ISSN 0103-9741

Monografias em Ciência da Computação
n° 40/05

LATTICE OPERATORS OVER ENTITY CLASSES

Antonio L. Furtado

Departamento de Informática

Monografias em Ciência da Computação, No. 40/05 ISSN 0103-9741
Editor: Prof. Carlos José Pereira de Lucena December 2005

LATTICE OPERATORS OVER ENTITY CLASSES[*]

Antonio L. Furtado

furtado@inf.puc-rio.br

Abstract: The static conceptual specification of information systems, adopting the Entity-Relationship (ER) model, has been, for a long time already, extended with *is_a* and *part_of* links, whereby specialization/ generalization and decomposition/ aggregation of entity classes can be introduced. Usually these links impose tree-like hierarchies, although more general cases of partial ordering are quite often encountered in practice. Here we shall consider *lattice hierarchies*, on which two dual operators are defined. Using these operators, we show one way to classify entity instances on the basis of their currently known properties, in turn organized as lists of property:value pairs (*frames*). A broad overview of the facilities offered by our method is presented, including queries involving structured requirements, also expressed by frames, which look for approximately matching instances.

Keywords: Information systems, ontologies, semantic hierarchies, lattices, logic programming.

Resumo: A especificação conceitual estática de sistemas de informação, adotando o modelo Entidades-Relacionamentos (ER), tem sido, já por bastante tempo, estendida pela incorporação de elos é_um e parte_de, através dos quais a especialização/ generalização e a decomposição/ agregação de classes podem ser introduzidas. Usualmente esses elos impõem hierarquias do tipo arborescente, embora casos mais gerais de ordem parcial sejam com freqüência encontrados na prática. Aqui iremos considerar *hierarquias de reticulados*, nos quais um par de operadores duais são definidos. Usando esses operadores, mostramos como classificar instâncias de entidades com base em suas propriedades correntemente conhecidas, por sua vez organizadas em listas de pares propriedade:valor. Uma visão sumária das facilidades oferecidas por nosso método é apresentada, incluindo consultas envolvendo requisitos estruturados, também expressos na mesma forma de lista de pares, visando encontrar no banco de dados instâncias aproximadamente correspondentes.

Palavras-chave: Sistemas de informação, ontologias, hierarquias semânticas, reticulados, programação em lógica.

[*] This work has been partly sponsored by the Ministério de Ciências e Tecnologia da Presidência da República Federativa do Brasil.

In charge for publications

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br

"Can Woman ever be constant? No, alas, always variable in upper case. However woman, or even 'Woman', acceptable constants shall remain."

Apochryphal Akkadian proverb

1. Introduction

The notion of generalization/ specialization of entity-classes, expressed by connections established by way of *is-a* links, was very soon [SN,BC] incorporated into the Entity-Relationship model, and, more recently, has gained further attention as a major structural feature of *ontologies* [RG].

In the simplest cases, the specialized classes are structured in tree-like hierarchies. However, to take into account the rather common existence of classes that simultaneously specialize more than one class, we shall study the case where the partial order induced by is_a produces a *lattice* structure. From algebraic theory [MB] we know that lattices are associated with two basic operators, called *meet* and *join*, whose operands are pairs of nodes in the lattice, and whose result is also a node. To indicate the application of meet to nodes N1 and N2, yielding node M as result, we write: $N1 \wedge N2 = M$. Similarly, for join, yielding node J, we write: $N1 \vee N2 = J$. According to the definition of these operators, M will be the closest node into which converge the paths (link sequences) going up from N1 and N2, whereas J corresponds to a donward convergence. In a so-called complete lattice, M and J always exist and are unique.

In the realm of information systems, entity classes are the nodes, a universal class simply called *entity* serves as the root, and an empty class called *nil* serves as the bottom node. Although a strict obedience to the formal requirements of complete lattices may perhaps contribute to a sound design discipline for information systems in general, we shall not insist on their rigorous obedience here, since this would often lead to the creation of additional entity classes which one would not care to distinguish in practice – in cases, for instance, where more than one criterion is separately used to introduce specialized classes.

In this paper, we shall consider the problem of *classifying an entity instance* on the basis of those of those of its *properties* which are currently known. These comprise the values of its *attributes*, and its participation in *binary relationship* instances, which may in turn have attributes with assigned values. A convenient data structure for collecting all properties of an entity or relationship instance, often used in artificial intelligence applications, is the *frame* structure [SC], which takes the form of a list of property:value pairs.

Besides instance frames, we shall consider entity class frames, in which the value of each property will, in general, appear as a variable. Whenever a class C' is indicated, via an is_a link, to be a specialization of another class C, two requirements must be satisfied by the specification of C':

(1). all properties of C are *inherited* by C';
(2). C' must prove to be more restrictive than C, either by the presence of additional properties of its own, and/or by having a more restricted value domain for one or more properties inherited from C.

By imposing requirement (2), we are excluding the possibility of defining cases of purely extensional specialization, wherein C' is considered a specialization of C merely because it denotes a set of instances that is a subset of those pertaining to C.

To restrict the value domain of a property P, the V component of the P:V pair figuring in the frame of C' should be indicated as something else than a variable. More specifically, V can be:

(a). if P is an attribute – a constant value, which may be the name of a value domain;
(b). if P is a relationship – the identifier of a specific entity instance, or the name of an entity class.

The paper is organized as follows. Section 2 reviews the clausal notation whereby we adapt the Entity-Relationship concepts to the Logic Programming paradigm. Section 3 elaborates on the advantages of frames in highly-structured object-oriented queries. The central topic of the paper, frame-based classication, is covered in section 4, and related explanations and compatibility issues are treated in section 5. The matching techniques used for classification are seen, in section 6, as instrumental to finding instances that satisfy the desired requirements as closely as possible, even though in an incomplete way. Section 5 reports our first steps towards the incorporation of part_of hierarchies. Section 8 contains concluding remarks.

2. Clausal representation of both data and meta-data

2.1 Clauses for specifying the schema

Using notational conventions already described in previous works [Fu], the schema of an information system (meta-data) will be represented throughout this paper by logic programming *clauses*, which specify the entity classes and their attributes, as well as the binary relationships between entity classes and their attributes:

- entity(<entity class>,<identifying attribute name>)
- attribute(<entity class>,<attribute name>)
- relationship(<relationship name>,[<entity class>,<entity class>])
- attribute(<relationship name>,<attribute name>)

Examples:

```
entity(company,denomination).
attribute(company,headquarters).
```

```
relationship(works,[employee,company]).
attribute(works,start_year).
```

Virtual attributes, whose values are determined on demand, rather than stored extensionally, are associated with evaluation rules:

- virtual_attribute(<entity class>,<attribute name>,<evaluation rule>)

as in:

```
virtual_attribute(student,monthly_pay,r3).
r3([fee:V1,lodging:V2,restaurant:V3],[monthly_pay:V4]) :-
  sum([V1,V2,V3],V4), not (V4 = 0).
```

An entity class may specialize one or more entity classes, as indicated by is_a links:

- is_a(<entity class>,<entity class>)

such as, for example:

```
is_a(student,person).
is_a(employee,person).
is_a(trainee,student).
is_a(trainee,employee).
is_a(law_student,student).
is_a(tech_student,student).
```

We stipulate that all classes that specialize the same class must have the same identifying attribute. So, since name is the identifying attribute of person, it is also used as such for all specialized entity classes mentioned above. As usual, the other (non-identifying) attributes are inherited, recursively, along the is_a connections. Participation in relationships is likewise inherited.

As said in the Introduction, we adopt the assumption that what makes a class C' a specialization of a class C1 is that, besides the inherited properties (attributes and participation in relationships), C' must have additional properties to effectively distinguish it from C1. If C' also specializes another parent class C2, it will, simply through this circumstance of multiple inheritance, establish its distinction from both C1 and C2, the class trainee being an obvious example. But if C' specializes C1 alone, it will require either new properties or some sort of restriction in the allowed values of some properties inherited from C1. To restrict the values of attributes and the kind of entity instances associated by relationships, we provide, respectively:

- attr_val(<entity class>,<attribute name>,<restricted value>)
- rel_ent(<entity class>,<relationship name>,<restricted entity instance>)

Thus, the specialized classes tech_student, car_company and car_employee are fully characterized as follows:

```
entity(tech_student,name).
attr_val(tech_student,area,technical_sciences).
is_a(tech_student,student).

entity(car_company,denomination).
attr_val(car_company,headquarters,'Carfax').

entity(car_employee,name).
rel_ent(car_employee,works/1,car_company).
is_a(car_company,company).
```

As shown in these examples, the restriction may consist of the indication of a specific value, such as 'Carfax'. But, as a consequence of also allowing lattice-structured value domains, the restriction may be expressed by indicating a more specialized (sub-)domain. The attribute area has such structure; its root is simply area itself, and social_sciences, human_sciences and technical_sciences come next in the hierarchy. If a given student has, say, engineering (which is below technical_sciences) as his area, he will qualify as a tech_student. On the other hand, the name of a specialized entity class may be used to restrict the instances allowed to be related. Thus, a car_employee is defined by being restricted to work in a car_company. This example also serves to introduce a notational convention: recalling that we only admit binary relationships, we can write works/1 as if it were an attribute of employee (whereas works/2 would function as an attribute of company, the second participant in the works relationship).

A relationship can be introduced intensionally, by providing a conditional clause, as in:

```
relationship(gives_legal_assistance, [solicitor,employee]).
gives_legal_assistance([E1,E2]) :-
  works([E1,C]), works([E2,C]),
  area(E1,law), not area(E2,law).
```

where solicitor is a two-level removed specialization of employee, in view of the links is_a(trainee,employee) – already mentioned – and is_a(solicitor,trainee). But this creates a difficulty, which illustrates a more general problem: according to the clauses above, a solicitor would have the property gives_legal_assistance/1; in addition, as a consequence of inheritance from employee, he would also have gives_legal_assistance/2. This, however, represents a situation that one might wish to exclude. As an employee with legal training, a solicitor would be in a position to assist those of his colleagues who lack such training, but would not need, in turn, to receive similar help from another (as implied by the inherited -/2 property).

Our option to circumvent this inconvenience recognizes inheritance while explicitly barring its effect. It involves a special value, undef, which is used to express that an instance of an indicated entity class cannot have an actual value for the property in question. So, we specify for solicitor:

```
rel_ent(solicitor,gives_legal_assistance/2,undef).
```

Note that this solution would equally apply to the quite common problem caused by properties which are typical of a general entity class, but by no means mandatory for all its sub-classes. By default, birds are reputed to have the flight property, so that most instances would be qualified somehow (e.g. by a Boolean *yes* value) for the flight attribute, whereas some would not, due perhaps to some accident or congenital malformation. But in the penguin sub-class (and also ostrich, etc.) the attribute would be restricted by the clause attr_val(penguin,flight,undef), to indicate that no instance thereof can even receive any value at all for this attribute.

2.1 Clauses for specifying instances

The schema clauses define the kinds of instances that can occur in the mini-world of the information system, and what properties they may have. Fulfilling their role as *meta-data*, they in turn induce the equally clausal notation to express facts – i.e. the represented *data* – describing the existing instances. Having included the appropriate clauses in the schema, we shall be justified to represent data in clauses, according to the following correspondences:

- entity(<entity class>,<identifying attribute name>) ⇒ <entity class>(<identifier>)
- attribute(<entity class>,<attribute name>) ⇒ <attribute name>(<identifier>,<attribute value>)
- relationship(<relationship name>,[<entity class>,<entity class>]) ⇒ <relationship name>([<identifier>,<identifier>])
- attribute(<relationship name>,<attribute name>) ⇒ <attribute name>([<identifier>,<identifier>],<attribute value>)

Notice that entity instances are denoted by their *identifiers* (i.e. the values of their identifying attributes). In this paper, we use name and denomination as, respectively, the identifying attributes of person and company (and of their descendants in the is_a hierarchy), and identifiers such as 'Jonathan' to designate an instance of person, and 'VL' for an instance of company. As might be expected, the binary relationship instances are denoted by pairs of identifiers indicating the participating entity instances.

Some examples are:

```
area('Jonathan',law).
salary('Jonathan',100).
works(['Jonathan','VL']).
start_year(['Jonathan','VL'],2001).
status(['Jonathan','VL'],permanent).
badge_numb('Jonathan',jur-134).
```

noting that the schema clauses authorizing the first, the third and the fourth instance clauses above are, respectively:

```
attribute(student,area).
relationship(works,[employee,company]).
attribute(works,start_year).
```

As a case of special interest, the schema clause:

```
entity(solicitor,name).
```

allows us to write the instance clause

```
solicitor('Jonathan').
```

and one may want to check whether this particular clause expresses a *classification* of the 'Jonathan' instance that happens to be coherent with the other facts supplied about him. To motivate the more detailed discussion to be presented later, we anticipate the remark that, in practice, we often have rather incomplete information about a number of entity instances, which may or may not be complemented later with more data being supplied as time goes by. The additional data may, in some cases, lead us to revise a previous classification.

Suppose that, at a an early moment, the only known fact about Jonathan is his study area, supplied by the clause area('Jonathan',law). This is enough to conclude that he is a student – but, actually, we can do better than that: he can more precisely be classified as a law_student, since he meets one declared requirement of this more specialized class. Assume that a second instance clause is supplied at a later moment, telling about his salary. Possibly the fact had been true before, but had not been communicated immediately, or else the fact was the recent result of a state-change affecting this person. In both cases, if Jonathan has a salary then he is an employee, but, being also a law_student, he meets the more stringent requirements of solicitor (understood, in old Brazilian terminology, as a trainee specialized in legal matters). The subject of how to classify with the available data will be treated in section 4. But, before going into this, we shall briefly show how the simultaneous availability of data and meta-data in a uniform clausal notation can pave the way for an ample variety of queries over the information system.

3. Frame-based query facilities

Entity classes and entity instances are objects about which there may exist a time-varying amount of information, scattered through a number of separate clauses, which may be the object of direct individual queries. For example:

- query on a schema clause:

```
?- attribute(employee,A).
A = salary ;
A = office_phone
```

- query on an instance clause:

```
?- salary('Jonathan',S).
S = 100
```

- query on an instance clause involving a virtual attribute:

```
?- virt(years_of_work(works(['Jonathan',C]),Y)).
C = VL
Y = 4
```

However, it is much convenient to combine all information pertaining to a given class or instance in a compact data structure, known as a *frame*, widely used, already for a long time, in Artificial Intelligence applications [SC]. Recall that frames can be represented as a lists of property-value pairs.

Frames for entity classes are built from the schema clauses. As an example, consider the frame of the tech_student class, obtained in response to a call to the predicate class_frame(<entity class>,<frame>):

```
?- class_frame(tech_student,Fc).
Fc = [area : technical_sciences,fee : A,gender : B,lodging : C,monthly_pay : D,
name : E,restaurant : F,room_phone : G,scholarship : H]
```

In general, all values of properties are given as variables in class frames. Exceptions occur only when some attr_val or rel_ent clause has been specified as a characterizing restriction of a specialized class. In Fc above, this happens with attribute area, as an effect of the clause attr_val(tech_student,area,technical_sciences).

In a quite similar way, we can ask for the frame of some instance. Suppose the database has some information about a person by the name of Bertillon. All pieces of information available about him will be put together and displayed in frame format, if a query with the predicate inst_frame(<identifier>,<frame>) is posed:

```
?- inst_frame('Bertillon',F).
F = [area : engineering,fee : 45]
```

For a more populated instance frame, the ubiquitous Jonathan is a better example:

```
?- inst_frame('Jonathan',F).
F = [area:law, badge_numb:jur-134, salary:100, gives_legal_assistance/1:['Lucy', 'Mina'],
works/1:'VL']
```

By inspecting F, we can predict that the name Jonathan will figure in replies to queries such as: Who has law as study area and earns a salary? Who has law as study area and works for a car_company? Who else works for the same company? The predicate

- find(<entity instance>,<frame>)

can be used in all such cases, the frame being formulated so as to express what is asked:

```
?- find(I,[area:law,salary:S]).
?- find(I,[area:law,works/1:car_company]).
?- find('Jonathan',[works/1:C]), find(C,[works/2:S]).
```

The third query, which goes in two steps, yields as result:

```
C = 'VL'
S = ['Jonathan', 'Lucy', 'Mina']
```

The frame of an entity class or entity instance contains information about the participation in relationships, but not about the relationship own attributes. These are kept in separate frames, corresponding to specific relationship instances, which can be retrieved via the predicate

- r_find(<relationship identifier>,<frame>)

in general queries such as:

```
 ?- r_inst_frame(I,F).

I = works(['Jonathan', 'VL'])
F = [denomination:'VL', name:'Jonathan', start_year:2001, status:permanent] ;

I = works(['Hugo', 'BK'])
F = [denomination:'BK', name:'Hugo', status:temporary] ;

I = works(['Lucy', 'VL'])
F = [denomination:'VL', name:'Lucy', status:temporary] ;

I = works(['Mina', 'VL'])
F = [denomination:'VL', name:'Mina', status:permanent] ;
```

For a more specific example, consider a query formulated by someone who, quite clearly, has only a vague idea of the current facts: "I met someone who is somehow related to Jonathan. Please tell me about the nature of this liaison. Also tell me all you know about this person. By the way, I seem to recall that the person was wearing a badge."

```
?- r_find(I,[name:'Jonathan',name:X]), inst_frame(X,F), on(badge_numb:N,F).

I = gives_legal_assistance(['Jonathan', 'Lucy'])
X = 'Lucy'
F = [badge_numb:inf-123, gives_legal_assistance/2:'Jonathan', works/1:'VL']
N = inf-123
```

Both relationship attributes involved in the query above are identifying attributes. For an example with ordinary non-identifying relationship attributes, we can ask whether Jonathan's job is permanent or temporary, and for how long he has been engaged in it. This second information is a typical virtual attribute, to be computed from the start_year attribute and from the current date known to the processing equipment. Virtual attributes are often relatively expensive to include in queries, which led us to provide separately a number of predicates, with the same name of the conventional ones except for an ending 'x', which do take into account all virtual attributes applicable:

```
 ?- r_findx(I,[name:'Jonathan',years_of_work:Y]).

I = works(['Jonathan', 'VL'])
Y = 4
```

Aggregate-type queries can also be asked on both entity and relationship frames, using the predicate

```
all(<term>,<frame>,<result>)
```

If <term> is simply a variable, also figuring in <frame>, then <result> will return the list of all values obtained by matching the frame against the database instances. Alternatively, <term> can take the form of eval(<arithmetic expression>), again returning a list of values in <result>. Finally, <term> can be a call to one of the usual functions on aggregates, in which case <result> will show the single global value computed. The available functions are sum, count, countv (which counts only the values explicitly represented as numbers), max, min, and avg (average). A few examples follow:

```
?- all(F,[fee:F],R).
R = [50, 45, 100, nominal]

?-  r_all(I,[denomination:'VL',name:I,status:permanent],S).
S = [Jonathan, Mina]

?- all(eval(F+10),[fee:F],R).
R = [60, 55, 110]

?- all(sum(F),[fee:F],R).
R = 195

?- all(max(F),[fee:F],R).
R = 100
```

The find family of predicates (find, findx, r_find, r_findx, all, allx) looks for instances corresponding exactly to what is being asked. In section 5 we shall mention a more flexible device. But let us now consider the subject of classification, which is also involved in section 5, where class compatibility and explanations related to classification criteria are important issues.

4. Classification using lattice operators on frames

Lattice structures pervade many domains. Sets come immediately to mind, recalling that the meet operator corresponds to set intersection and join to set union. Term domains also form lattices, over which unification corresponds to join and its dual, namely most specific generalization [Kn,Pl,Re,WM], corresponds to meet. Greatest common divisor (gcd) and least common multiple (lcm) are the join and meet operators over the lattice of natural numbers under divisibility.

As an intuitive introduction to our approach to classification, we shall consider the gcd/ lcm case. Natural numbers can be displayed in a frame structure similar to the property-values lists we have been using, treating each prime factor as a property and the power raised to which the factor appears in the number as value. With this convention, the numbers 24 and 60 can be represented by frames F1 and F2 below:

F1 = [2:3, 3:1]
F2 = [2:2, 3:1, 5:1]

The result of gcd(24,60) is $24 \wedge 60 = 12$, and of lcm(24,60) is $24 \vee 60 = 120$.

Let us draw a lattice-like diagram to put together these four numbers, side by side with their frames:

12 = [2:2, 3:1]

24 = [2:3, 3:1]   60 = [2:2, 3:1, 5:1]

120 = [2:3, 3:1, 5:1]

As known from elementary arithmetic, the gcd contains only the prime factors held in common, with the smallest exponents (24 has 2:3, 60 has 2:2, so 12 has 2:2). And the lcm contains all factors, common or not, with the largest exponents. This is analogous to what happens with the properties of entities connected by is_a links: all properties, common or not, are passed down by *inheritance*, since moving down means to specialize, which is tantamount to requiring that all properties (their union) should be cumulatively satisfied by the specialized instances. Conversely, moving up, i.e. generalizing, involves retaining only what is held in common (the intersection). Compare the above numerical example to:

person - [age: 25]

employee - [age: 25,salary:100]   student - [age: 25,area: chemistry]

trainee – [age: 25,salary:100,area: chemistry]

Now consider the two nodes in the second level of this 4-level hierarchy. Moving up to level 1 means to retain only what is shared, in this case the single property age: 25. Classes employee and student are thus *generalized* to class person. Descending to level 4 is accompanied by passing down all properties, common or not. But the common properties must agree with respect to the assigned values; two different ages, say 25 and 30, would produce a *conflict*, causing the process to fail. The resulting class trainee is clearly more restrictive than employee and student, since it puts together all the requirements of these two classes, which confirms the intuition that it *specializes* both of them.

Moving up and moving down correspond, in the lattice of the natural numbers, to the meet and join lattice operations, respectively. Likewise, in the is_a lattice of entity classes, we have that:

employee $\wedge$ student = person
employee $\vee$ student = trainee

or, equivalently, working on the corresponding frames:

[age: 25,salary:100] $\wedge$ [age: 25,area: chemistry] = [age: 25]
[age: 25,salary:100] $\vee$ [age: 25,area: chemistry] = [age: 25,salary:100,area: chemistry]

The above information could certainly refer to one same real-world object, a person who, at the same time, is a student and an employee, hence a trainee (if we so define this concept). Now let us consider a slightly different situation. Look again at the case of Bertillon, already mentioned in the previous section. Suppose that all that is recorded in the database about him is the information collected in frame F, to be compared with frame Fc:

F = [area: engineering, fee: 45]
Fc = [area : technical_sciences,fee : A,gender : B,lodging : C,monthly_pay : D,
name : E,restaurant : F,room_phone : G,scholarship : H]

which is the frame of class tech_student, also seen before.

F includes of course a considerably larger number of properties than Fc. But it is clear that F *matches* the larger – and also more general – frame Fc, in that all properties of F are accounted for in Fc, and there are no conflicting values in the two frames. Variable A, appearing in fee:A of Fc, can clearly be *unified* with the number 45, and the value engineering of area in F is, as mentioned previously, a specialization of technical_sciences in the value domain of attribute area. In other words, the meet and join pair of operations that we just illustrated works with an equally positive result in this case:

area: engineering $\wedge$ area: technical_sciences = area: technical_sciences
fee: 45 $\wedge$ fee: A = fee: A

area: engineering $\vee$ area: technical_sciences = area: engineering
fee: 45 $\vee$ fee: A = fee: 45

This meet/ join test provides the basis for defining, in correspondence with is_a (which, in formal parlance, is a *partial-ordering operator* on entity classes), a partial-ordering operator on frames, <frame> <= <frame>, which succeeds, instantiating any constituent variables as a side effect, whenever the first frame can be unified with or is a specialization of the second frame. To ensure coherence between the two partial-ordering operators, given two entity classes C1, with class frame F1, and C2, with class frame F2, it should be true that F1 <= F2 if and only if C1 is_a C2. Moreover, since being instance of a class is a basic level form of class inclusion, if I, with frame F, is an instance of a class C, with frame Fc, then it is permissible to claim by extension that I is_a C, and hence F <= Fc should hold. As a consequence, the query below will succeed:

```
?- inst_frame('Bertillon',F), class_frame(tech_student,Fc), F <= Fc.
```

Matching instance frames against class frames can thus be regarded as the central task in the classification process. However, as will be discussed in the sequel, one still has to cope with the incompleteness of the information available about a given database instance, and must devise some suitable criteria for choosing among more than one candidate class.

Returning to the Bertillon example, we saw that he might well be classified as a tech_student. Informally speaking, all properities of Bertillon known to us (until this instant of time) are accounted for, and he has no properties that are lacking in, or conflicting with, the specification of tech_student. But, instead of tech_student, could Bertillon be a trainee? Yes, if we knew that he had other properties, such as salary, for example – but right now we do not know that, and therefore calling him a trainee would be, for the time being, an unwarrated conclusion. Could we call him simply a student? Yes, but then we would not take advantage of the information we have about his study area. On the other hand, we could not call him a law student (assuming that the area property is single-valued for each entity instance), since there is a value conflict, indicated by the failure of the join

area: law $\vee$ area: technical_sciences

Informally speaking, we base our classification method, whereby we try to fit an instance frame Fi with some class frame Fc of a class C, on three criteria, in decreasing order of importance:

a. the number of *properties shared* by Fi and Fc;
b. the number of *value hits*, i.e. the cases wherein properties in Fc require restricted values, which happen to be present in Fi;
c. the *depth* to which one goes down to find C, i.e. its level in the lattice of entity classes.

As might be expected, larger measures for a and b increase the fitness, whereas c is to be taken inversely. To compare the fitness of a given instance to two candidate classes C1 and C2, one lexicographic comparison is enough, as expressed in Prolog notation. C1 will be a better fit than C2 whenever:

```
[a1, b1, -c1] @> [a2,b2,-c2]
```

For Bertillon, these triples for three candidate classes are, respectively:

student –	[2,0,-2]
tech_student –	[2,1,-3]
trainee –	[2,0,-3]

So, in accordance with our proposed criteria, he is classified as a tech_student. Yet we admit that his properties are *compatible* with the two other classes. The same cannot be said of class law_student, due to the value conflict with respect to the area attribute.

In slightly more detail, our classification algorithm tries to fit an instance frame Fi against a class frame Fc as follows. Classes are taken in breadth-first order, in view of criterion c. It goes down in the lattice until finding classes sharing *all* properties indicated in Fi (perfect satisfaction of criterion a) – which is true if the result of Fi $\wedge$ Fc retains all properties of Fc. And meanwhile it makes sure that there are no value conflicts with the class properties – which is true if the execution of Fi $\vee$ Fc does not fail. Having found such classes, it looks for (possible) specializations which can offer the maximum number of value hits with Fi (criterion b), in an attempt to refine the classification that may carry it further down in the lattice. Only one class is finally retained, although there may be more than one optimal solution.

Several predicates are available in connection with classification, including:

- classify_frame(<frame>,<entity class>)
- inst_of(<identifier>,<entity class>)
- confirm(<identifier>,<entity class>)
- revise(<identifier>,<old entity class>,<new entity class>)
- expand_frame(<frame>,<entity class>,<full frame>)

As often happens in Prolog, each argument of inst_of can be variable or constant. So, for example, we can obtain for Bertillon:

```
?- inst_of('Bertillon', C)
C = tech_student
```

or, letting the first argument be a variable:

```
?- inst_of(I,solicitor).
I = Hugo ;
I = Jonathan
```

or, with variables in both positions:

```
?- inst_of(I,C).
I = Hercule
C = trainee ;

I = Bertillon
C = tech_student ;

I = Dupin
C = student
........
etc.
```

Some classes involve virtual attributes in their specification, such as active_company, which is a company having at least an indicated number of people working for it who have the value permanent for the relationship attribute status. As before, versions ending with 'x' of these predicates are available:

```
?- inst_ofx(I,active_company).
I = VL
```

The confirm predicate classifies an instance I in its first argument, showing the obtained class C in the second argument. If C is spelled out by the user, the predicate nevertheless checks whether the classification is correct. And then it inserts a clause of the form C(<identifier>) in the database (via the built-in assert predicate). If this is done for Bertillon, by:

```
?- confirm('Bertillon',tech_student).
```

the clause tech_student('Bertillon') will be inserted. And if later it is recorded in the database that he is earning a salary, a call to predicate revise:

```
?- revise('Bertillon',Old,New).
```

will return Old = tech_student, New = trainee, and will retract tech_student('Bertillon') and insert trainee('Bertillon'). Both predicates ask the user's explicit permission before performing these database updates.

The expand_frame (and expand_framex) predicates are useful to add properties to an instance frame by joining it with a class frame. The added properties will usually appear as uninstantiated variables (except when restricted values have been specified, and thus figure in the class frame). If the second argument, referring to an entity class, is a variable, the frame will first be classified and then joined with the obtained class frame; however the user can prefer to indicate a class explicitly, which will be used if it is found to be compatible with the classification determined by the system. So, for Dupin, whom as seen above is now classified as a student, but could in the future be found to be a law_student, the two eventualities can be explored:

```
?- inst_frame('Dupin',F), expand_frame(F,C,Fe).

F = [fee : 100,lodging : 25,restaurant : 18]
C = student
Fe = [area : A,fee : 100,gender : B,lodging: 25,monthly_pay : C,name : D,restaurant : 
18,room_phone : E,scholarship : F]

?- inst_frame('Dupin',F), expand_frame(F,law_student,Fe).

F = [fee : 100,lodging : 25,restaurant : 18]
Fe = [area : law,fee : 100,gender : A,lodging : 25,monthly_pay : B,name : C,restaurant : 
18,room_phone : D,scholarship : E]
```

The inst_framex (or r_inst_framex for relationship instances) predicate is particularly helpful: it expands the initial frame with the properties of the class determined by the system, evaluates all virtual properties over this extended frame, which is again extended with the virtual properties obtained, and finally drops all properties whose values are still uninstantiated variables or are marked with undef. A few examples follow, noting that contact, monthly_pay, revenue, situation, size, and years_of_work are all virtual properties:

```
?- inst_framex('Hercule',F).
F = [contact:[room-123, office-456], monthly_pay:50, name:Hercule, revenue:130,
salary:100, scholarship:30]

?- inst_framex('VL',F).
F = [denomination:VL, headquarters:Carfax, situation:operational, size:3,
works/2:[Jonathan, Lucy, Mina]]

?- r_inst_framex(works(['Jonathan','VL']),F).
F = [denomination:VL, name:Jonathan, start_year:2001, status:permanent,
years_of_work:4]
```

Classification can also be usefully applied to sets of two or more frames. Once a set of frames S is formed, via, for example, the built-in setof predicate, one can, among other things:

1. obtain a frame M, as the most specific generalization of the frames in S
2. determine what is the most specific general classification that may cover all the frames in S
3. obtain the set of different classifications attributed to the frames in S

Predicates frame_msg, classify_group, and all_in_group accomplish each of the above tasks. Frames such as M in item 1, obtained by applying most specific generalization over observed instances, may contribute to *data mining* efforts, in that they capture what these instances have in common, which will be useful knowledge if they are typical enough to constitute a significant sample. And M can be used as a pattern to be matched against other instances arising in the future.

As a simple example encompassing the three tasks, consider the set of frames of all persons who work for company VL.

```
 ?- setof(F, I^(inst_framex(I,F), on(works/1 : 'VL', F)), S),
    frame_msg(S,M), classify_group(S,C), all_in_group(S,Cs).

S = [[area : law, badge_numb : jur-134, name: Jonathan, salary : 100,
gives_legal_assistance/1 : [Lucy, Mina], works/1:VL],
[badge_numb : inf-123, name : Lucy, gives_legal_assistance/2 : Jonathan, works/1 : VL],
[name : Mina, gives_legal_assistance/2 : Jonathan, works/1 : VL]]

M = [name : employee, works/1 : VL]
C = employee
Cs = [car_attendant, car_employee, solicitor]
```

5. Explanations and compatibility evaluations

Whenever we want to question why a certain classification was not chosen by the implemented algorithm, we can do so via the predicate

- why_not(<identifier>,<entity class>,<explanation>)

whose behaviour is best illustrated by a series of examples. In two situations, the answer implies that the indicated class is not compatible: either because there are no explicit properties in common, or because there is a value conflict. Here, BK is in fact a company, not a person. On the other hand, since its headquarters lie outside Carfax, it is denied to be a car_company:

```
?- why_not('BK', person, A).
no_explicit_properties_in_common

?- why_not('BK', car_company, A).
conflicting_values([headquarters:Bykeville \= Carfax])
```

In three other situations, the indicated classification is not wholly incompatible, but is inferior to the one found by the system (as judged through the adopted criteria):

```
?- why_not('Hugo', law_student, A).
better_match(solicitor, extra_properties([works/1:BK]), props_share: (3>2))

?- why_not('Bertillon', student, A).
better_match(tech_student, [area:technical_sciences], value_hits: (1>0))

?- why_not('Hercule', solicitor, A).
better_match(trainee, depth: (3<4))
```

But, as said before, there can exist more than one optimal solution according to the criteria applied by the system. Consider the two successive queries below:

```
?- inst_ofx('VL',C).
C = active_company

?- why_notx('VL', car_company, A).
equally_viable(car_company, active_company)
```

wherein, conceivably, the first query did not return the result expected by the user. Suppose now that the user wants to learn what is the import of "equally_viable" in this case. The predicate

- compatible(<identifier>,<entity class>,<evaluation>)

can be used to explore and evaluate all compatible classifications (including the preferred one). So, in order to further investigate the case of VL, one can enter:

```
?-  compatiblex('VL',C,E).

C = company
E = [depth:1, props_class:5, props_inst:5, props_share:5, value_hits:0] ;

C = car_company
E = [depth:2, props_class:5, props_inst:5, props_share:5, value_hits:1] ;

C = active_company
E = [depth:2, props_class:5, props_inst:5, props_share:5, value_hits:1]
```

which allows to conclude that, in terms of number of shared properties, number of value hits, and depth in the lattice, the two last classes received the same [5,1,-2] mark.

Obviously, there is nothing absolute about our chosen classification criteria. One may want to expand them or even to change them radically. We require a minimum of one property shared to justify assigning an instance to a class; a larger threshold can be imposed, or it may be argued that certain properties are more characteristic of a class than others. And there are other features, besides properties and their value domains, that may be brought into the picture to characterize a class, some of which could be expressed by rules (e.g. integrity constraints). And – as a aspect for future research, still within the topic of classification – there are the dynamic aspects, referring to how the instances of each class are affected by state changes, especially those associated with the execution of *operations* by the various agents playing different roles in the information system [Fu].

6. Searching for approximate solutions

The notion of compatibility evaluation leads to a mechanism to find – and compare – instances that do not necessarily satisfy all requirements formulated in a given search frame. Matching an instance I with frame Fi against a search frame Fs is like classifying Fi in an *arbitrary* entity class. A fundamental difference is that classification involves comparing Fi with the frames of the various predefined entity classes across their lattice structure, until the process succeeds at some *depth*. For arbitrary frames, the same as for classification, it makes sense to count the number of properties shared, as well as of value hits, but of course there is no question of depth – since only one frame comparison takes place.

When doing this kind of loosely restricted search with an arbitrary frame, all instances having at least one property in common may be of interest, even in the presence of conflicting values – although we rate conflicting answers more poorly than the compatible ones. The criterion for lexicographic comparison adopted here involves three numbers, counting, in this order of importance, how many conflicts (c), properties shared (ps), and value hits (vh) has the frame of an instance with respect to the given search frame. An instance I1 will be taken as a better approximation than I2 if:

```
[-c1,ps1,vh1] @> [-c2,ps2,vh2]
```

The following predicates are supplied, together with their 'x' versions. As with the find family introduced before, we can look either for entity or for relationship instances. Moreover we can either ask for all approximate answers, or just for those reaching the best rate:

```
- match(<identifier>,<frame>,<evaluation>)
- best_match(<identifier>,<frame>,<evaluation>)
- r_match(<identifier>,<frame>,<evaluation>)
- r_best_match(<identifier>,<frame>,<evaluation>)
```

To begin with, we show a pair of queries on entity instances, in which not even the best solution happens to meet all requirements. Hercule and Dupin are not reported to work in a car_company, and no fee information is supplied for Jonathan, Lucy and Mina. In addition, only Jonathan is known to study law. Two conflicting cases occur: Hugo works for a company that is not a car_company and Bertillon's area is engineering.

```
?- match(I,[area:law,works/1:car_company,fee:F],E).

I = Hercule
F = 50
E = [conflicts:0, props_class:3, props_inst:5, props_share:1, value_hits:0] ;

I = Bertillon
F = 45
E = [conflicts:1, props_class:3, props_inst:2, props_share:2, value_hits:0] ;
```

```
I = Dupin
F = 100
E = [conflicts:0, props_class:3, props_inst:3, props_share:1, value_hits:0] ;

I = Hugo
F = nominal
E = [conflicts:1, props_class:3, props_inst:3, props_share:3, value_hits:1] ;

I = Lucy
F = A
E = [conflicts:0, props_class:3, props_inst:3, props_share:1, value_hits:1] ;

I = Jonathan
F = A
E = [conflicts:0, props_class:3, props_inst:5, props_share:2, value_hits:2] ;

I = Mina
F = A
E = [conflicts:0, props_class:3, props_inst:2, props_share:1, value_hits:1]

?- best_match(I,[area:law,works/1:car_company,fee:F],E).

I = Jonathan
F = A
E = [conflicts:0, props_class:3, props_inst:5, props_share:2, value_hits:2]
```

And now let us see a search for relationship instances; four instances are found, which, again, fail to satisfy all requirements (start_year not informed or status not temporary):

```
 ?- r_match(I,[start_year:Y,status:temporary],E).

I = works([Jonathan, VL])
Y = 2001
E = [conflicts:1, props_class:2, props_inst:4, props_share:2, value_hits:0] ;

I = works([Hugo, BK])
Y = A
E = [conflicts:0, props_class:2, props_inst:3, props_share:1, value_hits:1] ;

I = works([Lucy, VL])
Y = A
E = [conflicts:0, props_class:2, props_inst:3, props_share:1, value_hits:1] ;

I = works([Mina, VL])
Y = A
E = [conflicts:1, props_class:2, props_inst:3, props_share:1, value_hits:0]
```

```
?- r_best_match(I,[start_year:Y,status:temporary],E).

I = works([Hugo, BK])
Y = A
E = [conflicts:0, props_class:2, props_inst:3, props_share:1, value_hits:1] ;

I = works([Lucy, VL])
Y = A
E = [conflicts:0, props_class:2, props_inst:3, props_share:1, value_hits:1]
```

Search frames, as said, are arbitrarily chosen by the user, but the choice may have some well-motivated origin. For example, we may ask which persons look more like Mina, in terms of their properties. In such cases an instance frame is retrieved to be used as search frame. As expected, the answer includes – but is not limited to – Mina herself:

```
?- inst_frame('Mina',F),best_match(I,F,R).

F = [gives_legal_assistance/2:Jonathan, works/1:VL]
I = Lucy
R = [conflicts:0, props_class:2, props_inst:3, props_share:2, value_hits:2] ;

F = [gives_legal_assistance/2:Jonathan, works/1:VL]
I = Mina
R = [conflicts:0, props_class:2, props_inst:2, props_share:2, value_hits:2]
```

Instead of using an instance frames as is, we may wish to modify it first, pehaps to relax certain requirements. Several features are available for frame manipulation, including the frame_purge predicate demonstrated below, which serves to remove from a frame F the properties whose values are in conflict with those of frame G:

```
?- inst_frame('Hugo',F),inst_frame('Dupin',G),frame_purge(F,G,F1).

F = [area:law, fee:nominal, works/1:BK]
G = [fee:100, lodging:25, restaurant:18]
F1 = [area:law, works/1:BK]
```

An extreme case of relaxing the requirements expressed by a frame is to convert the values of all its properties into variables, which is the objective of predicate mk_pv(<frame>,<frame>). The mk_pv predicate can be applied selectively in the <transformation predicate> position of another very general frame-manipulation predicate:

- replace(<property pairs>,<modified property pairs>,<trasformation predicate>,<frame>, <new frame>)

As shown below, this can be used to transform into variables the values of properties fee and works/1 in the frame F of Hugo:

```
?- inst_frame('Hugo',F),  replace([fee:_,works/1:_],L,mk_pv,F,F1).

F = [area:law, fee:nominal, works/1:BK]
L = [fee:A, works/1:B]
F1 = [area:law, fee:A, works/1:B]
```

7. Initial steps towards part_of hierachies

The design and implementation of part_of hierarchies is still at an early stage in our project. As done for is_a hierarchies, we allow part_of hierarchies to also induce lattice structures on entity classes. While is_a introduces the idea of specialization/ generalization, the part_of links bring in decomposition/ aggregation.

Using a familiar example, a company can possess departments as components. In turn, a branch can be part of one or more departments (understanding a branch as a local organization unit, under the joint control of various central departments). The major units, companies in our case, serve as roots, and any part P has as identifier a partially-ordered sequence of pairs of the form part:level (where level expresses the position of the part in the hierarchy), beginning with the root at level 1 and continuing down to P. Assuming that company VL has departments product and sales, and branch b_vp is part of both these departments, the full identifiers of these three sub-units are (as can be determined via pname(<individual part>,<full part identifier>)):

```
product - [VL:1, product:2]
sales - [VL:1, sales:2] ;
b_vp - [VL:1, product:2, sales:2, b_vp:3]
```

Because lattice structures are more general than simple tree-like structures, these flat sequence identifiers do not always allow to perceive the detailed links explicitly. For this purpose, predicate struct(<root>,<nested structure>) is provided. In our example, we have, for companies BK and VL:

```
?- struct(R,S).

R = BK
S = [BK, [audit], [product]] ;

R = VL
S = [VL, [sales, [b_vp]], [product, [b_vp]]
```

Both the lattice of classes and the lattices of instances (like those of VL and BK, above) can be explored by appropriate selectors, which permit to move across the elementary part_of links or, recursively, across part_of chains. The (fully recursive) examples below should be self explanatory. Notice that it is necessary to include the root as parameter, since parts below the roots are like the indeterminate *weak entities* category of the Entity-Relationship model [BC]. It would sound ambiguous to talk of a product department, when we have both "product department of VL" and "product department of BK". Also notice that the part_of connections are inherited along the is_a connections.

```
?- part_of_r(X,Y).

X = dept
Y = car_company ;

X = dept
Y = active_company ;

X = dept
Y = company ;

X = branch
Y = dept ;

X = branch
Y = car_company ;

X = branch
Y = active_company ;

X = branch
Y = company
```

```
?- forall((part_of(Root,X,Y,Level),pname(A,X),pname(B,Y)),
   (write(Root  -  A/ X   -  B/ Y  -  Level), nl)).

BK   -   audit/  [BK:1, audit:2]  -                     BK/  [BK:1]                    -  2
BK   -   product/   [BK:1, product:2]   -               BK/  [BK:1]                    -  2
VL   -   sales/   [VL:1, sales:2]  -                    VL/   [VL:1]                   -  2
VL   -   product/   [VL:1, product:2]  -                VL/   [VL:1]                   -  2
VL   -   b_vp/[VL:1, product:2, sales:2, b_vp:3]  -   VL/   [VL:1]                   -  3
VL   -   b_vp/[VL:1, product:2, sales:2, b_vp:3]  -   sales/ [VL:1, sales:2]         -  3
VL   -   b_vp/[VL:1, product:2, sales:2, b_vp:3]  -   product/ [VL:1, product:2]  -  3
```

Meet and join operations are provided over the part_of lattice, being used in predicates part_up and part_down (on full identifiers), or p_up and p_down (on the atomic names). For example:

```
?- part_up(['VL':1,product:2],['VL':1,sales:2],M), pname(A,M).
M = [VL:1]
A = VL
```

```
?- part_down(['VL':1,product:2],['VL':1,sales:2],J), pname(A,J).
J = [VL:1, product:2, sales:2, b_vp:3]
A = b_vp
```

As a final example, consider a query involving several of the notions discussed in this paper: find all available information on someone associated with a company that has an audit department:

```
?- r_find(I,[name:N,denomination:D]), p_of(D,audit,D,_),inst_of(N,C), inst_frame(N,F).

I = works([Hugo, BK])
N = Hugo
D = BK
C = solicitor
F = [area:law, fee:nominal, works/1:BK]
```

8. Concluding remarks

We have presented informally, and without trying to cover all details, a method to deal with information systems where entity classes form lattice structures under is_a and under part_of links, and are connected by binary relationships proper of the application domain. The main point of the presentation was how to classify entity instances on the basis of the currently available information as organized in frames, by applying lattice operators. Additionally, the reasoning underlining the classification process turned out to be adequate for searching for instances that satisfy, exactly or approximately, an ample variety of structured requirements.

At the present stage, the method has a number of limitations, among which we mention:

- Attributes of entities and relationships are allowed to be multi-valued, but our working examples have only involved single-valued attributes thus far.
- The harder problems with is_a with multiple inheritance were largely avoided at the cost of forcing a somewhat restrictive design strategy, whereby separate attributes cannot have the same name, and, being single-valued, have no chance to lead to the collisions that cause difficulties in multi-way inheritance.
- Contrary to the more usual object-oriented discipline, we do not rely on any sort of special object-identifier (*oid*) to establish individual existence of what turns out to be a same object instance at possibly several levels of specialization. With the adoption of unique object-identifiers we would not have to require, as we do, that instances of classes linked by is_a be identified by the same privileged attribute (name for person, denomination for company, in our example).
- Only binary relationships are allowed, which can be 1-1, 1-n or n-n. The bracket notation used facilitates the extension to more than binary relationships, but many problems will have to be treated before we are able to introduce them.

- The binary relationships are not broken into specialization/generalization classes, as are the entities through is_a links. So we did not have to face the problem of classifying a relationship on the basis of frames, which would pose a number of problems in view of our classification method. Indeed, the method would fail in the presence of more than one occurrence of the same attribute name, as happens with the (improperly) called 'unary' relationships, such as, in our example, the gives_legal_assistance_to relationship, in which name occurs twice as identifying attribute for the two participants (of classes solicitor and employee, both of which are specializations of person).
- The work with part_of hierarchies has just started and its integration with the other features has not been fully considered.

The current prototype implementation supports all features described in this paper. In particular, meet and join are implemented as *generic* lattice operators, in that they can work over other lattice structures besides those of information systems, such as the natural numbers under divisibility, sets under inclusion, etc., provided that their elements can be represented by frames of properties, and separate rules are specified to indicate, for each domain, how the values of properties in common are to be treated by meet and by join. So, for natural numbers, an op_meet rule determines the choice of the lowest values (smallest exponents of prime factors held in common), whereas an op_join rule chooses the highest ones. The non-common properties, on the other hand, do not require any special provision, since, for any lattice-structured domain, they are always treated in the same way: left out by meet and kept together by join.

The prototype runs in *Arity Prolog* and in *SWI Prolog*. Several algorithms involve inherently time-consuming mutually recursive calls, especially classification, recalling that frames are allowed to refer to lattice-structured sub-domains as attribute values, and to names of entity classes as relationship participants, to be in turn checked by classification. In addition, as remarked, the evaluation of virtual attributes is another inevitably expensive task.

And yet, having made an option for generality in this first non-optimized version, we expect that the prototype should be of considerable help towards the continuing development of the method.

References

[BC] C. Batini, S. Ceri and S. Navathe – *Conceptual Design – an Entity-Relationship Approach*. Redwood: Benjamin Cummings, 1992.

[Fu] A. L. Furtado – "Uma introdução ao uso de programação em lógica para modelagem conceitual" – monografia MCC42/04 – Dept. Informática da PUC-Rio, 2004.

[MB] S. MacLane and G. Birkhoff – *Algebra* – MacMillan (1967).

[Kn] K. Knight – "Unification: a multidisciplinary survey" - *ACM Computing Surveys*, 21, I (1989) 93-124.

[P1] G.D. Plotkin – "A note on inductive generalization" - in *Machine Intelligence* 5 – B. Meltzer and D. Michie (eds.) – Edinburgh University Press (1970) 153-163.

[Re] J.C. Reynolds – "Transformational systems and the algebraic structure of atomic formulas" - in *Machine Intelligence* 5 – B. Meltzer and D. Michie (cds.) – Edinburgh University Press (1970) 135-152.

[RG] M. Rosemann, P. Green – *Business Systems Analysis with Ontologies* - Idea Group Publishing, 2005.

[SC] R. C. Schank and K. M. Colby – *Computer models of thought and language* – W. H. Freeman and Company (1973).

[SN] C. S. dos Santos, E. J. Neuhold e A. L. Furtado – "A data type approach to the entity-relationship model" – *International Conference on the Entity-Relationship Approach to Systems Analysis and Design* – 1979.

[WM]A. Walker, M. McCord, J. F. Sowa and W. G. Wilson – *Knowledge systems and Prolog* – Addison-Wesley (1987).

Appendix I

Schema Specification

```
% EXAMPLE - ER SCHEMA


% Entities and their properties

entity(person,name).
attribute(person,gender).

entity(employee,name).
attribute(employee,salary).
attribute(employee,office_phone).
is_a(employee,person).

entity(car_employee,name).
rel_ent(car_employee,works/1,car_company).
is_a(car_employee,employee).

entity(student,name).
attribute(student,area).
attribute(student,scholarship).
attribute(student,room_phone).
attribute(student,fee).
attribute(student,lodging).
attribute(student,restaurant).
virtual_attribute(student,monthly_pay,r3).
r3([fee:V1,lodging:V2,restaurant:V3],[monthly_pay:V4]) :-
  sum([V1,V2,V3],V4), not (V4 = 0).
is_a(student,person).

entity(tech_student,name).
attr_val(tech_student,area,technical_sciences).
is_a(tech_student,student).

entity(law_student,name).
attr_val(law_student,area,law).
is_a(law_student,student).

entity(visiting_law_student,name).
attr_val(visiting_law_student,fee,nominal).
is_a(visiting_law_student,law_student).

entity(trainee,name).
attribute(trainee,badge_numb).
domain(trainee,badge_numb,term).
virtual_attribute(trainee,contact,r1).
r1([room_phone:P1,office_phone:P2],[contact:[room-P1,office-P2]]) :-
  not var(P1),not var(P2).
virtual_attribute(trainee,revenue,r2).
r2([salary:S1,scholarship:S2],
   [revenue:S3,salary:S1,scholarship:S2]) :-
  not var(S1), not var(S2),S3 is S1 + S2.
is_a(trainee,employee).
is_a(trainee,student).

entity(solicitor,name).
attr_val(solicitor,badge_numb,jur-X).
rel_ent(solicitor,gives_legal_assistance/2,undef).
is_a(solicitor,trainee).
is_a(solicitor,law_student).

entity(car_attendant,name).
is_a(car_attendant,trainee).
is_a(car_attendant,car_employee).

entity(company,denomination).
attribute(company,headquarters).
virtual_attribute(company,size,r4).
r4([works/2:Es,size:_],[works/2:Es,size:M]) :-
  not var(Es),
  count(Es,M).
virtual_attribute(company,situation,r5).
r5([works/2:Es,situation:K],
   [works/2:Es,situation:K]) :-
  not var(Es),
  xsetof(I,(on(I,Es),status([I,C],permanent)),S),
  count(S,N),
  (N >= 2, K = operational;
   N < 2, K = inactive).

entity(car_company,denomination).
attr_val(car_company,headquarters,'Carfax').
is_a(car_company,company).

entity(active_company,denomination).
attr_val(active_company,situation,operational).
is_a(active_company,company).

% Structured entities and their properties

entity(dept,d_id).
attribute(dept,headcount).
part_of(dept,company).

entity(branch,b_id).
attribute(branch,district).
part_of(branch,dept).

% Relationships and their properties

relationship(works,[employee,company]).
attribute(works,start_year).
attribute(works,status).
attribute(works,today_stamp).
r_attr_val(works,today_stamp,T) :-
  date(date(Y,M,D)),
  T = (D/M/Y).
virtual_attribute(works,years_of_work,r6).
r6([start_year:Ys,today_stamp:(_/_/Yt)],
   [start_year:Ys,years_of_work:Yw]) :-
   not var(Ys),!,
   (not var(Yt); var(Yt),date(date(Yt,_,_))),
   Yw is Yt - Ys.
r6([today_stamp:S],[]).

relationship(gives_legal_assistance, [solicitor,employee]).
gives_legal_assistance([E1,E2]) :-
  works([E1,C]), works([E2,C]),
  area(E1,law), not area(E2,law).

% A general property definition

attribute(E,A) :- virtual_attribute(E,A,_).

% hierarchy of the study area domain

% major classification

is_a(area,social_sciences,study_area).
is_a(area,human_sciences,study_area).
is_a(area,technical_sciences,study_area).

% social

is_a(area,law,social_sciences).
is_a(area,management,social_sciences).
is_a(area,economics,social_sciences).

% humanities

is_a(area,philology,human_sciences).
is_a(area,design,human_sciences).
is_a(area,philosophy,human_sciences).

% technology

is_a(area,informatics,technical_sciences).
is_a(area,engineering,technical_sciences).
is_a(area,chemistry,technical_sciences).
is_a(area,physics,technical_sciences).

% multi-disciplinary

is_a(area,computational_linguistics,philology).
is_a(area,computational_linguistics,informatics).
is_a(area,digital_entertainment,design).
is_a(area,digital_entertainment,informatics).
is_a(area,architecture,design).
is_a(area,architecture,engineering).
```

Appendix II

A State of the Information System

```
% EXAMPLE - STATE

salary('Hercule',100).
scholarship('Hercule',30).
fee('Hercule',50).
room_phone('Hercule',123).
office_phone('Hercule',456).

area('Jonathan',law).
salary('Jonathan',100).
works(['Jonathan','VL']).
start_year(['Jonathan','VL'],2001).
status(['Jonathan','VL'],permanent).
badge_numb('Jonathan',jur-134).

headquarters('VL','Carfax').

area('Hugo',law).
fee('Hugo',nominal).
works(['Hugo','BK']).
status(['Hugo','BK'],temporary).

headquarters('BK','Bykeville').

badge_numb('Lucy',inf-123).
works(['Lucy','VL']).
status(['Lucy','VL'],temporary).

works(['Mina','VL']).
status(['Mina','VL'],permanent).

area('Bertillon',engineering).
fee('Bertillon',45).

fee('Dupin',100).
lodging('Dupin',25).
restaurant('Dupin',18).

dept(['VL':1,product:2]).
headcount(['VL':1,product:2],2).

dept(['VL':1,sales:2]).
headcount(['VL':1,sales:2],1).

branch(['VL':1,product:2,sales:2,b_vp:3]).
district(['VL':1,product:2,sales:2,b_vp:3],'Trnsvn').

dept(['BK':1,product:2]).
headcount(['BK':1,product:2],1).

dept(['BK':1,audit:2]).
headcount(['BK':1,audit:2],1).
```