Extending Ontology to Behaviour in Communities of Interoperating Information Systems Agents

Robert M. Colomb

Technical Report 21/02 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,

Abstract

This paper argues that a community of interoperating information systems agents requires an ontology not only of the objects in the world, the agents themselves and the message types available; but also of the complex behavioural protocols through which tasks are accomplished. The universe of these applications is described using formal upper ontologies and the material ontology of institutional fact, and a method of representing a subsumption lattice of behavioural protocols using the process algebra representation of finite state machines is adapted from the literature. This behavioural subtype structure is shown to be compatible with the other aspects of the ontology. The resulting system is applied to a community of agents.

Extending Ontology to Behaviour in Communities of Interoperating Information Systems Agents

Robert M. Colomb

School of Information Technology and Electrical Engineering
The University of Queensland
Queensland 4072 Australia
colomb@itee.uq.edu.au
Work performed while visiting LADSEB-CNR; Corso Stati Uniti, 4; Padova, Italia

Abstract

This paper argues that a community of interoperating information systems agents requires an ontology not only of the objects in the world, the agents themselves and the message types available; but also of the complex behavioural protocols through which tasks are accomplished. The universe of these applications is described using formal upper ontologies and the material ontology of institutional fact, and a method of representing a subsumption lattice of behavioural protocols using the process algebra representation of finite state machines is adapted from the literature. This behavioural subtype structure is shown to be compatible with the other aspects of the ontology. The resulting system is applied to a community of agents.

Introduction

A community of agents is a collection of programs which interact with each other in order to achieve their respective ends as determined by the policies programmed into the agents by their owners. The agents interoperate by means of shared events, which are generally implemented by exchanges of messages.

It is well established that a community of agents must be supported by an ontology describing the objects involved in the interaction. An ontology is a standardized vocabulary and semantics. The agents must commit to the ontology: that is, they must agree on the terms to be used and what they mean [4].

The ontology includes the universe of objects which the messages exchanged by the agents are about (for example the SNOMED system for medical records, or the claimed universal ontology Cyc). It includes the agents themselves (organized in systems like Yellow Pages directories). It includes also the types of messages that can be exchanged and their content (for example the Z39.50 protocol for information exchange or the various Electronic Document Interchange (EDI) standards). Finally, and this is the point of this paper, it must also include the sequences of messages that must be exchanged in order to achieve particular tasks. Ideally, all of these aspects of the ontology should be coherent both intellectually for human understanding and representationally, for use by the agent programs.

Intellectual and representational coherence can be achieved partly by the use of a formal upper ontology like DOLCE [5] or the Bunge-Wand-Weber (BWW) system [11], and partly by very general material ontological distinctions like the distinction between brute fact and institutional fact given by Searle [6]. The material ontological stance of the present work is that almost all the content of most information systems consist of institutional facts, and that almost all the shared events of agent communities consist of speech acts which create institutional facts. The main formal ontological stance is the distinction between endurants and perdurants, based on the formal mereological primitive *part-of*.

In the sequel we first explain the ontological framework used, then focus on events and event sequences using the formal concept of finite state machine represented in process algebras. We then apply this framework to interoperation in agent communities, then show how the behavioural dimension of agent interoperation can be fitted in to a subtype structure, first at the level of two interoperating agents then at the level of an entire agent community.

Ontological Framework

Endurants and Perdurants

The distinction between endurants and perdurants is central to the DOLCE formal ontology [5] and also (under different names) to the BWW system [11].

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

Perdurants are entities that *happen* in time. A perdurant can happen at a point in time, or can have temporal parts. There are two subclasses of perdurant: event and stative. An *event* is definite. Even if it is complex, it is capable of completion. If it has temporal parts, none of them are of the same kind as the whole. A *stative* is indefinite. It can have temporal parts which are the same kind as the whole. We might think of an event as happening, while a stative is going on. See [7] section 4.1, where a perdurant is called an occurrent.

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

Exmples of statives include raining, sitting, being alive, being dead, being ready, running, working, cleaning (indefinitely – cleaning the kitchen floor is an event, since its temporal parts are things like cleaning in front of the stove, cleaning under the table, and so on, which are not cleaning the kitchen floor).

The BWW system includes events but not statives.

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

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

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

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

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

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

This view, expanded from [5] and [11] is generally consistent with [7] and [8].

Institutional facts

In the present work, the concern is not the ontology or metaphysics of the whole world, but only the specific aspect of the world needed to understand interoperating information systems as one finds in electronic commerce and the semantic web. Information systems of this sort are concerned almost exclusively with the special kinds of things called by John Searle [6] institutional facts and speech acts.

Searle recognizes two kinds of fact, *brute fact*, which is independent of human society, and *institutional fact*, which depends on human society for its existence. An institutional fact is a brute fact which has a social significance. Searle encapsulates the relationship as “brute fact X counts as institutional fact Y in context C”. An institutional fact is created by a *speech act*, and is sustained in existence by records of the speech act having been performed. An institutional fact is an endurant, as are the records through which it continues to exist.

A speech act is an event. It is something done which has a social significance in an institutional context. We can adapt Searle’s formula: “human activity X counts as speech act Y in context C”. Giving someone a name is a speech act. The activity is filling out a form and lodging it at a government registry of births. The context includes the person signing the form being a parent, the office the correct office of the proper government, and so on. Making a purchase is a conflation of two speech acts, buying to the purchaser and selling to the vendor. The activity is complex, for example lodging an order at the appropriate desk, handing over the goods, checking that the goods meet specifications, handing over a cheque, handing the cheque in to a bank, and the bank’s eventual transfer of funds from the purchaser’s account to the vendor’s account. The context includes the vendor having title to the goods, the paper being handled by properly designated staff, the purchaser having in fact an account at their bank with sufficient funds, and so on.

The institutional facts involved are in the first case the person having their name and in the second the purchase of the goods. The institutional facts are sustained in existence in the first case by the memories of the people who know and interact with the person, and also by the record of the birth certificate maintained by the government office. In the second case, the purchase is sustained in existence by the purchaser having undisputed title to the goods (possession and the memories of relevant people), backed up by copies of the invoices and payment records maintained by the various parties.

Institutional facts are special partly because they are created by discrete deliberate human events. They are also radically unchanging. My body has undergone many changes in the many years since I was given my name, but my name remains exactly the same. My car has undergone many changes since I purchased it, but the fact of the purchase remains exactly the same. The story of endurants and perdurants told in the previous section applies in a very straightforward way to institutional facts and speech acts.

Interoperating agents in information systems

We want to apply our ontological concepts to worlds consisting of collections of agents interoperating among information systems. It makes sense to begin with the ontology of this world.

At the outermost layer, one of these worlds consists entirely of agents. An agent is a computer program, either autonomous or functioning as an interface between a human and the other agents. An agent is an endurant, in much the same sort of way that a person is.

Agents interact with each other solely by interchange of discrete messages. These messages can be of two sorts: performative or informative. An *informative* message asks a question or gives an answer. For example the exchange:

1. Tell me the price and availability of product 1234
2. Sorry, we do not supply product 1234

consists of two informatives. The exchange

3. I am placing an order for 10 units of product 1234 on 17 October, 2002
4. Your order for 10 units of product 1234 ordered 17 October, 2002 has been shipped on 18 October, 2002, and your account has been debited.

Consists of two performatives. A *performative* is a speech act which establishes an institutional fact. Message 3 establishes the institutional fact of an order having been opened, which authorizes the vendor to ship and leads him to expect payment. Message 4 completes the purchase, closing the order, creating two new institutional facts of a purchase made and an order filled while destroying the institutional fact of an order being open.

The act of sending (and receiving) a message is a perdurant, specifically an event, since it happens at a given time. On the other hand, the message itself is an endurant. Once composed and sent, it can be stored by both the composer and sender agent, perhaps to be used as a record of the performative.

Many, perhaps most, messages in an agent world are complex – they have parts. The exchange 3 and 4 is a purchase with two temporal parts, order and fulfillment. In general, communication in an agent world is limited to tokens of a small number of types of messages. Each message type has a small and definite collection of types of parts. The types of messages and their parts is established by a convention among the organizations operating the agents. The convention is published as a standard, such as one of the electronic data interchange (EDI) standards, or the Z39.50 standard used in the library world.

At any given time, an agent is capable of receiving and responding to tokens of only some of the types of messages available in its world. The convention may require that a performative *making a purchase* can only be performed if the performative *establishing an account* has previously been completed. Furthermore, the messages which are parts of complex performatives must be exchanged in constrained sequences. A more complex purchase transaction involves seven types of message: request for quotation (*RFQ*) issued by the purchaser, *Quote* by the supplier, *Purchase Order* by the purchaser, *Delivery Advice* by the supplier, *Delivery Acknowledgement* by the purchaser, *Invoice* by the supplier and finally *Payment* by the purchaser. This particular type of purchase transaction requires that the messages be sent in that prescribed order.

So the behavioural propensities of an agent change in discrete ways over time, remaining stable between changes. It makes sense to consider the behavioural propensities of an agent during the interval between changes as a stative, more specifically as a state, since every temporal part of a stable behaviour propensity is the same behavioural propensity.

It happens that a widely used method of designing agents is that of *finite state machines* (*FSM*). The method has a vocabulary to describe the behavioural propensities of a system which includes the terms state, event and transition. The FSM term “state” describes something which would be described by the DOLCE term “state”, and similarly for the term “event”. An FSM “transition” is an association of a state, an event, and a new state.

In FSM, the behavioural propensities of an agent are prescribed by a design expressed in one of a number of formally equivalent methods, one of which is an FSM diagram. An FSM diagram is an endurant, as is any plan or design. So the design view of an agent is an endurant, while the behaviour of the agent is a sequence of perdurants: states changed by events. Recall that the history of the agent is also a perdurant, but the record of that history is an endurant.

So for agents describable by FSMs, there is a formal isomorphism between its behaviour (perdurants) and structure (endurants). The design constrains the behaviour, whose history is formally equivalent to the record of the history. An agent, like any endurant, is present in the world via qualities which inhere in it. One of the qualities inhering in an agent is its design and another the record of its history. So an agent’s behaviour is completely characterized by its qualities. Note that this characterization tells us that an agent describable by an FSM is very limited in comparison with a general agent, in particular in comparison with an unconstrained human.

Complex speech acts

Our concern in the present work is with interoperating agents. These agents cooperate in the performance of speech acts which establish institutional facts. A single speech act will generally involve behaviour from more than one agent. For this reason the context of speech acts generally includes the concept of *role*. An agent playing a particular role in a speech act will have a certain repertoire of behaviour, which will generally differ from the repertoire available to agents playing other roles.

Speech acts can be complex in more than one way. Besides the possibility that several behaviours may need to be coordinate to perform a speech act, a speech act can have temporal parts – subordinate speech acts which together make up the whole complex act. Each of these temporal parts may itself involve coordinated behaviour.

For example, the complex speech act *purchase transaction* described above, which involves the exchange of seven messages beginning with *RFQ* and ending with *Payment*, has two roles, a *Purchaser* and a

Supplier. It can be convenient to analyse the speech act into several temporal parts, each of which is a speech act:

- *Establish product and price*: Purchaser issues a *RFQ*, Supplier issues a *Quotation*
- *Place order*: Purchaser issues a *Purchase Order*
- *Deliver*: Supplier makes a delivery and issues a *Delivery Advice*, Purchaser issues a *Delivery Acknowledgement*.
- *Payment*: Supplier issues an *Invoice*, Purchaser issues a *Payment*.

Just as the behaviour of the agents can be described as a FSM, so also can the articulation into temporal parts of a speech act. Each speech act part provides the context in which are valid other speech act parts of particular types. The part *Place order* is valid only if *Establish product and price* has been previously performed; the part *Deliver* is valid only if *Place order* has been previously performed; the part *Payment* is valid only if the part *Deliver* has been previously performed.

What we will call *behaviour* in the sequel is just this sort of articulation of complex speech acts. A single event therefore is a speech act which involves possibly several roles and possibly the exchange of several messages among the agents playing those roles. There may or may not be constraints on the sequence of messages.

Ontology of Behaviour

It is well known that endurants can often be organized into subtype lattices based on constraints on their qualities. In general, a subtype has qualities which are strictly more constrained than any supertype. Therefore, if we are able to establish a system of constraints on the design or record of the history of an agent, we can include these qualities in the subtype structure. Since the record of the history of an agent is completely characterized by its design, we can limit our search for a system of constraints to the design only.

Process algebra representation of FSM

One of the equivalent ways of representing the design of an FSM is as a process algebra. The *process algebra* represents a FSM by a description of allowed sequences of events using what are called regular expressions. A *regular expression* is composed from an alphabet of symbols representing the possible event types recognized in any state, using several operators. If *a* and *b* are events, then the *sequence* operator "." allows the designation of *a* followed by *b* as *a.b*. That is to say that if the FSM recognizes event *a*, it always transitions to a state in which it will recognize event *b*. The *selection* operator '+' allows the designation of a state in which either *a* or *b* are recognized by *a+b*. A FSM can be in a state in which a particular event type is recognized which results in the FSM remaining in the same state. In this situation, the event type can recur and again be recognized, and the situation is repeated indefinitely. The process algebra has an *iteration* operator '*' to express this situation. If *a* is an event, then *a** represents that event recurring indefinitely. Finally, the process algebra has a *perform in parallel* operator '||', with *a*||*b* expressing that events *a* and *b* must both occur, but the order is not specified (|| is derived: a||b = (a.b + b.a)).

Besides the sequence, selection and iteration operators, the process algebra has two constants: "do nothing" represented by "1" and "deadlock" represented by "0". *Deadlock* represents a state in which no event is recognized – the FSM is frozen. The alphabet, constants and operators can be combined in regular expressions using parentheses for grouping. The process algebra is so called because some regular expressions are equivalent to others, according to the following laws. If e, e' and e" are events in the alphabet, then

1. e + 0 = e
2. e + e = e
3. e + e’ = e’ + e
4. e + (e’ + e”) = (e + e’) + e”
5. 1.e = e.1 = e
6. e.0 = 0.e = 0
7. e.(e’.e”) = (e.e’).e”
8. e.(e’ + e”) = e.e’ + e.e”
9. (e’ + e”).e = e’.e + e”.e

The alphabet of events has three distinguished kinds: *creators*, *destructors* and *modifiers*. Creators bring an instance of an object into existence, while an object recognizing a destructor destroys itself. Modifiers are events which are recognized by an already existing object, and which do not destroy it. An alphabet for an object must have at least one creator and at least one destructor. The alphabet of event types for an object *A* is designated *ev(A).*

A regular expression can be rewritten using the process algebra into a collection of alternatives each of which is a sequence. Each of these alternatives represents a valid sequence of events, called a *scenario*. This collection of scenarios is called the *regular language* associated with the regular expression. We designate the regular expression associated with object A as *exp(A)* and the regular language derived from it as *lang(A).*

The regular expression is the design for the behaviour of the FSM, while the regular language is the collection of all possible (records of) histories.

Behavioural subtypes

Snoeck and Dedene [9] recognize a *behavioural subtype* relationship between two objects G and S (S is a subtype of G). Their characterization is based on the notion that a subtype is less constrained than its supertype, so their results are not directly applicable to the present work. However, if we adapt their system to the notion of a subtype as being more constrained than its supertypes, then we get the formulation that S is a subtype of G if

ev(S) is a subset of ev(G). (1)

every sequence of events valid for S is also valid for G. (G may recognize additional events, but must recognize at least every sequence of events its subtype does.) (2)

To formally represent point 2, we first need some definitions.

If A and B are two objects, then we can project the behaviour of B onto behaviour of A using an operator “|”.

- lang(B|A) $=_{df}$ scenarios in lang(B) removing all events in (ev(B) - ev(A)).
- exp(B|A) $=_{df}$ expression generating lang(B|A).

In other words, we go through the scenarios for B and remove all events not in the alphabet of A. Exp(B|A) is the regular expression generating the resulting language.

So point 2 becomes

lang(G|S) is a superset of lang(S) (2’)

This generates a partial order > on the regular expressions. We define

- exp(B) > exp(A) $=_{df}$ lang(B) is a superset of lang(A),

so (2’) becomes

exp(G|S) > exp(S) (2”)

We overload the partial order designator, so that we also write G > S if G is a supertype of S. Notice that a consequence of these definitions is that a regular expression containing an iteration operator subsumes an otherwise identical expression lacking the iteration operator. The expression <e*> > <e>.

Example

The following example is adapted from [9].

Imagine a library where different kinds of items can be consulted: single issues of journals, volumes of journals, books, CD-roms, …, and so on. All available items can be searched for by means of an on-line catalogue. Issues of journals can not be lent out; books and volumes of journals can. Only loans of books can be renewed. To prevent loss, CD-roms are kept in a separate place and must be lent out at the loan desk. They must not leave the library. Equipment is provided to view CD-roms and to print informatioin. Printing is charged when the CD-rom is returned.

A possible library model could contain the following object type definitions:

Version 1

ITEM = <{create, classify, borrow, return, declassify, remove, renew, lose, print},
create.classify.(borrow+return+renew+lose+print)*.declassify.remove>

BOOK = <{create, classify, borrow, return, declassify, remove, renew, lose},
create.classify.(borrow.renew*.return)*.
(borrow. renew*.lose+1).declassify.remove>

VOLUME = <{create, classify, borrow, return, declassify, remove, lose},
create.classify.(borrow.return)*.(borrow.lose+1).declassify.remove>

ISSUE = <{create, classify, declassify, remove, lose}, create.classify.(lose+1).declassify.remove>

CD-ROM = <{create, classify, borrow, return, declassify, remove, print, lose},
create.classify.(borrow.print*.return)*.(borrow.print*.lose+1).declassify.remove>

BOOK, ISSUE, VOLUME, CD-ROM are specialization types of ITEM:

ITEM < BOOK, ITEM < ISSUE, ITEM < VOLUME, ITEM < CD-ROM

Note that the behaviour of the supertype ITEM is very weakly constrained. It would be possible to refine the behaviour of the supertype to the regular expression created by taking as alternatives the four subtype regular expressions. This would, however, make it difficult to introduce additional subtypes.

Application of OntoClean concepts

We have so far developed a definition for behavioural subtype which is consistent with the standard concept of subtype in information systems, where the subtype is more constrained than the supertype. In the information systems case, the constraints apply to attributes or relationships which are optional in the supertype; while in the behavioural case, the constraints apply to allowed scenarios, or sequences of events recognized by the supertype.

In the standard information systems formulation, systems of subtypes are subject only to the constraint that each instance of the top supertype, including instances of all subtypes, must be capable of being identified by a combination of attributes of the top supertype. These attributes are therefore mandatory. There are also generally logical expressions on the values of attributes or relationships with which an instance of a supertype can be identified as an instance of a given subtype (defined types in description logic terms). Our library example in its data model would have attributes through which an item could be identified as a book, volume, issue or CD-rom.

The OntoClean system [12] features a number of meta-properties of attributes and relationships which make it easier to talk about the subsumption hierarchy, and which provide a discipline for the use of primitive subtypes in description logic systems. These features can apply to behavioural subtypes, and thereby increase their usefulness.

We can think of the alphabet, regular expression and language of an object as sorts of properties. One of the OntoClean metaproperties is essentiality. An *essential* property is one which has the same value for all instances of a type. We can partition the alphabet of a type into essential and non-essential event types. If ev(T) is the alphabet of type T, we can call them ess(ev(T)) and non-ess(ev(T)), respectively. This would naturally be interpreted as a requirement that all subtypes of T have ess(ev(T)) as subsets of their alphabet. We could therefore simplify the representation of the type system by including in the declaration of the alphabet of a subtype only those event types which are non-essential in the supertype. The subtype would inherit the essential event types.

Since behaviour consists of combinations of event types, both essential and non-essential, it is not sufficient to simply propagate essentiality from the alphabet to the language. We need also to designate scenarios as essential or non-essential. An essential scenario would of course consist only of essential event types. The stipulation would therefore be that every scenario in the language of a type when projected onto the essential alphabet would be an essential scenario.

Formally, if we have type G, we designate ess-lang(G) as a set of scenarios over ess(ev(G)), then

E1. lang(G|ess(ev(G))) must be a superset of ess-lang(G).

By the definition of essential, if S is a subtype of G, then

E2. lang(S|ess(ev(G))) must be a superset of ess-lang(G)

and

E3. exp(S|ess(ev(G))) is the regular expression generating lang(S|ess(ev(G)))

A specialization of essential property is rigidity. A *rigid* property is one which is not only essential but also serves to identify an individual as an instance of its type. It is conceivable that one might want to identify a type at least partly by the events or sequence of events it recognizes.

We now re-cast the library example in this enhanced system. We will designate essential events in the alphabet by making them **bold**. Similarly, we will designate the essential aspects of an expression where possible by **bolding**. We have stipulated that create, classify, declassify, return and lose are events that must be able to happen to any event, but that not every scenario must include losing the item. Essential event types have been removed from all the subtypes. The expression for all the subtypes includes the essential expression for ITEM.

Version 2

ITEM = <{**create**, **classify**, borrow, return, **declassify**, **remove**, renew, **lose**, print},
create.**classify**.(borrow+return+renew+(**1+lose)**+print)*.**declassify**.**remove**>

BOOK = <{borrow, return, renew}, create.classify.(borrow.renew*.return)*.
(borrow. renew*.lose+1).declassify.remove>

VOLUME = <{borrow, return}, create.classify.(borrow.return)*.(borrow.lose+1).declassify.remove>

ISSUE = <{}, create.classify.(lose+1).declassify.remove>

CD-ROM = <{borrow, return, print},
create.classify.(borrow.print*.return)*.(borrow.print*.lose+1).declassify.remove>

BOOK, ISSUE, VOLUME, CD-ROM are specialization types of ITEM:

ITEM < BOOK, ITEM < ISSUE, ITEM < VOLUME, ITEM < CD-ROM

Versions 1 and 2 of the example have served to illustrate the syntax and semantics of the behavioural subtype, but not the power of the concept. The subtype system is flat. We can follow Snoeck and Dedene and improve the structure, removing redundancy by creating an intermediate type LOAN-ITEM which has essential behaviour the same as VOLUME. We add to the syntax that the expression of a type indicates the insertion of its introduced event types by *italics*. The library example becomes

Version 3

ITEM < LOAN-ITEM, ITEM < ISSUE,

LOAN-ITEM < CD-ROM, LOAN-ITEM < VOLUME, LOAN-ITEM < BOOK

ITEM = <{**create**, **classify**, borrow, return, **declassify**, **remove**, renew, **lose**, print},
create.**classify**.(borrow+return+renew+(**1+lose)**+print)*.**declassify**.**remove**>

LOAN-ITEM = <{**borrow**, **return**},
create.classify.(*borrow.return*)*.(*borrow.*lose+1).declassify.remove>

BOOK = <{renew}, create.classify.(borrow.*renew**.return)*.
(borrow.*renew**.lose+1).declassify.remove>

VOLUME = <{},
create.classify.(borrow.return)*.(borrow.lose+1).declassify.remove>
CD-ROM = <{print},
create.classify.(borrow.*print**.return)*.
(borrow.*print**.lose+1).declassify.remove>

ISSUE = <{}, create.classify.(lose+1).declassify.remove>

Notice that although CD-ROM recognizes and responds to the event types inherited from LOAN-ITEM, it uses different methods to do so. A CD-rom can not be removed from the library. The system allows subtypes to override methods inherited from supertypes, so long as their behaviour sequence is formally identical.

An agent world

We would like to have a characterization of a world of agents, that is to say the collection of agents which interact within a particular domain, be it an electronic commerce exchange, a group of museums using Z39.50, or a fragment of the semantic web organized around some group of topics. There are three elements to the problem: the agents themselves, the messages they exchange (their behaviour), and the universe to which they refer in their messages.

The universe to which they refer is generally organized according to an ontology of endurants, expressed as a conceptual model. In previous sections of the present work, we have shown how the agents' behaviour can be organized according to an ontology of perdurants derived from an ontology of plans which are endurants. The agents themselves are endurants, so can be organized according to an ontology derived both from their universe and from their behaviour. This section shows how an integrated view can be constructed.

The key idea is the recognition that the universe consists of institutional facts. The agents are sending messages which are either queries and responses concerning institutional facts (informatives) or are speech acts which alter the universe of institutional facts (performatives). Institutional facts are complex structures, organized in multiple layers of contexts. The organization of contexts is what gives coherence to the agent world.

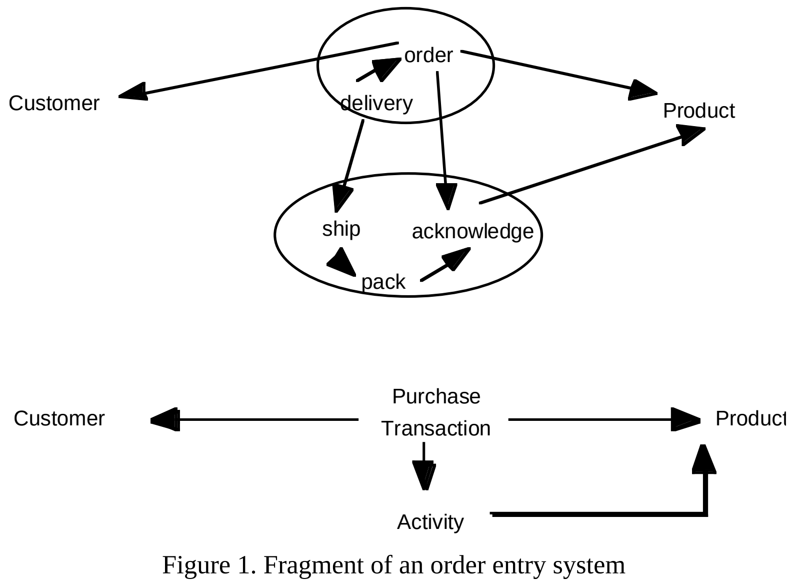


Figure 1. Fragment of an order entry system

Figure 1 shows a fragment of an order entry system, presented at two levels of abstraction, expressed as an entity-relationship (ER) model. (Entities are shown by names, relationships by arrows. Relationships are un-named unless necessary for disambiguation. An end of an arrow with an arrowhead has cardinality *one*, while an end without an arrowhead has cardinality *many*. The relationships are all assumed to be mandatory on the many side. The notation is intended to clearly show the functional dependencies. An arrow shows a functional relationship whose domain is the entity at the plain end and codomain the entity at the end with the arrowhead. By assumption, all functional relationships are total. We will call an instance of the domain of a functional relationship a *source* instance and the corresponding instance of the codomain a *target* instance.)

The more abstract representation in the lower part of the figure has four entities, each of which are types of institutional fact. An instance of *Customer* is a record of the speech act of establishment of a business relationship with another agent, whose details are recorded in the institutional fact as qualities, represented in the ER model as attribute values. An instance of *Product* is a record of the speech act of offering for sale a type of product. The speech act includes a number of elements like description and price, represented in the ER model as attributes. An instance of *Purchase Transaction* is a record of a customer's having successfully purchased a quantity of a particular product on a particular date. An instance of *Activity* is a record of what the organization does when it accepts and fills an order (hand over the goods, accept payment, and so on).

A dynamic view

Following [2] or [10], we can interpret an instance of a total functional relationship as saying that the target instance must exist in the codomain before or at least at the same time as the source instance in the domain. In Figure 1, this says that in order for an activity to be completed, the corresponding product must have been already offered for sale, and that in order for a purchase transaction to be completed, not only must the product have been already offered for sale, but the corresponding customer relationship must have been established and the corresponding activity completed. From a static point of view, the target institutional facts must already exist. They are parts of the context of the source institutional facts. From a dynamic point of view, the speech acts creating the target institutional facts must have been performed before the speech acts creating the source institutional facts.

Using the process algebra representation, we have an alphabet of four events

est-cust:	establish a customer relationship with someone
off-prod:	offer a product for sale
act:	complete the activity implementing a purchase
purch:	complete a purchase transaction

and the behaviour is represented by the regular expression

<(est-cust||off-prod).(act.purch)*> (3)

The key point of this example is that the alphabet of the process is derived from the need to create instances of the four entities in the model, and regular expression is derived from the functional dependencies in the model. The dynamic aspect is implied by the static aspect.

Note that the converse is also true. The static aspect is implied by the dynamic. The entities are records of the occurrence of the events in the alphabet, while the functional dependences are guaranteed by the sequence restrictions.

In Figure 1, the upper part is a refinement of the lower part. An instance of the institutional fact type *Activity* has parts of type *Ship*, *Pack* and *Acknowledge*, while an instance of the institutional fact type has parts of type *Order* and *Delivery*. All the parts are also institutional facts, with their corresponding speech acts.

Refinement of the speech acts gives the regular expression

<(est-cust||off-prod).(acknowledge.(order||(pack.ship)).deliver)*> (4)

The rules of the refinement are given in [1]. In the static view, they require that each part of *Purchase Transaction* be functionally dependent on a part of *Activity* in such a way that the transitive dependencies are preserved. In the dynamic view they require that each part of *purch* follow a part of *act*. The details are beyond the scope of the present paper, and in any case are not essential for the present purposes, except to indicate that the refinement in the process view appears more complex than it does in the static view. This is of course an artifact of the difference between a one-dimensional and a two-dimensional notation.

We have illustrated the argument of Snoeck and Dedene [10] that the dynamic aspect of the system is equivalent to the static aspect, with examples in (3) and (4). In fact, there are subtle problems with both examples, one of which is that in example (4) the event *acknowledge* precedes the event *order*. This is certainly counterintuitive, since the order originates with the agent playing the customer role, and is logically the start of the whole process. Our problem with (4) raises the point that the populations of databases are generally updated in transactions, in which possibly several changes are made at the same time. In fact, the population of a database implementing Figure 1 would be updated in a transaction in which instances were added simultaneously to both *order* an *acknowledgement*. The static view is to this extent less refined than the dynamic view.

The issue becomes clearer when we step back to a more ontologicial view. In Figure 1, the institutional fact recorded in an instance of *order* represents the completion of a speech act of the supplier accepting the order. That speech act has parts, namely first the customer requests to place an order, then the supplier acknowledges the customer’s request, thereby accepting the order. The record of the speech act *accept an order* includes the records of the speech acts *request an order* and *acknowledge the request*. The transaction records only the completion of *accept an order*, while the dynamic view must consider the sequence of parts.

The regular expression (4) is better as

<(est-cust||off-prod).(order.acknowledge.(pack.ship.deliver))*> (4’)

A second such problem is present in the model of Figure 1, namely the relationships *deliver*, *pack* and *ship*. The speech act *deliver an order* includes the speech acts *pack* and *ship*. Furthermore, a reasonable enterprise interpretation of the model in Figure 1 would have the process initiated by the issuance of a delivery order, which would trigger the relevant part of the organization to schedule a shipping order, which in turn would trigger the issuance of a packing order. On completion of the packing, the shipping could proceed to completion, thereby allowing the delivery to complete. In the same way as a completed *order* institutional fact is recorded in instances of both the entities *order* and *acknowledgement*, a completed *delivery* institutional fact is recorded in instances of all three of *delivery*, *ship* and *pack*. If the database is recording completed institutional facts, then the three *delivery*, *ship* and *pack* would be updated simultaneously in a transaction. Refining the dynamic aspect, regular expression (4’) becomes

<(est-cust||off-prod).(order.acknowledge.deliver.ship.pack)*> (4”)

Our regular expression now expresses the correct sequence of actions, but there is still a minor problem in that the packaging has been lost that makes the complex *order* and *acknowledgement* distinct from *deliver*, *ship* and *pack*. For similar reasons, the sequence of *act* and *purch* in (3) must be reversed.

We can see that, contrary to the claim of [10], existence dependency does not tell the whole story. The conceptual model expresses the static structure of the data on completion of all the institutional facts represented in it, but does not adequately specify the sequence of speech acts creating complex institutional facts.

OntoClean clarifies the situation

The OntoClean system of meta-properties supports both existence dependency and parthood [3]. A complex institutional fact is essentially dependent on its parts. That is, existence of all its parts is essential for the existence of the whole. This is why the static view of Figure 1 shows the whole being dependent on its parts. The whole of an institutional fact is an endurant. But the creation of the institutional fact is an event, and the event has temporal parts. Occurrence of the first temporal part begins the event, which is not completed until the occurrence of the last temporal part. Because the whole is essentially dependent on its parts, the static view which shows only wholes must be updated with the whole and its parts simultaneously, in a transaction.

Returning to the ER model of Figure 1, the essential part analysis shows that the representation system used in ER uses the same notation to designate ontologically different relationships between atomic entities. This kind of situation is called *ontological overloading* by Weber [11]. In the bottom, more abstract, model the relationships whose targets are *Customer* and *Product* are all existence dependencies, while the relationship whose source is *Purchase Transaction* and whose target is *Activity* expresses an essential part relationship. In the refinement, *Purchase Transaction* has two parts, *Order* and *Delivery*. In their turn, *Order* has an essential part *Acknowledgement* and *Delivery* has an essential part *Ship*, which has an essential part *Pack*. The relationships from *Delivery* to *Order*, and from *Pack* to *Acknowledge* plus the derived relationships from *Ship* and *Deliver* to *Acknowledge* are all existence dependencies.

Since the ER model represents existence dependency and essential parthood in the same way, additional specification must be added to the design to be able to specify behaviour. One way to do this is by annotating the ER diagram, as in Figure 2. The essential part relationship has a thicker arrow than the existence dependency.

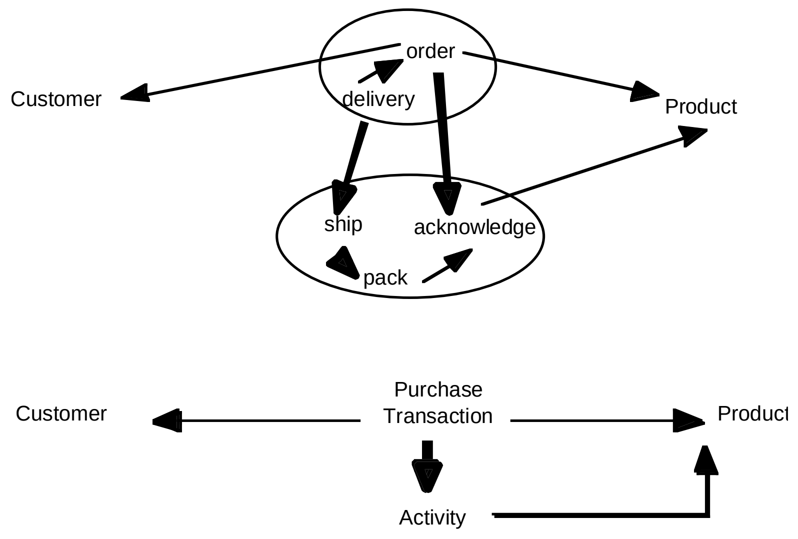


Figure 2 Figure 1 with essential part relationship noted

The behaviour can now be generated against the arrows for existence dependency and with the arrows for essential parthood. The resulting regular expression is

<(est-cust||off-prod).((order.acknowledge).(deliver.ship.pack))*> (5)

The speech acts establishing a customer relationship and offering a product for sale provide the context for purchase transactions. The temporal part *order* establishes the context for its completion in *acknowledge* and then for temporal the part *deliver*, which in its turn establishes the context for its completion in *ship*, which establishes the context for its completion in *pack*. When all of *purchase transaction*'s temporal parts have occurred, then the institutional fact of having made a purchase is created.

Interoperation

The agent through which an information system interoperates with others does not in general expose all its internal operations to its partners. It makes visible only sufficient data structure to support the speech acts involving other parties, and only those speech acts are known about by its partners.

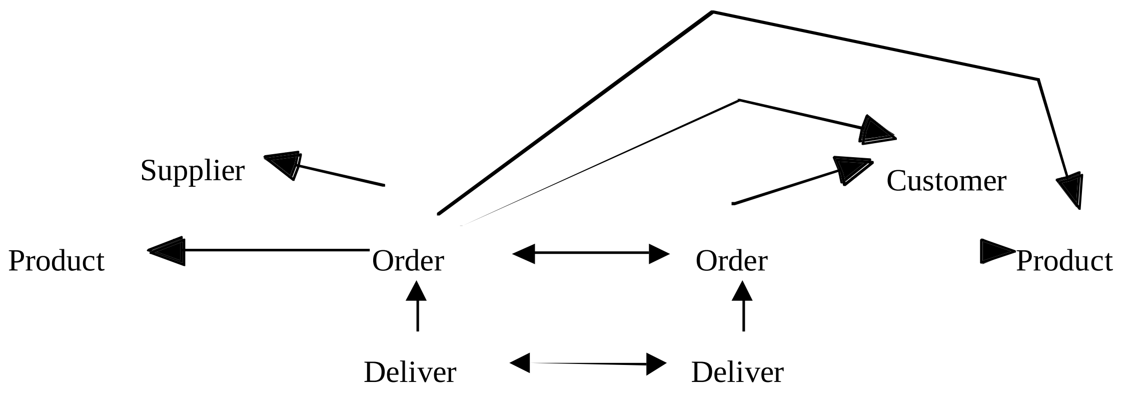


Figure 3: A simple e-commerce interaction

In Figure 3, we have the order entry system of Figure 1 on the right, and a counterpart purchasing system shown on the left. The only objects published by the order entry system are *Customer*, *Product*, *Order* and *Deliver*. The counterpart purchasing system publishes only *Supplier*, *Product*, *Order* and *Deliver*.

In order for any interoperation to take place, the two structures must be coordinated. The speech acts generally involve behaviour of more than one agent. In particular, the instance of *Customer* referring to the particular purchaser and the instance of *Supplier* referring to the particular supplier would have been created simultaneously in the same speech act in which the institutional fact of the customer-supplier relationship was established. Also, the instance of product involved in the sale must be in a one-to-one correspondence between the two. This correspondence would have been created previously be one or more informatives, probably initiated by the purchaser. The records of the institutional facts of the order having been placed and the delivery having been made are kept in both systems.

The behaviour of the interaction has the behaviour *Purch* with the alphabet

establish-customer-supplier-relationship (est-cust-supp)
offer-product (off-prod)
establish-correspondence-between-products (est-corr)
order
deliver

governed by the regular expression

<off-prod.(est-cust-supp||est-corr).(order.deliver)*> (6)

If we designate by *OE* the behaviour of the order entry system governed by the regular expression (4), and identify the speech act *est-cust* of *OE* with *est-cust-supp* of *Purch*, we can see that the projection of OE onto Purch has the regular expression

exp(OE|Purch) = <(off-prod||est-cust-supp).(order.deliver)*> (7)

In order to specify the interoperation in (6), we need to impose a restriction on the parallel execution of events *off-prod||est-cust-supp*, as well as add the informative *est-corr*, which is not visible in the order entry system. This shows that in general we need to refine the specification of individual systems when we specify their interoperation. In order for the behaviour of the order entry system to conform to regular expression (6) when it is projected onto the interoperation along with the data, regular expression (5) would have to be

< off-prod.(est-cust||est-corr).((order.acknowledge).(deliver.ship.pack))*> (8)

The refinement required is to recognize that offering a product for sale has precedence over establishing a customer relationship, making provision for the informative in the alphabet, and requiring that the informative precede the order.

Role of behavioural subtypes

Ontologies are used to organize the objects in a world of agents. The essential qualities of objects higher in the subsumption network can be used to simplify interactions concerning objects lower in the network. For example, if the agent world concerns the purchase of software, the products offered for sale may be organized into a hierarchy as shown in Figure 4. Essential qualities of *Product* include that the product type = software (a rigid quality) and license conditions common to all software purchases. There are two subtypes, that sold on CD with a delivery time common to all products and that sold by download, with the ftp site used for downloading as an essential quality.

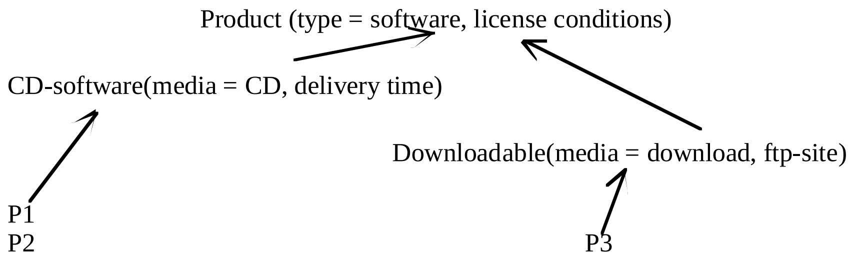

Product (type = software, license conditions)
CD-software(media = CD, delivery time)
Downloadable(media = download, ftp-site)
P1
P2
P3

Figure 4 A simple subsumption hierarchy

All of the terms used in Figure 4 are of course registered in the ontology, so that the supplier and potential purchasers must all commit to the ontology in order to be able to use the terms reliably. An agent intending to purchase software can make a series of queries (informatives) on the supplier’s product catalog to make a decision to buy say product *P3*. The product catalog is a system of endurants.

How does the purchasing agent actually accomplish the purchase? The actual purchase is an event. For purchasing downloadable software, the event is simple, just download. *Download* is a speech act in which the agent playing the purchaser role sends a message identifying the product to be downloaded and providing electronic funds transfer details, while the agent playing the supplier role executes the electronic funds transfer and the download itself.

For purchasing CD-software, the event has temporal parts: first order the software then take delivery. Not only is the latter more complex than the former, they use different methods. Clearly, the alphabet of elementary events must be in the ontology as a set of names (endurants referring to perdurants). So the supplier and intending purchasers can agree on what *order*, *deliver* and *download* mean and what data must or can be included in the messages implementing these elementary events. Not only that, we have established above that we can assign specifications of behaviour as qualities of endurants. In this way, the appropriate purchasing behaviour can be assigned as essential qualities at the appropriate level in the subsumption network, as in Figure 5. The behavioural qualities form a subsumption hierarchy, since they satisfy the requirements (1) for events and (2) for the behaviour.

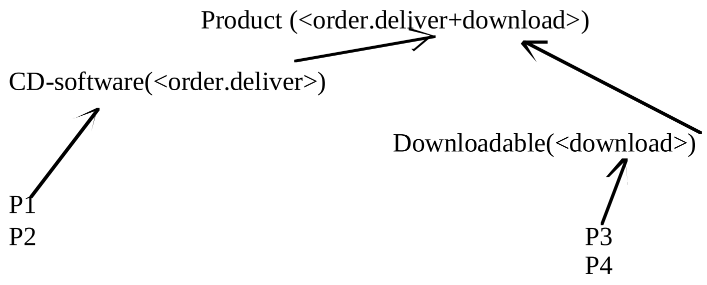

Product (<order.deliver+download>)
CD-software(<order.deliver>)
Downloadable(<download>)
P1
P2
P3
P4

Figure 5. Subsumption hierarchy with behavioural qualities

We have now that it is possible to associate with any object in an ontology supporting an information system attributes whose values specify how to interact with it. These attribute values represent qualities in a quality space which is aligned with the subsumption network in which the objects are organized.

Which agent performs which roles is specified only implicitly. The agent managing the product catalog and the order entry function will only attempt speech acts in which it can play the supplier role, while an agent intending to purchase will only attempt speech acts in which it can perform the purchaser role. Therefore the role names must be included in the ontology as well as the speech acts.

From dyad to community

The behavioural qualities and their subsumption structure of the previous section was developed in the context of a single interaction of two agents. In practice, a community will consist of many agents which interoperate over a long period. The agents will be programmed for complex behaviour such as

- keeping a manufacturing plant operating fully with minimum inventory of parts and materials at a minimum cost
- purchasing complex goods and services by tender
- responding to tenders for complex goods and services
- planning travel itineraries
- buying and selling goods at auction

Agents will be programmed to exhibit more or less intelligent behaviour, ideally to be able to adapt their strategies based on assessment of success or failure.

There is a wide variety of agent interaction patterns (*protocols*), even for the restricted activity of purchasing.

A minimal protocol is a simple purchase, where a customer first identifies a single product or service desired then executes the purchase. The customer and vendor do not explicitly maintain a relationship. In particular, payment is made by credit card or electronic funds transfer at the time of purchase. Examples include purchase of tickets on “no-frills” airlines, purchasing software by download, booking hotels, concert tickets, retail cash sales generally. An alphabet for a simple purchase is {offer-product, identify-product-price, purchase} with a regular expression

simple-purchase = <offer-product.(identify-product-price.purchase)*> (9)

We have already in Figure 5 introduced a contrasting protocol, where a physical product is involved which must be delivered:

deliver-purchase = <offer-product.(identify-product-price. purchase.deliver)*> (10)

A simpler protocol occurs in markets where the products are unique, such as art, used goods or real estate. There the product is offered only once, but still is delivered

unique-purchase = <offer-product.identify-product-price. purchase.deliver> (11)

A more complex protocol occurs where the purchase is for a number of different goods at one time, but still without establishing a business relationship. This protocol occurs in many retail applications and also in complex travel itineraries

shopping-cart = <offer-product*.(identify-product-price*.purchase)*> (12)

which can be without delivery (12) or with delivery

del-shopping-cart = <offer-product*.(identify-product-price*.purchase.deliver)*> (13)

Some sites make partial deliveries

part-del-shopping-cart = <offer-product*.(identify-product-price*.purchase.deliver*)*> (14)

There is a protocol where the vendor and purchaser establish a relationship. The example is built on (14), but a comparable paradigm could be built on any of the protocols (9) to (14)

rel-part-del-shopping-cart = <offer-product*.est-cust.(identify-product-price*.purchase.deliver*)*> (15)

Similarly, the purchase can be divided into the temporal parts order, deliver, bill and pay, where billing aggregates several orders (a normal and fairly simple business-to-business protocol):

bill-part-del-shopping-cart = <offer-product*.est-cust.(identify-product-price*.
order.deliver*)*.bill.pay> (16)

There are many protocols besides these shown, some of which are exceedingly complex. They can involve *requests for quotation*, *quotes* and *purchase orders* in place of the simple *identify-product-price* method of commiting to a purchase. Some industries have complex discount structures which depend on cumulative purchases over nominated periods of time. And so on.

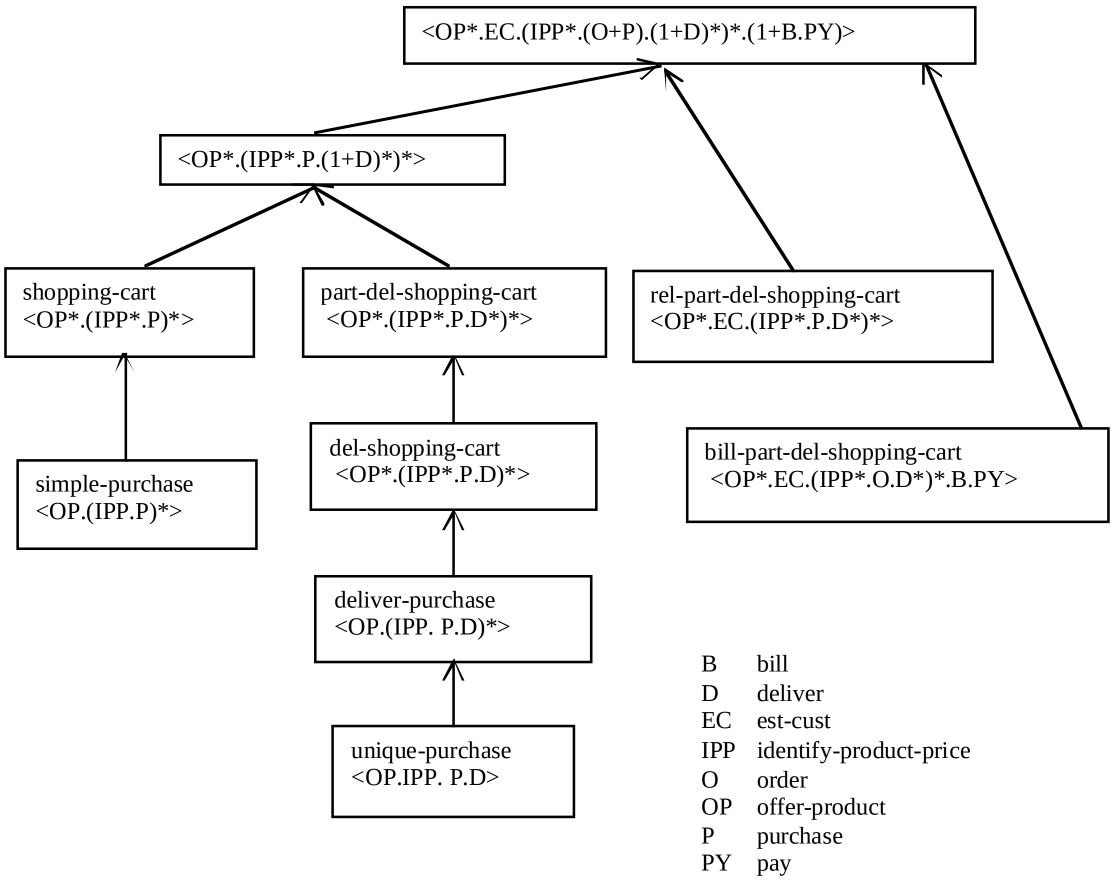
A community of agents operating through some sort of exchange functions much like a particular industry in the physical world, like liquor retailing, home renovations or domestic real estate. Even though there are many different protocols in commerce generally, a particular industry uses only a few.

Furthermore, each interaction strategy implemented in an agent must assume a particular protocol. In order for agents to interact, they must share protocols. It is therefore an advantage for the developers of the agents if the agent community supports a limited number of protocols.

Use of a limited number of protocols in a particular agent community therefore makes sense both from a business and a technical point of view.

An agent community needs an ontology describing the things the interagent interactions are about. We have previously established that it is possible to develop an ontology of interaction patterns. Therefore, the limited number of interaction patterns supported by a particular agent community can be included in its ontology. Figure 6 shows the interaction patterns (9) to (16) organized as a subsumption hierarchy. Note that the alphabet has been abbreviated for clarity of visualization of the whole, according to the legend shown.

With this structure, an agent participating in an exchange can advertise the interaction patterns it supports in addition to its product catalog. It only needs to advertise the most general pattern, since an agent with a subsumed pattern can interoperate successfully. Should different classes of products support different interaction patterns, the interaction pattern can be associated with the subsumption structure at the most general level relevant, as in Figure 5.


<OP*.EC.(IPP*.(O+P).(1+D)*)*.(1+B.PY)>
<OP*.(IPP*.P.(1+D)*)*>
shopping-cart <OP*.(IPP*.P)*>
part-del-shopping-cart <OP*.(IPP*.P.D*)*>
rel-part-del-shopping-cart <OP*.EC.(IPP*.P.D*)*>
simple-purchase <OP.(IPP.P)*>
del-shopping-cart <OP*.(IPP*.P.D)*>
bill-part-del-shopping-cart <OP*.EC.(IPP*.O.D*)*.B.PY>
deliver-purchase <OP.(IPP. P.D)*>
unique-purchase <OP.IPP. P.D>

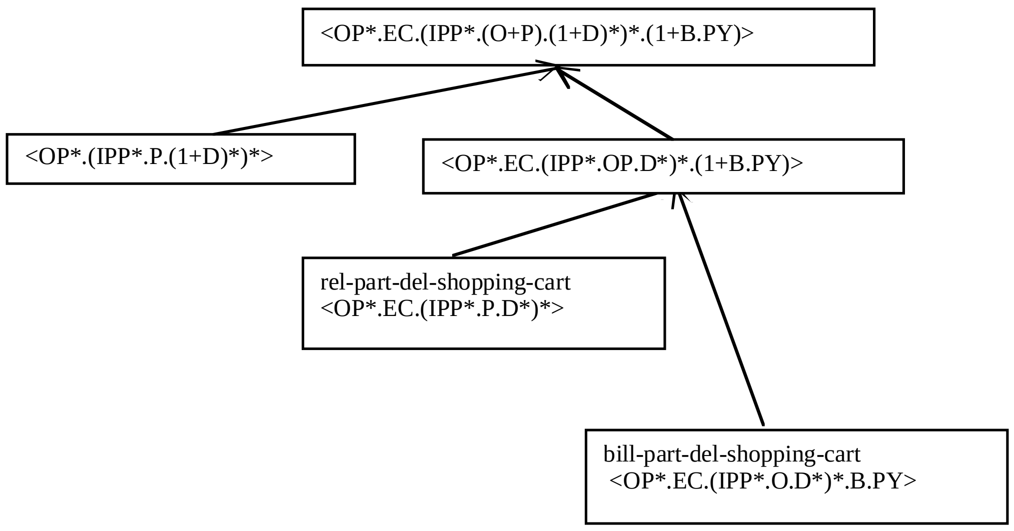
B	bill
D	deliver
EC	est-cust
IPP	identify-product-price
O	order
OP	offer-product
P	purchase
PY	pay

Figure 6 A subsumption hierarchy for commerce protocols

Interaction with other parts of the ontology

The behavioural subtype system proposed here is consistent with and interacts with the usual ontologies of endurants. We have seen already that the behavioural qualities form a quality space which can be the basis of defined types in the product catalog. The atomic events are exchanges of messages. The sending of a message is a perdurant, but the message itself is an endurant, since it is wholly present and its text can be made to persist. The messages can have a subtype structure, which is isomorphic to a subtype structure for atomic events.

For example, in Figure 6, the two rightmost protocols *rel-part-del-shopping-cart* and *bill-part-del-shopping-cart* have respectively the atomic events *purchase* (*P*) and *order* (*O*), which occur at homologous places in the protocols. The *purchase* event includes all the information in the *order* event, with the addition of message elements enabling funds transfer, for example credit card details. The funds transfer elements can be seen as a package of fields which are either present (in *purchase*) or absent (in *order*). We can construct a supertype where this complex is optional, called say *OP*. This would enable the two protocols to be organized into a subtype structure as in Figure 7, based on Figure 6 with the leftmost subtype system removed for clarity. The intermediate type on the right has the optional sequence B.PY and the message supertype OP. The leftmost leaf type has one message subtype and the optional behaviour absent, while the rightmost has the other subtype and the optional behaviour present.



<OP*.EC.(IPP*.(O+P).(1+D)*)*.(1+B.PY)>
<OP*.(IPP*.P.(1+D)*)*>
<OP*.EC.(IPP*.OP.D*)*.(1+B.PY)>
rel-part-del-shopping-cart
<OP*.EC.(IPP*.P.D*)*>
bill-part-del-shopping-cart
<OP*.EC.(IPP*.O.D*)*.B.PY>

Figure 7 Fragment of Figure 6 with behavioural subtypes based on message subtypes

What to do about mismatches

It was claimed above that an interoperation can take place if the two parties have different behaviour protocols, so long as one subsumes the other. On closer inspection, the situation is more complicated than that. If, for example, the purchaser follows the *unique-purchase* protocol (Figure 6) and the supplier the *deliver-purchase* protocol, there is no problem, but if the protocols are reversed, the interaction may not succeed.

Whether there is a problem depends on whether the partner supporting the subtype protocol can force the partner supporting the supertype protocol to limit its behaviour. Where the purchaser follows the *unique-purchase* protocol and the supplier the *deliver-purchase* protocol, the restriction to a single product per order is under the control of the purchaser. In the reverse case, the purchaser may want to place an order for several products, but the supplier can accept only one product per order. Here the problem is not too severe, since the purchaser can adjust its ordering strategy to the capabilities of the supplier. The purchaser can control the interaction protocol, since it initiates all sequences of events.

But suppose the purchaser follows the *del-shopping-cart* protocol and the supplier the *part-del-shopping-cart* protocol. In this case, the supplier initiates the deliveries so may break the order into parts even though the purchaser can’t handle multiple deliveries. The supplier must limit its behaviour to the capabilities of the purchaser.

In general, both parties must agree to behave according to the most constrained (subtype) protocol, so their interaction strategies must be flexible enough to support subtype behaviour protocols.

This leads to a consideration of what can be done if the two parties support protocols neither of which subsumes the other. The possibility exists that there may exist an acceptable common subtype.

Calculation of the greatest common subtype is easy if the only difference between the protocols is the presence or absence of the iteration operator. Suppose one party supports the *del-shopping-cart* protocol which allows more than one product per order but a single delivery for each product. The definition (13) is repeated below (17) for convenience

del-shopping-cart = <offer-product*.(identify-product-price*.purchase.deliver)*> (17)

The other party follows a protocol we will call *simple-part-del* in which one product can be ordered at a time, but partial deliveries are allowed (18)

simple-part-del = <offer-product*.(identify-product-price*.purchase.deliver*)> (18)

The greatest common subtype is computed simply by deleting the iteration operator at any place in the regular expressions where it does not occur in both, giving in this case (19)

common = <offer-product*.(identify-product-price*.purchase.deliver)> (19)

If both parties can limit their behaviour to *common*, then an interaction may proceed. This will require an exchange of messages to establish.

The same sort of tactic can work if the difference is in message subtypes. Suppose one party supports *rel-part-del-shopping-cart* of Figure 7, while the other supports *bill-part-del-shopping-cart*. The greatest common subtype algorithm can be augmented by replacing a clash between message subtypes by a common subtype of both, if such exists.

In any case, the initial exchange establishing a common protocol must have a state in which the attempted connection fails.

Conclusion

We have shown how the purdurant behaviour of agents can be organized into a system of subtypes which is based on the same principles as and is compatible with the subtype systems used to organize the endurants through which and on which the behaviour occurs. An agent can be programmed with arbitrarily intelligent behaviour within a standardized behaviour protocol. It can seek out other agents with which it can interact using the same subsumption mechanisms for behaviour protocols as it does for its message repertoire and the objects about which the interaction is about.

The subsumption structure of the behaviour protocols has been constructed from existence dependency and essential parthood, concepts central to the OntoClean meta-ontology, and the whole structure has been based on the endurant/perdurant distinction of DOLCE applied to the material ontological category of institutional fact. It shows how basic formal and material ontology can help us organize and comprehend the world of computerized agents.

References

1. R.M. Colomb, C.N.G. Dampney and M. Johnson “Category-theoretic fibration as an abstraction mechanism in information systems” *Acta Informatica*. Vol. 38, pp. 1-44, 2001

2. C.N.G. Dampney “Specifying a semantically adequate structure for information systems and databases.” *Proc. of the 6th. Int. Conf. on the Entity-Relationship Approach*, New York, 9-11 Nov., 1987; North Holland Publishing Company pp 143-164.

3. A. Gangemi, N. Guarino, C. Masolo and A. Oltramari (2001) “Understanding top-level ontological distinctions” *Proc IJCAI Workshop on Ontologies and Information Sharing*, 2001.

4. T.R. Gruber “Toward principles for the design of ontologies used for knowledge sharing” Knowledge Systems Laboratory Stanford University Tech. Rep. KSL 93-04, 1993

5. C. Masolo, S. Borgo, A. Gangemi, N. Guarino, A. Oltramari and L. Schneider (2002, Aug.) WonderWeb deliverable D17 version 2.0 LADSEB-CNR, Italy [online] http://www.ladseb.pd.cnr.it/infor/Ontology/Papers/OntologyPapers.html

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

7. P. Simons *Parts: a Study in Ontology* Oxford: Oxford University Press, 1987

8. P. Simons “How to exist at a time when you have no temporal parts” The Monist vol. 83, no. 3, July 2000

9. M. Snoeck and G. Dedene “Generalization/specialization and role in object-oriented conceptual modeling” *Data and Knowledge Engineering* vol. 19 no. 2, pp. 171-195, 1996

10. M. Snoeck and G. Dedene “Existence dependency: the key to semantic integrity between structural and behavioural aspects of object types” *IEEE Transactions on Software Engineering*, vol. 24, no. 4, 1998.

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

12. C. Welty and N. Guarino “Supporting ontological analysis of taxonomic relationships” *Data and Knowledge Engineering* vol. 39, no. 1, pp. 51-74, 2001