Use of Upper Ontologies for Interoperation of Information Systems:
A Tutorial
Robert M. Colomb

Technical Report 20/02 ISIB-CNR
Padova, Italy, November, 2002

National Research Council

Institute of Biomedical Engineering

ISIB-CNR

Corso Stati Uniti, 4

35127 Padova

Robert M. Colomb

School of Information Technology and Electrical Engineering
The University of Queensland
Queensland 4072 Australia
colomb@itee.uq.edu.au

Work performed partly while visiting LADSEB-CNR; Corso Stati Uniti, 4; Padova, Italia, and partly while visiting School of Mathematical and Computing Sciences, Heriot-Watt University, Edinburgh, UK

Abstract

This report is a tutorial on how formal upper ontologies can be useful in building systems which involve interoperating autonomous information systems. It begins with an analysis of the problems that need to be solved in order to build such systems, then looks at a material ontology of institutional facts and speech acts which accounts for most of the content of such systems. It then shows how the OntoClean meta-ontological system can help us make sense of the structures involved in the ontologies needed to support interoperating autonomous systems, then finally how the formal upper ontologies of DOLCE and the Bunge-Wand-Weber system can help clarify the content of the material ontologies needed.

Use of Upper Ontologies for Interoperation of Information Systems: A Tutorial

Robert M Colomb[1]
School of Information Technology and Electrical Engineering
The University of Queensland

24 October, 2002

Table of Contents

[1] Work done partly while visiting LADSEB-CNR; Corso Stati Uniti, 4; I-35127 Padova ITALY and partly while visiting School of Mathematical and Computing Sciences, Heriot-Watt University, Edinburgh, UK

1. Introduction

There are an enormous number of services available on the Internet. Accessing these services is often tedious and time-comsuming. Furthermore, the Internet is 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 Internet services. There are many such programs, for example:

1. Alerters which monitor publications for articles which satisfy an interest profile (or perhaps monitor chat room conversation openers in a similar way)
2. So-called meta-query facilities which broadcast a query to multiple information sources, producing a combined result (multiple libraries, multiple Web search engines).
3. Shopping bots which search multiple sources of standard products like books, CDs or software for best price or fastest delivery or some other criterion.
4. Auction bots which participate in auctions for designated objects, making bids implementing a bidding strategy.
5. Purchasing agents which execute complex purchasing protocols on electronic commerce exchanges.

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

1. 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.
2. Tendering for complex products or services, possibly assembling a whole purchase from partial tenders.

The purpose of this note is to explain how these agents can be constructed, and what resources are needed to construct them. We begin with analysis of a simple example.

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.

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"

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”

The first three results are bookseller trade organisation 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 specialised 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.

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 generalisation 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.

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.00

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*.

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 standardised message types and schemas, it becomes a relatively simple matter to implement a purchasing agent.

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:

1. A trusted list of sources with which to communicate
2. A standard set of types of messages with agreed semantics
3. 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.

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 specialities 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 specialise in chess books or used comics or the occult. Several of the booksellers in Figure 1 are pretty specialised. There is also an enormous range of people buying books – after all, each of the specialised 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 specialities or for the exchanges to have a deep taxonomy of classes of publication, very likely both. If we have lots of specialised exchanges, then we probably need exchanges of exchanges organised into a deep hierarchy of classes.

Besides being more specialised, 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 specialised outlets, and would have their own range of more or less specialised exchanges. The exchanges need to be organised 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 each has its 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 organisations, and where the products are far more specialised and the trades much more complex. B2B exchanges tend to be in specific industrial sectors and to include a complex structure of specialised 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 specialised sources of engineering, architectural, legal and financial services.

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

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. Fulfilment 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 fulfilment 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

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.

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 organisations. Besides appearing in the messages, the product details would generally appear in the same or closely related form in the *Product* tables of both organisations. 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 organisations are involved, in order to make sure that the subject of the communications is what both organisations 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.

2. 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. 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.

What we talk about in information systems

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.

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 organise 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. 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.

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 deterimining 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 records the order, 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 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 insitutional 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.

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 recognise 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. 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 labelled 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 a tabulation of the things existing in the environment of the two systems, the interoperating agents need to share an understanding of the background behaviours and practices underpinning the operation of the systems involved.

Build or buy

There are many interoperating communities of applications, and therefore many already existing ontologies. It is often cheaper and faster to base a new exchange on existing ontologies, perhaps with some modification, than it is to build an entirely new system from scratch. This section is quite sketchy. It is intended that the reader consult other readings to get a clearer understanding of any material with which they are unfamiliar.

Already existing ontologies

Some of these already-existing ontologies are message-oriented. In particular, the Z39.50[2] protocol specifies message types and background for requesting information, retrieving information requested, for making additional queries on the results of previous queries, and for specifying alerters, which are ongoing queries against a stream of new documents. The business message sequence *RFQ*, *Quote*, etc. comes from the Electronic Data Interchange (EDI) standards, which have been in use for many years and are being adapted for interoperating web site use by a number of organisations. These message-oriented ontologies also specify mandatory and optional fields which can occur in the messages.

Some of the ontologies are not lists of terms but lists of types of terms (attribute names). There are several such lists associated with the Z39.50 standard, the best-established of which is a standard terminology called *Bib-1* for making queries on library collections. Others are collections of instance terms, such as the SNOMED system which contains terminology for constructing records of medical consultations. The larger systems are generally organised as hierarchies of broader and narrower terms.

Most ontologies are intended to operate in a particular domain of interest, so would be of primary interest only to those working in those or related domains. The SNOMED system isn't much use for organising chess book catalogs, for example. There are a number of efforts intended to build universal ontologies that could potentially be used by any application, including Cyc[3] and SUMO[4]. It is not clear that these universal ontologies will be useful for interoperating information systems (they are developed in the Artificial Intelligence community and tend to be oriented towards natural language applications). A further widely known system of English words classified by syntactic and semantic categories is WordNet,[5] which is sometimes used as an ontology.

Finally, there are many systems of classifications working in the space of classification systems within and for exchanges. Prominent on the web is the Yahoo system. The publishers of the Dewey Decimal System widely used in libraries promote their classification system for web use. Most telephone systems come with a Yellow Pages directory of businesses by type. A smaller, hierarchically organised system for classfiying businesses is the Standard Industrial Classification (SIC) system published by the US Department of Labor. (Italian Yellow Pages directories are indexed by a hierarchical system based on a structure like the SIC.)

Hierarchies vs facets vs grammars

Most ontologies larger than a hundred terms are organised in some way. Most common is a simple hierarchical (or sometimes directed acyclic graph) structure, like Yahoo, the major library sysems, and the SIC. Some systems, generally more recently created, are composed of several smaller classification systems, called *facets*, which are orthogonal to each other, like the classic menu in a Chinese restaurant. These systems are easier to understand and to maintain, and they can be much larger than strict hierarchical systems in terms of potential leaf nodes. The SNOMED system is an example. SNOMED has about 150,000 terms organised into 11 facets, and it is possible to construct 10^{40} classifications. The Library of Congress System has nearly 1 million classes, but can have only the same number of leaf nodes.

Some faceted systems are organised in such a way that the non-leaf classes function as syntactic types in formal computer languages. These systems can be easily represented as XML Document Type Declarations. Some EDI systems work this way, as do some widely used medical records systems[6].

3. Ontologies for use with agent systems

The ontologies described in the previous chapter are generally intended for use by humans. The EDI messages are essentially automation of human-readable records. We are in this reading concerned with ontologies which can be used to support interoperation of more or less complex agent software. We need

[2] http://www.niso.org/standards/resources/Z39-50-200x.pdf
[3] http://www.opencyc.org/
[4] http://ontology.teknowledge.com/
[5] http://www.cogsci.princeton.edu/~wn/online/
[6] http://www.nlm.nih.gov/pubs/factsheets/umls.html

for the agents to be able to reason with the information structures, and we need to be able to write software which can be reused in many different systems. The existing ontologies tend to be weakly organised, and do not support complex reasoning.

It is difficult to reason with a untyped hierarchical system of terms. A classic example from the field of Artificial Intelligence concerns Clyde the elephant. The term “Clyde” occurs below the term “elephant” in a hierarchy, because Clyde is the name of an elephant. The term “elephant” occurs below the term “species” in the hierarchy, because “elephant” is the name of a species. One might make the inference that “Clyde” is below “species” because “Clyde” is the name of a species, but this inference is false. The problem is that in both relationships the narrower terms are instances and the broader terms sets of which the narrower term is a member, and the instance relationship is not transitive.

If we are going to have agents with complex programs interoperating in rich ways, we need the integrating ontologies to have a rich structure which supports complex reasoning. Ideally, we would like also this rich structure to be largely independent of the particular application, so that programs written for one area can be adapted easily to another.

Complex objects

Our information systems are often about things which have a complex structure. For example, for many purposes an aircraft (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 accomplishments (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 represent it as a collection of fragments. 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*.

Representation of identity and unity in a single information system

Single system example

Before we look at the issues of identity and unity for interoperating information systems, we will look at how these are represented in single information systems. Figure 2 shows a fragment of an information system supporting an order-entry/ order fulfilment 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, showing a distinction between the interaction with the customer (*Transaction*) and the processes that the organisation must undertake to fulfil the order (Activity). *Purchase Transaction* depends on a customer and a product, while *Activity* depends on a *Purchase Transaction* and a *Product*. In terms of Searle’s framework of institutional facts, 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.

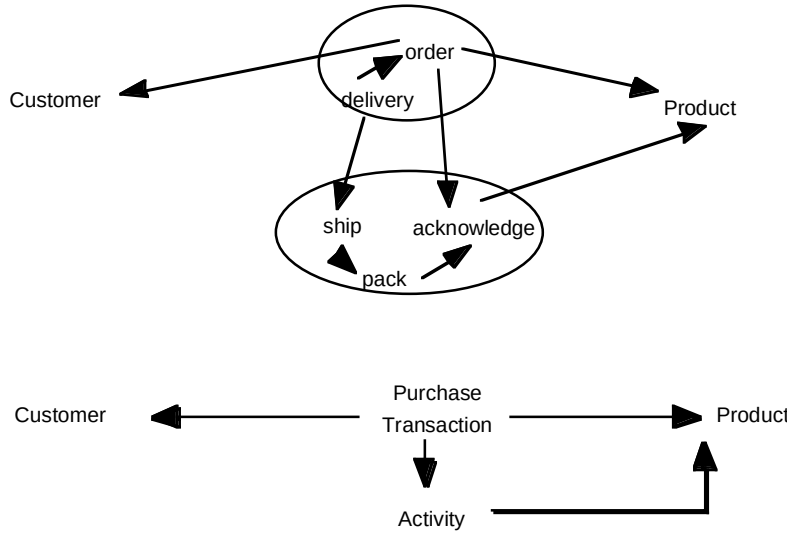

Figure 2: A fragment of a refinement of an order-processing system

The upper part of the Figure shows a refinement of the lower model, showing both *Purchase Transaction* and *Activity* as processes with parts. *Purchase Transaction* is an order followed by a *Delivery*, while the *Activity* process is first to acknowledge the order, then pack it, then ship it to the customer. The relationships between *Order* and *Acknowledge*, and between *Delivery* and *Ship*, are both refinements of the relationship between *Purchase Transaction* and *Activity*. This refinement illustrates that institutional facts can be viewed at coarser or finer granularities, depending on the purposes of the viewer.

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, in the OntoClean system it is not necessary for an entity to have unity in order for it to have identity. A bulk product has identity - we can identify water, eg, without having any representation of the unity of water.

However, a structure composed of parts can't have a single identity unless it has unity. We must be able to tell which parts belong to a particular whole. The OntoClean axioms for unity are developed using theory of wholes and parts, called mereology[7]. A whole is a collection of parts

P(x, y) is true if x is a part of y (1)

An object x is an integrated whole if it has a division into parts that is a closed system under a suitable equivalence relation[8] ω called the *unifying relation*. The axiom given is

$\forall y(P(y, x) \rightarrow \forall z(P(z, x) \leftrightarrow \omega(z, y))$ (2)

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.

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

$\phi(x)$ and $\phi(y) \rightarrow (\rho(x, y) \leftrightarrow x = y)$ (3)

[7] See Mereology in Explanations
[8] See Equivalence Relation in Explanations

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 with institutional facts, therefore the entities of concern 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-computerised 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.

Application to the example – individual objects

We first look at a representation of a single entity which is the source of no many-to-one relationships, in the example of Figure 2 *Customer* and *Product*. (We will call these universal targets *independent entities*.) The semantics of the ER model are based on set theory, so the principal interpretation of the representation of the entity 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”. Since we don’t yet have an identity criterion, the unifying relationship is represented lexically rather than logically as a list of attribute values, say, or as a set of attribute name/value pairs rather than as variables in a logical formula or an SQL query[9].

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). 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 at least partly lexical.

In database theory, an instance of an entity 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 *product number* or *customer identifier*. So in the case of *Product*, an identifying relation can often be represented logically as

$$\rho_1(x, y) =_{df} \text{product number}(x) = \text{product number}(y) \qquad (4)$$

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

However, the identifying relation (4) is not sufficient. It works well for the internal working of the organisation, but is not generally a satisfactory way of representing identity outside the particular organisational context. 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.

[9] See Logical/Lexical in Explanations

There could be a number of such candidate keys. For example, in the United States an organisation 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.

A significant point emerges here. What the information systems designer thinks of as an instance can be from the perspective of the ontology a type, 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 types 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 type. This identifying plate is often the only difference between otherwise indiscernable objects of the same product number.

The citric acid example isn't like that, though. One need for identifying an instance of this product type 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 type. 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 (eg 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).

OntoClean uses the terminology that a type has a property that *carries* identity or unity if all instances of that type 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 below, types generally occur in subsumption hierarchies, where each more specific level can introduce new properties. The type 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 type is said to *supply* identity or unity, respectively.

This gives us a taxonomy. In terms of OntoClean, the type *Customer* is a collection of individuals. (An *individual* has both unity and identity). The entity *Product* is a collection of types, which may or may not consist of individuals. The individuals may be distinguished only by name. But we have seen that the types in *Product* often carry identity (as an instance of the identified type), but not unity so that the lots of product are not able to be identified other than by the container they may be in. We can call these properties (which carry type identity but not unity) *bulk* properties as distinguished from *countable* properties (which carry both identity and unity so yield individuals).

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 type may be "Apple – Granny Smith in cases of 80". The cases carry no identification. So at the level of maintenance of inventory, shipping and receiving, the product is 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 type, 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.

Dependent entities

In information systems applications entities like *Purchase Transaction* which are the source of mandatory many-to-one relationships (we will call *dependent* entities) pose a somewhat more complex problem. The representation of these entities (still referring to business objects) will generally include identification of the business objects represented by entities which are targets of the mandatory many-to-one relationships. Semantically, the independent entities define the most stable aspects of the operation since they must have instances in order for the other entities to have instances. The dependent entities represent records of the more dynamic aspects of the operation. The institutional facts referred to by the populations of the dependent entities have as some of their properties the identifiers of the records of other institutional facts. These other institutional facts exist for the speech act creating the present institutional fact to be validly framed, so these records are important properties. An invoice may require a purchase order, a delivery advice and a delivery acknowledgement, for example.

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 independent entities which are appropriate to the particular context. (This is essentially the formal mechanism of *weak entities* in ER modelling.)

In our example, *Purchase Transaction* is dependent on both *Product* and *Customer*. The third dependency, on *Activity*, 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, so this dependency does not add anything.

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*.

In this example, the instances of *Purchase Transaction* are all individuals. Given the need for accounting and audit, it is hard to see how an instance of *Purchase Transaction* could be a bulk property.

Application to the example – classes

Instances have now been identified as instances of their respective entities. This leads us to ask how the entities are identified. Entities represent classes of business objects, and a class of business objects is often the subject of discourse.

Institutional facts are completely characterised 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 an entity, which formally is a set whose members are instances, but the normal practice is to define the entity 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 entity can have many different populations – indeed much of the code in an information system is devoted to updating the populations of entities. That is to say, much of the code is devoted to updating the representations of members of classes of business objects.

In practice, systems designers identify entities in several different ways. If the primary design vehicle is the ER model, the 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 entities. Often, the representation is a translation of the conceptual representation to a relational database table scheme, where the entity name becomes the table name, and the relationships and attributes column names, the former labeled foreign keys.

Other ways also are used. Sometimes several entities 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 an entity, the programmer needs to know which columns contain the preferred candidate key and which columns contain the attributes and foreign keys associated with that entity. It is also possible to represent a conceptual model in a single table with four columns: *entity*, *tuple*, *attribute*, *value*. Each row of this table contains a single value of a single attribute of a single tuple instance of a single entity. (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 entity in order to retrieve its instances.

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

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 organisation, so would have a name, say *Acme Corp Ordering System* (henceforth ACOS).

Either system in Figure 2 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.

However, it is not difficult to represent a unifying relation. One way is to supply an additional entity *ID* called the identifying entity, with exactly one instance and to require a (unique) many-to-one relationship *id* from each entity in the system to the identifying entity. We can parameterise the identifying entity by the name of the system, represented as ID_x. An entity used as an identifying entity 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

$$\omega_x(z, y) = \exists z,y \ (\mathrm{id}(z, \mathrm{ID}_x) \text{ and } \mathrm{id}(y, \mathrm{ID}_x)) \qquad (5)$$

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

Given the unifying relation (5), we can represent an identity criterion for "is system ACOS" as

$$\exists x \ \mathrm{ID}_x \text{ is a terminal object for system ACOS} \qquad (6)$$

Because they are both parameterised 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.

Interoperating systems

An order processing system would naturally interoperate in an electronic commerce environment with a purchasing system operated by another organisation. Further, neither system would generally expose itself in its entirety to the other. Each system in the interoperation would see something like in Figure 3. As with Figure 2, 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 a 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.)

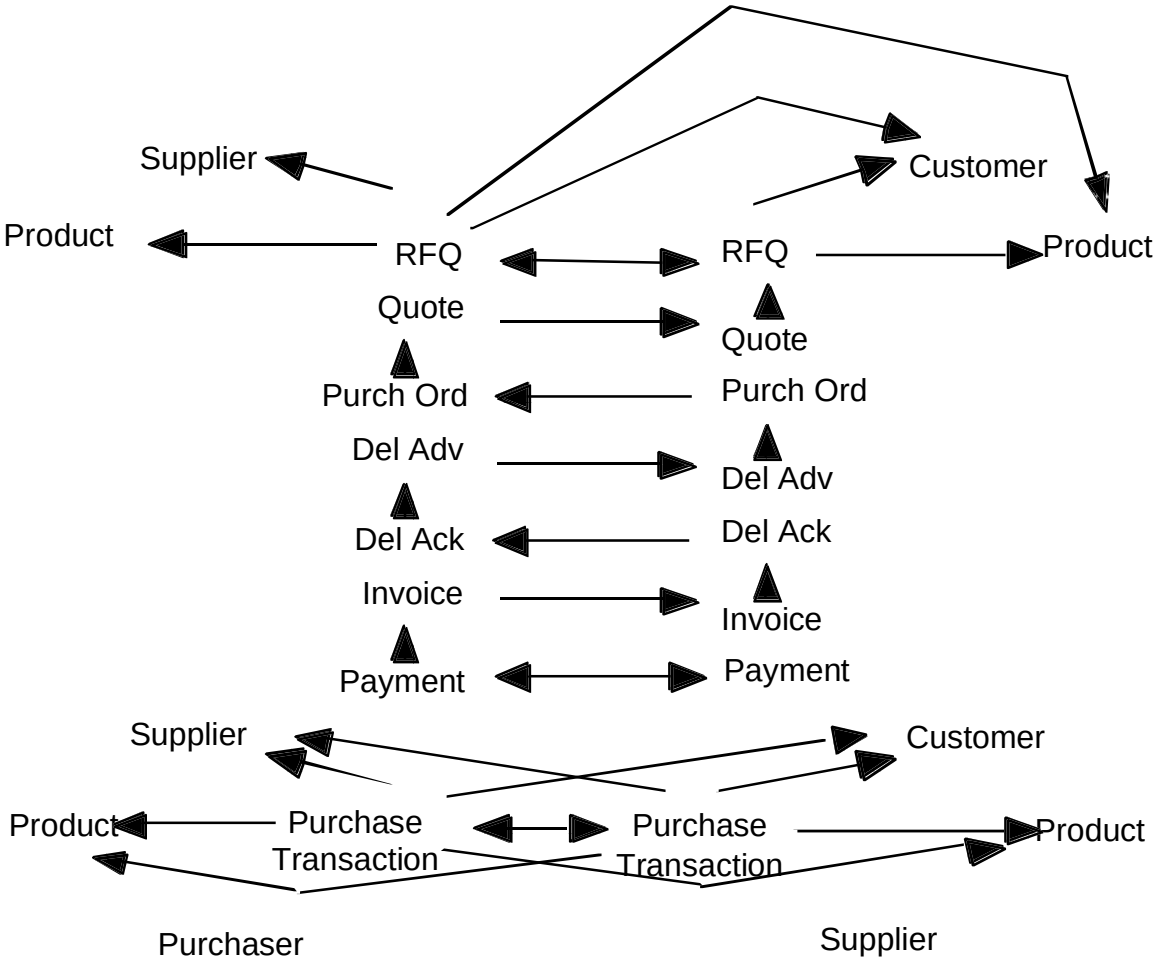

Figure 3: Purchasing and Supplier system interoperating

Still concentrating on the lower (abstract) system, note that on each side *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 2, but is closely related.) We focus first on the relationships involving *Product* on each side.

Product in each system is an independent entity. 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 subtype of products shared between a given customer and supplier.

The relationships involving *Supplier* and *Customer* raise additional issues. It is normal for the Purchaser system to have an entity *Supplier* (as in Figure 2), and for the Supplier system to have an analogous entity *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 Suppler 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 an entity of the other, in order for the two systems to interoperate.

We move now to the refinement of the *Purchase Transaction* entities in the upper part of Figure 3. A business transaction is an institutional fact normally created in a series of speech acts organised 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 3, 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 types, 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 last message (*Payment*) has a similar one-to-one relationship representing acknowledgement of the closure of the interaction.

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 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* entity.

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* (representing the whole by one of its parts), 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 entities 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 organisations 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 subtypes and subsumption in the next section.

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 a unifying relation, so that some things in the environment are what we might think of as bulk rather than distinguishible objects. We have seen that our information systems deal logically both with definite objects and with types. We have used the concept *type* in several places, relying on its usual interpretation in database theory and practice, and in logic. We now look more deeply into how types work, and the conditions under which our agents can make use of them.

Subtypes and subsumption

A *type* in database or logic is a set of things which is represented according to a schema. In an ER model, entities are types, as are some relationships. In a database a type is represented in possibly several tables held together by unifying relations and identified using identity relations. The representation of a type in a logic system is similar to its representation in a database. All of the entities in Figures 2 and 3 represent types. The set of web pages indexed by a particular search engine is a type. In particular, the set of services covered by an e-commerce exchange is a type.

Qualities and properties

An instance of a type is represented by a collection of qualities. The properties representing an entity type are its attributes. In database or logical representations, properties are represented by attribute values in relations. A *quality* is a specific value of a property.

In a tradition going back to Aristotle, it is common to think of a type as having two sorts of qualities, essential and accidental. An *essential* quality of a type is a quality that a thing must have if it is to be a member of that type. An *accidental* quality 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 quality of a human being, while *having red hair* might be considered accidental. *Having feathers* might be considered an essential quality 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 quality of that object. So it is possible for something to be an *essential whole* if its unifying relation is an essential quality. Once the whole exists, it is always a whole. In Figure 3, 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 quality and property can be seen in this example. *Having hair of a colour* is a property of the type human being, the boolean *ability to fly* is a property of the type bird. The concept of property applies mainly to accidental qualities, since an instance of a type can have different qualities drawn from the value set of the property. An essential quality must be the same for all instances of a type, by definition. However, an essential quality of a type might be an accidental quality of a supertype, so it makes sense to think of properties here, too. *Having a brain* is a boolean property of a supertype of human being that includes dead people as well as living. *Having feathers* is a boolean property of a supertype 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 2, 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 quality 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 type might not have a quality drawn from an optional property, so an optional property cannot be essential.

We have that an essential property for a type is necessarily present for every instance of that type, and always has the same quality value. There is a related, but stronger, kind of property called a *rigid* property which is not only essential, but sufficient to determine the type. 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 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 fenceposts 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 characterised 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 characterise it as a diamond. It is an essential but not rigid property. A rigid property is something like a candidate key for a type. All instances of a type have the same qualities drawn from the type's 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 3. The type 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 computerised 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 types of institutional facts are largely lexical, and are often essentially the same thing as the name of the type.

Subtypes and subsumption

Types are interesting mainly because we can have subtypes. Type B is a *subtype* of type A if every instance of type B is also an instance of type A. We say in this case that type A *subsumes* type B, and that A is a *supertype* of B. The population of instances of a subtype is a subset of the population of instances of a supertype. There are two different ways of defining subtypes, called defined and primitive[10].

The defined subtype is most common in database systems. A *defined* subtype is the result of a predicate on the supertype, so it is defined logically. We may have two subtypes of *Customer*: *Individual* and *Organisational*, distinguished by the value of an attribute called *Customer Type*. We may have two subtypes of *Product*: *Inventory* and *Service*, distinguished by whether or not there is an inventory record for the product. We may have two subtypes of *Student*: *Passed* and *Failed*, distinguished by whether the grade is respectively greater than or equal to 4 or less than 4.

A *primitive* subtype is determined by a declaration, that is lexically. We may have separate tables for invoices and payments but declare them both as primitive subtypes of *Financial transaction*. Hierarchical term lists found in classification systems, such as discussed in Chapter 3 are defined as primitive subtypes.

Unity, identity, necessity and rigidity are metaproperties closely related to subsumption. These metaproperties are called *formal* properties. 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.

Recall that we say that a type supplies a property if it holds for that type and not for any supertypes, while a type carries a property it inherits from a supertype. The same terminology applies to qualities and to the meta-properties of qualities and properties: identity, unity, essentiality and rigidity. The OntoClean method annotates properties and qualities with their formal meta-properties:

[10] This terminology comes from description logics

1. identity +I
2. unity +U
3. essentiality +M
4. rigidity +R

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 type the property necessarily has meta-property A. So the annotation +R on a property of a type says that the type carries rigidity. Recall that rigidity means that all instances of the type have the same quality 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 –M is not essential for all instances, so that different instances can have different qualities. The sign '-' indicates 'not', so –R is 'not rigid', -M is 'not essential', -I is 'not identity', -U is 'not unity'. If a property is annotated ~A, then the meta-property is not necessary for any instance. A property annotated ~M can be updated, so that the quality associated with any instance may change. The sign '~' indicates 'anti', so ~R is 'anti-rigid', ~M 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 +, -, ~ M/ R and mandatory/optional is

1. +R must be mandatory (note +R -> +M)
2. +M must be mandatory (note ~R -> ~M, so +M cannot be ~R)
3. -R, -M, ~R, ~M can be optional

Metaproperties and subsumption

These meta-properties regulate subsumption. A property of a type can never be weakened in a subtype. A rigid property (+R) of a type has the same quality in all instances, so must have the same quality in all instances of all subtypes. A necessary property (+M) also has the same quality in all instances, so also must have the same quality in all instances of all subtypes. A property which carries identity (+I) can identify any instance of a type, so must be able to identify any instance of any subtype. A property which carries unity (+U) is the basis of keeping together the fragments of any instance of the type, so must be able to keep together any instance of any subtype.

By similar reasoning, a subtype can never have a property ~A if the supertype has that property +A. Further, a subtype can never have a property +M, +I, +U if the supertype has that property ~M, ~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, and an anti-unity property is not capable of being the basis of a unifying relation for any instance. However, a supertype can have an antirigid property which is rigid for a subtype. The conventional *Customer Type* attribute is anti-rigid for the type *Customer*, but rigid for the subtypes *Individual Customer* and *Organisational Customer*, having e.g. the value 'I' and 'O' respectively.

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

Some of the consequences of these regulations are:

The set-instance relationship is not a supertype-subtype 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 eg "table name *invoice*", which is ~I for individual invoices since it is always the same. The instances have the +I property eg "invoice number", which is ~I for the set of invoices, since it has many different values. The same argument works for the Clyde the elephant example above. 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.

The meta-relationship is not a supertype-subtype relationship. Imagine we have an ER diagram with no subtypes, so all the entities represent types with rigid properties. One might be tempted to create a single supertype of which all the entities are subtypes on the basis that all the entities have rigid properties. This fails for essentially the same reason that the set-instance relationship fails, namely that the identity criterion for the proposed supertype is that the proposed subtypes all have a rigid property. This criterion cannot identify instances of the individual entities.

The whole-part relationship is not necessarily a supertype-subtype 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 2 as a supertype-subtype 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 supertype-subtype 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 +M +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 supertype-subtype relationship.

Interoperation example

We will now illustrate our new terminology by re-interpreting the electronic commerce interoperation example from Figure 3. 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* entities 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* types, only the identifiers. The rigid property of the *Product* type 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 type 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 type 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 types for both purchasers and suppliers, which requires a speech act establishing the correspondence between the individual system identifiers and the instance identifiers of the corresponding types, in the same way as the *Purchaser* and *Supplier* cases. The two types in the exchange's system may have overlapping subtypes, namely identifiers of organisations who are both purchasers and suppliers. The anti-essential instance-level identifying property will be an equivalence class of the two correspondences 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 organisation. This is a defined subtype. The exchange may or may not publish its supplier, purchaser and overlapping types.

We have established a type associated with each pair of supplier and purchaser which is the products in common between the two. Its rigid property will be a name like ‘product’ qualified by both the supplier’s and purchaser’s identifiers. Its instance-identifying property will be the anti-essential pairing of the supplier’s and purchaser’s *Product* identifiers. We can aggregate these pairwise subtypes into a product catalog for the entire exchange by taking the union of all the pairwise subtypes. The instance identifier will be an equivalence class where two pairs are equivalent if they share either a purchaser’s or supplier’s product identifier.

The resulting exchange-level product catalog will be a catalog of all products actually traded on the exchange, where products of individual suppliers are equivalent if their identifiers are each equivalent to a purchaser’s product identifier. This is essentially the collection of product equivalence institutional facts established by each of the supplier/purchaser pairs. There may be products of different suppliers which are in fact equivalent but which no supplier has declared so, and products which no supplier has considered, so the resulting catalog may be considered to be incomplete.

If the exchange wanted to have a unified product catalog, it would be necessary to make the speech acts to individually relate the products of each catalog to each other. Each instance of a product in the unified catalog would be an institutional fact consisting of an equivalence class of instances of products in individual suppliers’ catalogs. It is very common in such catalogs for there to be a taxonomy of subtypes 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 taxonomy based on product properties is orthogonal to a taxonomy based on supplier and purchaser, so the resulting subtype structure of a unified product catalog would be a faceted system.

Finally, the exchange operates by means of the message types indicated in Figure 3. The seven message types indicated in the upper part of Figure 3 may be considered as a decomposition into parts of the message type *Purchase Transaction* of the lower part of the Figure. Both the parts and the whole are types, but we have seen that the parts are not subtypes of the whole.

Figure 3 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.

The fact that a transaction’s parts come into existence over time, and in fact may never come into existence, suggests a state view of the transaction, as illustrated in Figure 4. Each successive state is reached on receipt of the next expected message, or on a speech act reflecting the decision by the organisation that the message will never come. In the second case the transaction is finished, in the sense that no more parts will come into existence, but failed (*Failed Quote, Unable to Ship, Bad Debt*). The state where a whole transaction exists is *Complete*. The other states are all non-terminal, awaiting either a message from the other party or a report of an action by the owning party.

From a data perspective, the successive states look very much like successive subtypes. Each state includes all the data of the previous state, plus more. Each non-terminal state includes a disjunction of possible continuations. Each continuation implies the disjunction. Thus the states are subtypes both from a set-theoretic (as one finds in databases and programming languages) and a proof-theoretic (as one finds in description logics) perspective. The rigid identifier of each subtype is the subset of parts which exist.

However, the names of the states do not follow the semantic abstraction hierarchy normally used in exposition of subsumption hierarchies (thing, physical object, person, student, student in my class, etc.). In this kind of application, the states are named in a metonymic fashion. (*Metonymic* naming is naming the whole after a part.) There is no requirement that the ‘normal’ natural language semantics of the names of subtypes be in subtype relationships in ‘normal’ natural language.

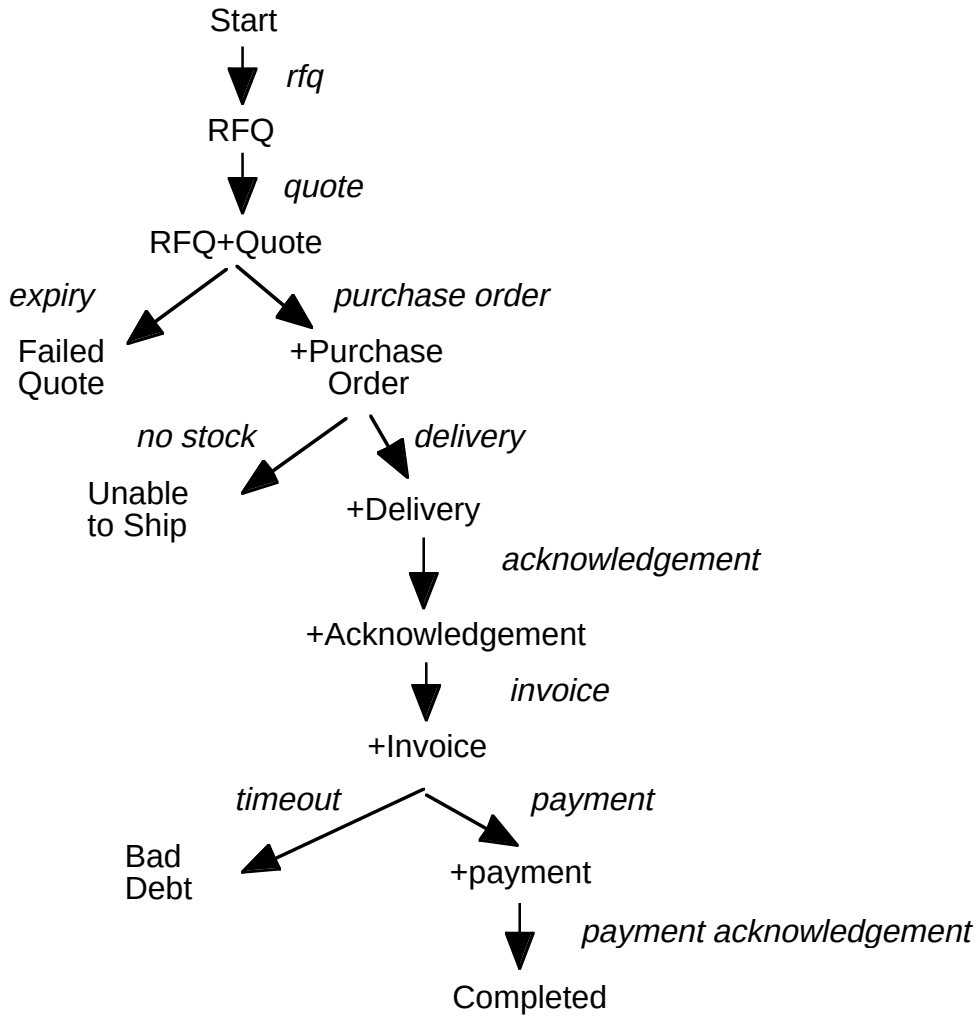

Figure 4: A state view of the coming-into-existence of parts of a transaction

If we take such a subtype view, we can take advantage of the preservation of identity through subtypes in the OntoClean system. Identity of a completed transaction can be supplied by *RFQ* or *+Purchase Order*, and carried down through the hierarchy. The failed transactions also can get their identity in this way – *Failed Quote* from *RFQ*, *Unable to Ship* and *Bad Debt* from *+Purchase Order*, for example. So also can the various incomplete but still active states.

A more complex example

Our first extended example was based on an e-commerce system, where the subtype 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 subtype a product belongs to is determined by a mixture of primitive and defined types 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 type "aircraft". The first thing we need to do is to establish that there is something "outside" any of the systems that corresponds to the type *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 type 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 type corresponding to both *Customer* 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 type 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 organisations 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 type *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 (what do you do with *Blacklist*?).

If we are going to have a type, it needs a rigid property. Since we are making a type 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 constituient 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 subtype 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 subtype of *Aircraft*?

We have that a view can be considered a subtype if all the mandatory properties of the supertype are in the view, necessarily including the rigid property for the type and the anti-essential instance identifying property. So far, we have the rigid property for the type 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 supertype?

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 subtypes.

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 subtypes and parts at the same time? The argument given above is that the whole-part relationship is not a supertype-subtype 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 subtypes.

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 subtype/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 subtype/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.

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 supertype-subtype 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 reliabily than without, and can more confidently build widely distributed interoperating systems with rich structures permitting interaction among complex intelligent agents.

4 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. As mentioned in the Introduction, 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.

It is extremely doubtful that these universal ontologies can be used as the basis for ontologies supporting interoperating information systems, because 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.

That similar terms can differ significantly in meaning from system to system is called *semantic heterogeneity*. The arguments that semantic heterogeneity generally defeats attempts to integrate autonomous information systems are either blindingly obvious (to people with experience in trying to put systems together in practice) or strange and difficult (to people without such experience). Students especially can be expected to lack relevant experience, and should consult the supplementary readings.

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 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 the previous Chapter. These ontologies are different, but are compatible with each other. Each emphasises different aspects of form.

BWW system[11]

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.

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.

[11] This section adapted from the notes to the University of Queensland course CO362: Business Information Systems; Module 2: Systems Concepts developed originally by Prof. R. Weber.

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.

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.

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.

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 Softdrinks Ltd might be coupled because the citric acid is an ingredient of the softdrink. Sales of the softdrink 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*.

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 2 is a system in this ontological sense. So also are each of the interoperating purchaser/supplier systems of Figure 3 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 3 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 3 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:

6. Its composition is a subset of another system's composition. In other words, all things in the subsystem are also things in another system.
7. Its environment is a subset of the environment of the other system joined with the difference between the composition of the other system and composition of the subsystem. In other words, first we take the things that are in environment of the system. To these we add the things that are in the composition of the system 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.
8. Its structure is a subset of the other system's structure. In other words, all internal couplings and all external couplings in the subsystem are also internal couplings and external couplings of the other system.

Each of the purchaser, supplier and transaction fragments of Figure 3 is a subsystem.

10. Input and Output

The *input* of a thing is the set of state changes (events) that have arisen to 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 3. 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 organisation 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.

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). 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.

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 3 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.

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. 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.

DOLCE system

The DOLCE system also consists of a number of concepts, which are organised into a hierarchy. Many of the classes in the system are based on the part/whole relationship[12].

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.

    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.

    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 (eg 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.

3. 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.

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

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

4. 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 3 is an occurrence, with the temporal parts *RFQ*, *Quote*, etc. In general, the speech act which creates an institutional fact is an occurrence.

    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.

        3.2.1 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.

[12] See Mereology under Explanations

3.2.2 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.

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 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 *heriditary*/*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.

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 3 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.

Providing a rich vocabulary

In this note we have pursued an extensive electronic commerce example, which has been presented in three different ways. In Figure 2, we have shown the structure of the individual systems, in Figure 3 their interoperation and in Figure 4 the possible states of their interoperation. The tool used to show this structure is Enhanced Entity-Relationship (EER) modeling. It could equally have been shown using Universal Modeling Language (UML) or one of many other tools of a similar kind.

These tools generally consist of a small set of primitive modeling objects and a syntax for constructing complex structures from the primitives. EER has the primitives *entity*, *relationship*, *attribute* and *subtype*. The various UML tools are organised in a similar way. These tools are flat, in that although it is possible to develop models of great complexity in them, there are no intermediate levels of description between the primitives and the entire model. There are a few tools that do permit intermediate levels of structure, for example Data Flow Diagrams (DFD). However, DFDs have a self-similar breakdown, so that the different levels use the same vocabulary, and that vocabulary is also quite limited. Its primitives are *process*, *external entity*, *data flow* and *data store*. Further, the primitives used in DFD do not relate well to the

primitives used in EER or UML. DFD stems from a top-down design approach, while EER and UML stem from a bottom-up approach, and there is not really a suitable intermediate representation.

If we use a conceptual modeling systems like EER, we can specify a collection of atomic types with a rich set of relationships governed by a rich set of constraints (cardinality, subype, etc.). While these systems provide a good model of the contents of a database, they do not describe individual complex objects very well. It is therefore difficult to use them to represent the complex objects that may be shared among information systems. The reason is that even though these objects are integrated with the other objects in the databases supporting each application, they must be able to be detached from one system and re-attached to another. Figure 3 taken as an EER diagram is an example. How are the various message types separated from each other and the environment of *Supplier*, *Customer*, *Product*?

We have already seen in the discussion around Figure 4 that the meta-properties can lead us to the somewhat surprising representation of the messages of Figure 3 as a network of states and transitions of Figure 4. Thinking in terms of a formal upper ontology can make this representation obvious. In the following, we will use the DOLCE system, augmented with elements of BWW.

Figure 2 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* are both perdurants, subclass event.

Now a perdurant or occurrence exists in time, and can have temporal parts. Once an occurrence or part of an occurrence has past, 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 state represented as 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, while the dependencies among the temporal parts of the events represent states where the system is receptive to certain events. A delivery event is only recognised by the system if an order event has previously occurred.

Some of the occurrences/ events in Figure 2 are internal to the system (*delivery*, *pack*, *ship*) and some come from the environment of the system (*order*, *acknowledgement*). Figure 3 shows two systems, each of which is an environment for the other, together with their coupling as a class of occurrences (events) represented by *purchase transaction* and its articulation into temporal parts. Again we want to keep a memory of the events, which we do both as records (objects) and as state, which is essentially what Figure 4 shows. The states have a complex relationship, which is represented both as accumulations of records (the subtypes) and as receptivity to events (the subtype relationships can be read as state transitions). The state as receptivity to events is represented by emergent properties of the subtypes. For example, the state of a subtype can be calculated from the record of events by finding the event which has no other event depending on it, that is by traversing the dependency chain to its end.

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 meta-properties (unity, identity, rigidity, essentiality, etc.) hold the more specific representations together and maintain the relationships with the more general. Further, the semantic distinctions between endurant and perdurant and their subclasses increases our understanding of the system and its parts. The main deficiency is that there is not a good theory of how the dependencies behave under decomposition, so there is some mushiness. There are candidates for such theories, but they are complex and not widely accepted at present. In any case the mushiness remaining is tolerable.

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)[13], developed by a consortium of web-based education providers supported by the US Department of Education. The GEM ontology is published as a set of attribute names shown in Figure 5, most of which have a set of permitted values.

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

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

Figure 5 The GEM ontology

If we look at Figure 5 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* becomes relevant as a list of subtypes of *Resource*, giving a primitive type system with one level. The *Subject* property is organised into a type-subtype system of two levels (eg *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 6. 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.

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 6 Selection of GEM Resource Types

Further and richer structure comes from the attribute *Relation Type*, a sample of the terms in which are in Figure 7. These relation types are all relationships between resources, which can be represented in the DOLCE and BWW ontologies using various forms. For example,

1. *hasBibliographicInfoIn* is a dependency relationship;
2. *isRevisionHistoryFor*connects a whole to a part, which is a BWW *History of a Thing*;
3. *isCriticalReviewOf* can be represented as a causal relation (the resource causes the review resource);
4. *isContentRatingFor*can probably be seen as a DOLCE *Property*.

(Dublin Core[14] is a standard of metadata for web data items, consisting of 10 properties including *Name*, *Language*, *Version* and *DataType*.)

[14] http://dublincore.org/

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 7 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.

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:

1. the integers support equality together with arithmetical operations including division with remainder and binary relational predicates;
2. reals, in their computer incarnation as floating point numbers, support unqualified division (except by 0) but not equality;
3. sets support union, intersection, difference, membership test, add and delete member, and so on;
4. the subtype relation supports navigation between subtype and supertype, including a transitive closure operation;
5. the part-whole relationship supports the mereological operations and predicates[15], including mereological sum and transitive closure of parthood.

[15] See Mereology in Explanations

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 standardises 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 langauge for an information source as well as a software library.

Some of the functions and operations stemming from the ontologies presented include:

1. Every perdurant/ occurrence has participating endurants
2. Every event has an associated time
3. An occurrence persists through a record, which is an object
4. Every endurant has an associated history, which is a sequence of objects which are records of events
5. 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).
6. 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.

Content-neutral interoperation

Ontology-builders sometimes lose sight of the fact that a very large amount of interoperation among autonomous sites involves the exchange of information among collaborators – say a group of physicists working on problems in nuclear fusion or a group of ontology researchers working on issues in the design of ontologies. These interoperations involve the sharing or exchange of structured information but involve no a priori commitment to its content. Two researchers share views of their respective computer screens, or copies of a formatted work-in-progress documents. This sharing requires both sides to agree on the form of the shared information, but leaves the interpretation of the content to the humans. The humans might disagree on how to interpret the content, or on judgments of the likelihood of its truth or usefulness. The information systems interoperation is neutral with respect to content.

In these situations, a formal ontology can be useful since it provides rich structures for representing form, but is content-neutral. The markup structure of exchanged documents can make use of the whole-part relationship, for example, or the record of exchanges can make use of the distinction between perdurants and endurants by keeping track of the change events and the parts of the documents changed in those events.

People in these situations often also share what amounts to data, which does have an agreed ontology. The physicists share the characteristics of the fusion reactors on which their experiments are run as well as the results of the experiments. Biologists working on the mouse genome share the DNA data and data on the development of the mouse embryo. Ontology researchers share the ontologies they are discussing. It should be an advantage if the formal structures used to organize the data were the same as the formal structures used to share the screens and documents, because it would simplify the access, representation and query software.

Readings

For the definitive statement of the problem of interoperating information systems

Sheth, A.P. and Larsen, J. (1990) Federated database systems for managing distributed, heterogeneous and autonomous databases. *ACM Computing Surveys: Special Issue on Heterogeneous Databases* 22, 3 183-236.

For an analysis of why interoperating information systems is a difficult problem

Colomb, R. M. (1997) Impact of semantic heterogeneity on federating databases. *The Computer Journal* 40, 5, 235 -244.

For the theory of institutional facts

Searle, J. R. (1995) *The Construction of Social Reality* The Free Press, New York

For a widely-quoted definition of ontology

Gruber, T.R. (1993) *Toward Principles for the Design of Ontologies Used for Knowledge Sharing* Technical Report KSL 93-04 Knowledge Systems Laboratory Stanford University

For the Bunge-Wand-Weber (BWW) ontology

Weber, R. (1997) *Ontological Foundations of Information Systems* Coopers & Lybrand Accounting Research Methodology. Monograph No. 4. Melbourne.

For the OntoClean method

Guarino, N. and Welty, C. (2002). "Evaluating Ontological Decisions with OntoClean" *Communications of the ACM*, 45(2): 61-65.

Welty, C. and Guarino, N. (2001) Supporting ontological analysis of taxonomic relationships. *Data and Knowledge Engineering* 39, 1, 51-74.

For the DOLCE ontology

Masolo, C., Borgo, S., Gangemi, A. Guarino, N. and Oltramari, A. (2002) WonderWeb deliverable D17 version 2.0 http://www.ladseb.pd.cnr.it/infor/Ontology/Papers/OntologyPapers.html

(There are a number of useful papers there on ontologies, the OntoClean method, and DOLCE.)

For the general theory and practice of classification systems

Colomb, R. M. (2002) *Information Spaces: the architecture of cyberspace* Springer, London

Explanations

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 organised 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.

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.
8. An object is *atomic* if it has no proper parts.
9. 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.

INDEX