsTo owl:equivalentProperty iaaf:belongsTo

Declare as owl:sameAs the individuals shared by the IOC and the various *Federation* instances, for example (see part a of chapter 5 solution)

ioc:100mFreestyleMen owl:sameAs fina:100mSprintMenLongCourse
ioc:100mSprintMen owl:sameAs iaaf:100mSprintMen

Similarly, declare as owl:sameAs the individuals which are shared by the IOC and various *NOC* instances as instances of *Person*.

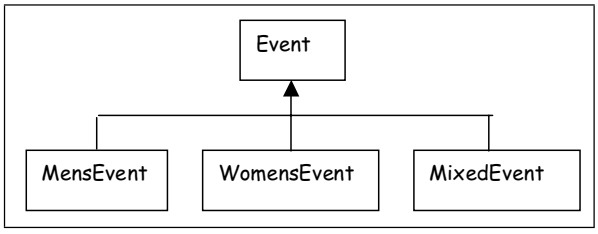
Declare as owl:FunctionalProperty the properties *belongsTo*, *eventOf*, *raceOf*, *worldRecord*, *olympicRecord*, *in*, *finishes*, *isOf*, *represents*, *nGold*, *nSilver*, *nBronze*, *affiliatedWith*.

Declare as owl:InverseFunctional the properties *selects*, *chooses*, *authorizes*, *defines*, *competesIn* and *selectsTeam*.

Define the domains of mandatory properties as subclasses of the restriction classes MinCardinality = 1 on the properties. This applies to

LOC on inverseOf selects
Sport on inverseOf chooses
Sport on inverseOf defines
FunctionalGroup on inverseOf authorizes
Discipline on belongsTo
Event on eventOf
Race on raceOf
Entry on in
Entry on inverseOf competesIn
Person on represents
NationalTeam on inverseOf selectsTeam
NOC on affiliatedWith

Define subclasses of *Event*



MensEvent owl:equivalentClass the restriction class on restrictedTo hasValue male
WomensEvent owl:equivalentClass the restriction class on restrictedTo hasValue female
MixedEvent owl:equivalentClass
owl:intersectionOf Event
owl:complementOf
owl:unionOf MensEvent WomensEvent

It would be possible to define subclasses of *Athlete* by *NationalTeam*, for example

AustralianAthlete owl:equivalentClass
owl:intersectionOf Athlete
Restriction on represents owl:hasValue "Australia"

then define subclasses of Team by NationalTeam, for example

AustralianTeam owl:equivalentClass
Restriction on inverseOf memberOf
owl:allValuesFrom AustralianAthlete

then define a subclass

AustralianCompetitor owl:equivalentClass
owl:unionOf AustralianAthlete AustralianTeam

and finally a subclass of *Event*: *EventInterestingToAustralia*, that is any event where an Australian competes, by a chain of restrictions

EventInterestingToAustralia owl:equivalentClass
Restriction on inverseOf raceOf owl:someValuesFrom
Restriction on inverseOf in owl:someValuesFrom
Restriction on inverseOf competesIn
owl:allValuesFrom AustralianCompetitor

However, this is a bit cumbersome, as there are nearly 200 national teams.

b. Describe a concept you can't represent in OWL (make a plausible extension if necessary).

There are many concepts that would be useful in the Olympics ontology that can't be represented in OWL. For example:

OWL doesn't support composition of properties. It would be useful to have a property *competesInEvent* whose domain is *Competitor* and whose range is *Event* formed by the composition of *competesIn*, *in* and *raceOf*. Also a property *medalWon* whose domain is *Athlete* and whose range is *Medal*, described in part b of the solution to the Chapter 6 exercise.

OWL doesn't support comparison of property values, so can't tell whether the value of *completesIn* for an instance of Entry is less than the Olympic record for the event, even if the property composition of *in*, *raceOf* and *olympicRecord* were possible.

c. How would you use the facilities of OWL Full in this ontology (make a plausible extension if necessary)?

The most significant capability of OWL Full not available in OWL DL is that Thing, OWLClass and Property are not disjoint.

A combination of Class and Property is what is known in UML as an association class. It would be an advantage if we could replace *Entry* by a property whose domain is *Competitor* and whose range is *Race*, which itself could be the domain of *finishes* and *completesIn*.

A combination of Thing and Class enables a class whose instances are themselves classes, like the classes rdfs:Class and owl:Class. The classes *AustralianAthlete*, *AustralianTeam*, *AustralianCompetitor* and *EventsInterestingTo Australia* defined in part a would have one counterpart for each of the nearly 200 *NationalTeam* instances. It could be an advantage to define a class *AthleteByCountry* whose instances were the classes *AustralianAthlete*, *ChineseAthlete*, *AmericanAthlete*, etc. Similarly for the other three cases.

A combination of Thing and Property enables a class whose instances are properties. The discussion of representation of speech acts in part c of the solution to chapter 10 could be implemented, instead of by blank nodes, by a class whose instances were the properties whose instances are the records of the speech acts. These instances could be linked to their respective *Player* and *Context* by properties whose domain is the resulting class of properties. The instances of *SpeechAct* would then be *affiliatedWith*, *selects*, *authorizes*, *chooses*, *defines*, *belongsTo*, *eventOf*, *raceOf*, *selectsTeam* and so on. The class *SpeechAct* would be the domain of the properties *madeBy* and *madeIn* as in the chapter 10 solution, but the property *recordedAs* would no longer be needed.

Chapter 12

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following.

a) Describe examples of bulk classes in the ontology.

The dataranges *Number* and *Time* are both bulk classes, since they lack unity. The number of gold medals is an aggregation measured in a dimension of units. The finish times are aggregrations, measured in a dimension of hundredths of a second. Both are represented as integers, as the dataranges have type *nonNegativeInteger* (Number) and *positiveInteger* (Time).

The datarange *FinishPosition* has also type *positiveInteger*, but this class is countable. *FinishPosition* 1 is distinguished from *FinishPosition* 29, for example. Similarly, the datarange *Medal* is countable. However, the instances of *FinishPosition* and *Medal* are like the product codes discussed in Chapter 5. They can be interpreted as bulk classes lacking identity (anonymous). In particular, the instance "gold" of *Medal* represents a class of speech act, the award of a gold medal. But there are many gold medals awarded in the various events of the various sports, and the medals are not the same. The different instances of "gold" are identified by their containers, in this case the *Event* for which they are awarded (composition of *in* and *raceOf* for a *Race* of type "final" and *finishes* = 1).

b) Find situations where the concept/ representation class distinction is useful. (Make plausible extensions if necessary).

The class *Person*, especially the subclass *Athlete*, is of interest to people whose different languages number in the hundreds. While many languages share the latin alphabet of the official languages of the IOC, very many do not. Names therefore have many representations. It makes sense to think of the class *Person* as a concept class with many representation classes, one at least for each writing system used by one of the languages of people interested in the Olympics.

In fact, all the classes used in recording the results of the Olympics as would appear on web pages can be considered to be concept classes, with many representation classes used in rendering the Olympics results pages in different languages and writing systems.

In the Chapter, dimension systems are discussed as good candidates for representation classes. In the specification and design aspects of the Olympics (not represented in the fragment of Chapter 4) there are many instances of dimensioned quantities. However, the Olympics ontology would include only specific dimensioned objects (the length of the swimming pool, for example), not a general dimension processing system. So it would probably not be useful to use representation classes for the dimensioned objects in this case.

c) Find deep part/whole structures (with more than one level of part), and describe them using the formal taxonomy of the Chapter. (The races, etc of an event might be a good candidate.)

An event is a complex object, with parts instances of *Race* associated by the property *raceOf*. A race is a complex object, with parts instances of *Entry* by the property *in*. An instance of *Entry* is associated with an instance of *Competitor* via *competesIn*, but it is probably not useful to think of the competitor as a part of the event, since a competitor may compete in many events. We could declare both *raceOf* and *in* as subproperties of *partOf*.

It makes sense to think of an instance of *Entry* as a part of the event, so *partOf* in this case is transitive. Both *Race* and *Entry* are related to *Event* functionally. The whole does not depend on any specific part, except that there must be at least one instance of *Race* and at least two instances of *Entry*. On the other hand, an instance of *Race* or *Entry* does not make any sense without an instance of *Event*, so the parts depend on the whole. Finally, the instance of *Event* is built up over time in a complex process. So it probably does not make sense to think of the event as an essential whole, even though when the event is completed its collection of parts is fixed.

d) Find examples of the metaproperties +unity, +identity +rigid, +essential. Model them using the n-ary association method from the Chapter.

We show the n-ary association in a repository (relational scheme) view.

Metaproperty			
withRespectTo	**on**	**takesValue**	**withStatus**
Event	(inv)raceOf	unity	necessarily
Race	(inv)in	unity	necessarily
Event	eventOf x name	identity	necessarily
Event	Exists-eventOf	rigid	necessarily
Event	Exists-belongsTo	essential	necessarily
Event	Exists(inv)defines	essential	necessarily
Event	Exists(inv)chooses	essential	necessarily
Event	Exists(inv)selects	essential	necessarily

We assume that *Event*, *Race*, etc are subclasses of the domain of a datatype property *name*, whose range is of type *string*.

e) Find situations in this ontology where various kinds of extent-descriptive metaclasses might be useful, and describe how in each case. Model them.

For the Olympics ontology a useful extent descriptor is the agency responsible for the speech acts creating instances of a class and therefore of its mandatory

properties. This was discussed in the solution for Chapter 8, part c, and in the solution for Chapter 11, part c.

Implementations of systems committed to the Olympics ontology are of two general kinds: those not accessible to the general public or those operating outside the Olympics competition period, and those operating during the competition period supporting the Olympics web site. In particular, it would be useful to the web site implementation to know which classes would have their extent fixed at the beginning of the competition period. The extents of these classes could then be freely copied to the many servers needed without concern for propagating updates. This predicate could be a simple Boolean, *true* if the population is frozen at the start of the competition.

Chapter 13

1. Consider the Olympics fragment from the Chapter 2 exercise and the representation in the solution to the Chapter 4 exercise and following. (Make plausible extensions if necessary).

a. Define subclasses using CL.

Subclasses *MensEvent*, *WomensEvent*, *MixedEvent* discussed in solution Chapter 11 part a.

```
(forall e (iff (MensEvent e) (and (Event e) (restrictedTo e, "men"))))
(forall e (iff (WomensEvent e) (and (Event e) (restrictedTo e, "women"))))
(forall e (iff (MixedEvent e) (and (Event e)
        (not (or (MensEvent e) (WomensEvent e))))))
```

The national teams discussed in the Chapter 11 solution following the Mens, Womens and Mixed events could be defined with a parameterised collection of subclasses of *Competitor*, *Athlete* and *Team*, where for example

```
(Competitor country competitor)
```

represents the subclass of *Competitor* who represent the particular country. The definitions are top-down:

```
(forall (ctr, comp) (iff (Competitor ctr comp)
(or (Team ctr comp) (Athlete ctr comp))))
(forall (ctr, comp) (iff (Athlete ctr comp)
      (and (Athlete comp) (represents comp ctr))))
(forall (ctr, comp) (iff (Team ctr comp)
      (and (Team comp) (forall c (implies
(memberOf c, comp) (Athlete, c, ctr))))))
```

The last definition requires an integrity constraint that all members of a team represent the same country.

Then we can define a parameterised series of subclasses

```
(EventInterestingTo country event)
```

which are the events interesting to people of a particular country, as

```
(forall (ctr, ev) (iff (EventInterestingTo ctr ev)
        (exists rc (and
                (raceOf rc ev) (exists ent (and (in ent rc)
                        (exists comp (and (competesIn comp ent)
                                (Competitor ctr comp)
            ))
        ))
    ))
))
```

assuming the domains and ranges of all properties are defined as integrity constraints.

These definitions are all much simpler than the OWL definitions described in the Chapter 11 solution. In particular, it would not be easy in OWL to define parameterised subclass structures.

From the solution to Chapter 11, part b, we have some definitions not possible at all in OWL. In particular a composite property *competesInEvent*, which we can define

```
(forall (ev, comp) (iff (competesInEvent comp ev) (exists (ent rc) (and
      (competesIn comp ent) (in ent rc) (raceOf rc ev)))))
```

and the property medalWon described in the Chapter 6 solution, as

```
(forall (ath med) (iff (medalWon ath med) (or
        (medalWonInd ath med) (medalWonTeam ath med))))
(forall (ath med) (iff (medalWonInd ath med) (and (Medal med)
        (exists ent (and (competesIn ath ent) (finishes ent med))))))
(forall (ath med) (iff (medalWonTeam ath med) (and (Medal med)
        (exists (ent tm) (and
                (memberOf ath tm)
                (competesIn tm ent)
                (finishes ent med)
        ))
)))
```

assuming that the instances of finishes whose object is a medal are constructed appropriately.

Both of these derived property definitions permit the definition of subclasses:

competesInEvent allows the definition of a parameterised collection of subclasses of *Competitor*: those competitors competing in an event.

```
(forall (comp, ev) (iff (CompetitorInEvent ev comp)
        (competesInEvent comp ev)))
```

medalWon allows the definition of a subclass of *Athlete*: *MedalWinner*

```
(forall ath (iff (MedalWinner ath) (exists med (medalWon ath med))))
```

b. Define some integrity constraints using CL.

Domain and range constraints on a property can be represented in CL. For example, the property *restrictedTo*

```
(forall (ev gen) (implies (restrictedTo ev gen) (and (Event ev) (Gender gen))))
```

Mandatory properties, eg *eventOf* on *Event*

```
(forall ev (implies (Event ev) (exists rc (and (Race rc) (eventOf ev rc)))))
```

That all members of a team must represent the same country

```
(forall tm (implies (and (Team tm) (exists (a1 a2 c1 c2)
          (and (memberOf tm a1) (represents a1 c1)
               (memberOf tm a2) (represents a2 c2)))
        (= c1 c2)
))
```

If an event is restricted to a gender, all athletes competing must be of that gender.

```
(forall (ev gen) (implies (restrictedTo ev gen)
          (forall comp
              (and
                  (implies
                      (and (competesInEvent ev comp)
                          (Athlete comp))
                      (isOf comp gen))
                  (implies
                      (and (competesInEvent ev comp)
                          (Team comp))
                      (forall ath
                          (and (memberOf ath comp)
                              (isOf ath gen)))
                  )
              )
))
```

c. Formulate some queries as class definitions in CL

All the class definitions in part a can be used as queries.

These and other definitions can be combined. For example, Australian medal winners

```
(forall ath (iff (AustralianMedalWinner ath)
              (and (MedalWinner ath)
                  (represents ath "Australia"))
))
```

Chapter 14

1. Construct a topic map from the wikipedia entry for the Athens Olympics, as cited in the exercise for Chapter 2. Use instances of all topic map constructs.

2004 Summer Olympics

From Wikipedia, the free encyclopedia

Games of the XXVIII Olympiad

The 2004 Summer Olympics, officially known as the Games of the XXVIII Olympiad, were held in Athens, Greece, over a period of 17 days from August 13 to August 29, 2004. Planners expected 10,500 athletes (in fact 11,099 competed) and 5,500 team officials from 202 countries. Athens 2004 marked the first time since the 1996 Summer Olympics that all countries with a National Olympic Committee were in attendance. There were a total of 301 medal events from 28 different sports.

Contents
1 Medal count
2 Bid and preparations

```
/* Topics */
/* Using the IOC web page as subject identifier */
[2004SummerOlympics = "Athens Olympics" ("Games of the XXVIII Olympiad"/IOC)
        @http://www.olympic.org/uk/games/past/index_uk.asp?OLGT=1&OLGY=2004]
[1996SummerOlympics = "Atlanta Olympics" ("Games of the XXVI Olympiad"/IOC)
        @http://www.olympic.org/uk/games/past/index_uk.asp?OLGT=1&OLGY=1996]
/* Using a Wikipedia section heading as a subject identifier */
[Bid_and_preparations
@http://en.wikipedia.org/wiki/2004_Summer_Olympics#Bid_and_preparations]
[Medal_count @http://en.wikipedia.org/wiki/2004_Summer_Olympics#Medal_count]
/* Using the Wikipedia URI as a subject identifier */
[SummerOlympics @http://simple.wikipedia.org/wiki/Summer_Olympic_Games]
[WinterOlympics @http://simple.wikipedia.org/wiki/Winter_Olympic_Games]
[OlympicGames @http://www.olympic.org/uk/games/index_uk.asp]
[AthensGreece = "Athens" ("Αθήνα"/el)  @http://en.wikipedia.org/wiki/Athens]
[CompleteOlympics = "All NOCs participated"]
[MostRecent]
[TopicMap = "Topic Map" @http://psi.topicmaps.org/iso13250/glossary/topic-map]
/* Occurrences */
{2004SummerOlympics, StartedOn, [[2004/08/13]]}
{2004SummerOlympics, EndedOn, [[2004/08/29]]}
{2004SummerOlympics, DaysLasted, [[17]]}
{2004SummerOlympics, AthletesExpected, [[10500]]}
{2004SummerOlympics, AthletesCompeted, [[11099]]}
{2004SummerOlympics, MedalEvents, [[301]]}
{2004SummerOlympics, Sports, [[28]]}
{2004SummerOlympics, WikiPediaEntryFor,
" http://en.wikipedia.org/wiki/2004_Summer_Olympics "}
{1996SummerOlympics, WikiPediaEntryFor,
"http://en.wikipedia.org/wiki/1996_Summer_Olympics"}
/* Topics which are types for occurrences */
[AthletesCompeted]                          [MedalEvents]
[AthletesExpected]                          [Sports]
[DaysLasted]                                [StartedOn]
[EndedOn]                                   [WikiPediaEntryFor]
/* Associations */
heldIn(2004SummerOlympics : event, AthensGreece : city)
medalsAwarded(2004SummerOlympics : event, Medal_count:discussion)
organizationFor(2004SummerOlympics : event, Bid_and_preparation : discussion)
mostRecentCompleteOlympics(2004SummerOlympics : later, 1996SummerOlympics : earlier)
partOf(mostRecentCompleteOlympics : whole, CompleteOlympics : part)
partOf(mostRecentCompleteOlympics : whole, MostRecent : part)
/* Types and instances */
subtype-supertype(SummerOlympics : subtype, WinterOlympics : subtype,
OlympicGames : supertype)
type-instance(2004SummerOlympics : instance, SummerOlympics :type)
type-instance(1996SummerOlympics : instance, SummerOlympics :type)
```

```
/* Topics used as types for associations */
[exampleOf]                                    [mostRecentCompleteOlympics]
[heldIn]                                       [organizationFor]
[medalsAwarded]                                [partOf]
/* Topics used as types for association roles */
[city]                                         [event]
[concept]                                      [later]
[discussion]                                   [part]
[document]                                     [whole]
[earlier]
/* Reification of this topic map */
#TOPICMAP ~2004SummerOlympicsTopicMap
exampleOf(2004SummerOlympicsTopicMap : document, TopicMap : concept)
/* Import generic topic maps structures */
#MERGEMAP "topicMapsBuiltins.htm"
```

2. Construct a topic map equivalent of a substantial portion of the OWL ontology from the Chapter 11 exercise. Provide representations of all the different OWL constructs.

```
/* Topics */
/* Using the IOC web page as subject identifier */
[IOC = "International Olympic Committee" @http://www.olympic.org/uk/]
[LOC = "Local Organizing Committee"
        @http://www.olympic.org/uk/organisation/ocog/index_uk.asp]
[Federation = "International Sports Federation"
        @http://www.olympic.org/uk/organisation/if/index_uk.asp]
[Sport @http://www.olympic.org/uk/sports/index_uk.asp]
[NOC  = "National Olympic Committee"
        @http://www.olympic.org/uk/organisation/noc/index_uk.asp]
/* These topics have only LTM IDs. They are defined using occurrences */
[Athlete]
{Athlete, definitionOf,
        [["Person who competes in an event, either individually or as part of a team"]]}
[Competitor]
{Competitor, definitionOf, [["Athlete or team of athletes who compete in an event"]]}
[Discipline]
{Discipline, definitionOf,
        [["One of several sports managed by the same international sports federation"]]}
[Event]
{Event, definitionOf, [["An item in a program of sports"]]}
[FunctionalGroup]
{FunctionalGroup, definitionOf, [["A part of a LOC with a specific responsibility"]]}
[NationalTeam]
{NationalTeam, definitionOf,
        [["Group of athletes and officials competing under auspices of a NOC"]]}
[Official]
{Official, definitionOf, [["Person in a position of authority with respect to an event"]]}
[Person]
{Person, definitionOf,
        [["Human being, registered by the LOC as a participant in the games"]]}
```

```
[Race] /* Used to classify instances of competition units in racing events *]
/* The competition units themselves have unique identifiers and are declared as topics in the
type-instance section */
{Race, definitionOf, [["A contest of speed as a unit of competition in a kind of event"]]}
[RaceType]
{RaceType, definitionOf, [["A classification of races in an event"]]}
[Team]
{Team, definitionOf, [["Group of athletes who compete in an event as a unit"]]}
/* Occurrences */
{400mFreeMen, worldRecord, [[3:40:08]]}          {China, nGold, [[32]]}
{400mFreeMen, olympicRecord,                     {China, nSilver, [[17]]}
        [[3:40:59]]}                             {China, nBronze, [[14]]}
{1500mWomen, worldRecord, [[3:50:46]]}           {Australia, nGold, [[17]]}
{1500mWomen, olympicRecord,                      {Australia, nSilver, [[16]]}
        [[3:53:96]]}                             {Australia, nBronze, [[16]]}
/* Topics which are types for occurrences */
[definitionOf]                                   [nSilver]
[nBronze]                                        [olympicRecord]
[nGold]                                          [worldRecord]
/* Associations */
chooses(IOC : IOC, BeijingOC : LOC)
defines(FINA : Federation, Aquatics:Sport)
defines(IAAF : Federation, Athletics:Sport)
belongsTo(Aquatics: Sport, Swimming : Discipline)
belongsTo(Athletics:Sport, Track : Discipline)
authorizes(BeijingOC:LOC, SwimRaceOrg : FunctionalGroup)
authorizes(BeijingOC:LOC, TrackRaceOrg : FunctionalGroup)
affiliatedWith(ChineseOC:NOC, IOC : IOC)
affiliatedWith(AustralianOC:NOC, IOC : IOC)
selectsTeam(AustralianOC:NOC, Australia : NationalTeam)
selectsTeam(ChineseOC:NOC, China : NationalTeam)
represents(Xie:Person, China : NationalTeam)
represents(Lee:Person, China : NationalTeam)
represents(Thorpe:Person, Australia : NationalTeam)
represents(Johnson:Person, Australia : NationalTeam)
eventOf(400mFreeMen : Event, Swimming : Discipline)
restrictedTo(400mFreeMen : Event, Male : Gender)
eventOf(1500mWomen : Event, Swimming : Discipline)
restrictedTo(1500mWomen: Event, Female : Gender)
competesIn(Thorpe : Competitor, 400mFreeMen : Event, AQ001 : Race, Final : RaceType)
        ~ Entry01
{Entry01, finishes, [[1]]}
{Entry01, completesIn, [[3:41:03]]}
isOf(Thorpe : Competitor, Male : Gender)
competesIn(Lee : Competitor, 1500mWomen: Event, TR001 : Race, Semifinal : RaceType)
        ~ Entry02
{Entry02, finishes, [[2]]}
{Entry02, completesIn, [[4:05:10]]}
isOf(Lee: Competitor, Female : Gender)
medalWon(Thorpe : Competitor, 400mFreeMen : Event, Gold : Medal)
```

```
officiatesIn(Xie : Official, 400mFreeMen : Event, AQ001 : Race, Final : RaceType)
officiatesIn (Johnson : Official, 1500mWomen: Event, TR001 : Race, Semifinal : RaceType)
/* Types and instances */
/* Note that where there are a small number of subtypes, the whole subtype structure is
declared in a single association */
subtype-supertype(Athlete : subtype, Team : subtype, Competitor : supertype)
subtype-supertype(Athlete : subtype, Official : subtype, Person : supertype)
/* Note that in some cases several instances are declared in a single association. This is done
where there are a fixed small number of instances. Where there are many instances, they are
each declared in their own association */
type-instance([BeijingOC =
        "Beijing Organizing Committee for the Games of the XXIX Olympiad"
        @http://en.beijing2008.com/] : instance, LOC:type)
type-instance([FINA = "Federation Internationale de Natation Amateur"
        @http://www.fina.org/]: instance, Federation:type)
type-instance([IAAF = "International Association of Athletics Federations"
        @http://www.iaaf.org/] : instance, Federation:type)
type-instance([Aquatics
        @http://www.olympic.org/uk/sports/programme/index_uk.asp?SportCode=AQ]
        : instance, Sport:type)
type-instance([Athletics
        @http://www.olympic.org/uk/sports/programme/index_uk.asp?SportCode=AT]
        : instance, Sport:type)
type-instance([Swimming @http://www.fina.org/swimming/sw.htm] : instance,
        Discipline:type)
type-instance([Final] : instance, [Semifinal] : instance, [Heat] : instance, [RaceType] : type)
type-instance([Track = "Track component of Track and Field"] : instance, Discipline:type)
type-instance([400mFreeMen = "400 metre men's freestyle swimming"] : instance,
        Event:type)
type-instance([1500mWomen = "1500 metre women's track"] : instance, Event:type)
type-instance([SwimRaceOrg= "Swimming Race Organizer Subcommittee"] : instance,
        FunctionalGroup:type)
/* Topics used to identify competition units in Aquatics */
type-instance([AQ001] : instance, [AQ002] : instance, [AQ003] : instance, Race : type)
type-instance([TrackRaceOrg = "Track Race Organizer Subcommittee"] : instance,
        FunctionalGroup:type)
/* Topics used to identify competition units in Track */
type-instance([TR001] : instance, [TR002] : instance, [TR003] : instance, Race : type)
type-instance([Final] : instance, [Semifinal] : instance, [Heat] : instance, RaceType : type)
type-instance([Gold]: instance, [Silver]: instance, [Bronze]: instance, [Medal]: type)
type-instance([ChineseOC = "Chinese National Olympic Committee"
        @en.olympic.cn/sport_for/ nfp_project/2005-07-14/618393.html] : instance,
        NOC:type)
type-instance([AustralianOC = "Australian Olympic Committee]
        @http://www.olympics.com.au/]: instance, NOC:type)
type-instance([Australia = "Olympic Team of Australia"]: instance, NationalTeam:type)
type-instance([China = "Olympic Team of China"] : instance, NationalTeam:type)
type-instance([Xie = "Guo Tong Xie"]: instance, Official:type)
type-instance([Lee = "Ya Ting Lee"] : instance, Athlete:type)
type-instance([Thorpe = "Ian Thorpe"] : instance, Athlete:type)
```

```
type-instance([Johnson = "Michael Johnson"]: instance, Official:type)
type-instance([Male]: instance, [Female]: instance,  [Gender]:type)
/* Creates topics [Male], [Female], [Gender] */
/* Topics used as types for associations */
[affiliatedWith]                                    [isOf]
[authorizes]                                        [memberOf]
[belongsTo]                                         [officiatesIn]
[chooses]                                           [represents]
[competesIn]                                        [restrictedTo]
[defines]                                           [selects]
[eventOf]                                           [selectsTeam]
[exampleOf]
/* Topics used as types for association roles */
/* Notice that in the associations the design decision has been made to follow the UML
convention for naming association member ends by assigning as association role types the
topics representing the type of the member end. Compare the graphical representation of
the ontology in the solution to the Chapter 4 exercise. This convention is arbitrary, and in
fact is possible only when there are no cases of two associations between the same two
classes. */
[concept]
[document]
/* Reification of this topic map */
#TOPICMAP ~2008SummerOlympicsTopicMap
exampleOf(2008SummerOlympicsTopicMap : document, TopicMap : concept)
/* Import generic topic maps structures */
#MERGEMAP "topicMapsBuiltins.htm"
```

3. Comment on the differences between the OWL and topic maps representations.

Classes and individuals are both generally represented as topics. Object properties are generally represented as associations, where the domain and range classes are represented as association role types. Datatype properties are generally represented as occurrences. But in the OWL representation, the properties are all represented at a type level while in the topic maps representation, the associations and occurrences are all at an instance level. The instances would be represented in OWL using RDF triples conforming to the type-level constraints specified. Topic maps does not have a method of representing type-level restrictions on associations or occurrences.

The cluster of properties representing the participation of competitors and officials in events is represented in OWL as a collection of properties *competesIn*, *officiatesIn*, *in*, *completesIn*, *finishes*, *raceOf* and *typeOfRace*. Since topic maps allow n-ary associations, these relationships are bundled together in the associations *competesIn*, *officiatesIn* and the derived *medalWon* (see the solution for the exercise from Chapter 13).

Note that IOC in the OWL representation is shown as a singleton class. In the topic map representation IOC is a Topic only, not a class whose instances are other topics. The topic is used as a type for roles in the associations whose types are selects or affiliatedWith. It is possible in OWL DL to represent IOC as a bare individual, but in that case we could declare neither the domain of selects nor the range of affiliatedWith. In OWL Full, IOC could be represented as a bare

individual and the domain and range declarations would make it a class also. In this case the class *IOC* would be empty.

There is considerable information in the topic maps representation not included in the RDFS or OWL representations. This includes IRIs, topic names and the *definitionOf* occurrences. This is a design decision, not a fundamental difference between topic maps and RDFS/OWL. The IRIs could be used to represent the comparable objects in the RDFS/OWL version. Topic names and definitions could be included as additional statements. The reason for the design decision is that topic maps invites these kinds of statements while RDFS/OWL permits them but does not specifically invite them.

Ontology and the Semantic Web
R.M. Colomb
IOS Press, 2007

Subject Index

This page intentionally left blank

This page intentionally left blank