Subtyping: conceptual and logical issues

by Dr. Terry Halpin, BSc, DipEd, BA, MLitStud, PhD
Director of Database Strategy, Visio Corporation

This paper first appeared in vol. 23, no. 6 of Database Newsletter and is reproduced by permission.

Subtyping is an important feature of semantic approaches to conceptual schema design and, more recently, object-oriented database design. However the relational model does not directly support subtyping, and CASE tools for mapping conceptual to relational schemas typically provide only very weak support for mapping subtypes. This paper surveys some of the main issues related to conceptual specification and relational mapping of subtypes, and indicates how Object-Role Modeling solves the associated problems.

Introduction

To enhance communication between modeler and domain expert, and to facilitate later changes in the application and/or the implementation DBMS, an information system should be specified at the conceptual level before it is mapped to some logical data model. At the conceptual level, information is expressed as elementary facts together with various business rules (constraints and derivation rules) which restrict allowable states and transitions on the database. How these facts and rules are grouped into structures depends on the target logical model (e.g. relational, network, hierarchic, object-oriented) and hence is not fundamentally a conceptual issue.

Like relational schemas, so-called "object-oriented" (OO) schemas provide little direct support for declarative business rules, although they do provide some rudimentary support for subtyping. It is well known that some classic OO approaches deriving from OO-programming are crippled, by failing to allow *migration between subtypes* (e.g. a contract employee is promoted to permanent employee) and failing to allow *multiple inheritance* (e.g. a student tutor is both an employee and a student).

In his NTH (the Newsletter on Type Hierarchies) framework, Ross [4-6] sets out several criteria for a truly declarative approach to subtyping. These criteria are fundamentally in agreement with semantic approaches to conceptual modeling such as *Object-Role Modeling (ORM)* and the more elegant versions of *Enhanced Entity-Relationship (EER)* Modeling. Both ORM and EER are at heart data modeling methods, that focus on *static constraints and rules* (these apply to each individual database state that occurs in the rule's lifetime). In the area of *dynamic constraints*, Ross provides some extensions (see his rules for subtype migration and initialization [6]).

By applying abstraction techniques to identify major object types on which operations may be defined [1], it is possible to extend ORM and EER to provide object-oriented views

which encapsulate operations (often misnamed "methods") with the data. However in this article we focus on the data perspective of subtyping, with the emphasis on conceptual specification and later mapping to the relational level. In particular, we address the following questions within the context of data modeling:

- Why subtype at all?
- How should subtypes be displayed?
- When should subtypes be used?
- What subtype constraints should be declared?
- How should subtypes map to a relational DBMS?

These questions are discussed briefly and rather informally. A detailed discussion of many of the issues may be found in [1, 2] while a formal treatment is presented in [3].

Why subtype at all?

Three important reasons for using subtypes in information systems modeling are:

1. To assert typing constraints
2. To assert classification schemes (taxonomy)
3. To encourage re-use of model components

Reason (1) is the most important. Here we declare that some information is recorded only for a specific subtype. For example, votes should be recorded only for permanent residents. Subtypes may also be used to classify an object type according to some criteria of interest. For example, we might classify people according to their sex, and introducing MalePerson and FemalePerson subtypes is one way of showing this. Thirdly, if a new subtype is introduced to a model, it automatically inherits the properties of its supertypes, and hence only its specific roles need to be declared; apart from this reduction in code duplication, the more generic supertypes are likely to find re-use in other applications.

How should subtypes be displayed?

An *object type* (e.g. Person or Car) may be thought of as the set of all its possible populations, and is usually depicted as a named loop. In ORM this loop is either an ellipse or a frame (rounded corners), whereas in ER it is typically a rectangle or a frame. The choice of loop shape is not really important. At any time, an object type will be populated by a set of objects. During the lifetime of an application, the state or population of the type changes from one set to another (sets themselves are unchanging). Given two object types, *A* and *B*, we say that *B* is a *subtype* of *A* if, for each state of the database, the population of *B* must be included in the population of *A*.

In information systems work, the only subtypes of interest are *proper subtypes*. We say that *B* is a proper subtype of *A* if (i) *B* is a subtype of *A* and (ii) there is a possible state in which the population of *A* includes some object that is not in *B*. For example, Woman is a proper subtype of Person. We could have a state in which all the people are women, and another in which there are also some men, but we never have a state in which a woman is not also a person. From now on, we use "subtype" as short for "proper subtype".

The two main ways in which subtyping is depicted graphically are:

1. Euler diagrams
2. Directed Acyclic Graphs

Euler diagrams depict relationships between subtypes spatially (unlike Venn diagrams, with which they are sometimes confused). For example, Figure 1 depicts the following information: *B*, *C* and *D* are subtypes of *A*, and *E* is a subtype of both *C* and *D*. Moreover, *B overlaps* with *C* (i.e. they may have a common instance) and *C* overlaps with *D*, but *B* and *D* are *mutually exclusive* (cannot have a common instance). For example: *A* = Person; *B* = Asian; *C* = Consultant; *D* = American; *E* = TexanConsultant.

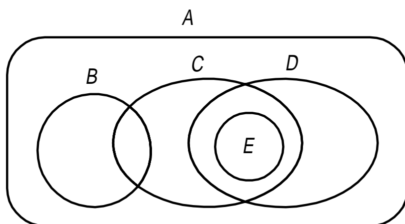

FIGURE 1 An Euler diagram.

Euler diagrams provide intuitive displays for simple cases but are too cumbersome for the complex subtype patterns often found in real applications, where an object type might have a large number of subtypes many of which overlap. Moreover, individual subtypes may have many specific details recorded for them, and there is simply no room to attach these details if the subtype nodes are crowded together inside their supertype nodes.

For such reasons, Euler diagrams are eschewed for non-trivial subtyping. Instead *directed acyclic graphs* (DAGs) are often used. A directed graph is simply a graph of nodes with directed connections, and acyclic means there are no cycles (here a consequence of proper subtyping). The subtype pattern in Figure 1 is represented in DAG form in Figure 2. Here an arrow from one node to another shows that the first is a subtype of the second. Since subtypehood is transitive, indirect connections are omitted (e.g. since *E* is a subtype of *C*, and *C* is a subtype of *A*, it follows that *E* is a subtype of *A*, so there is no need to display this implied connection). Instead of using arrowheads, some notations assume the direction is always upwards; however this makes it very difficult to layout detailed schemas.

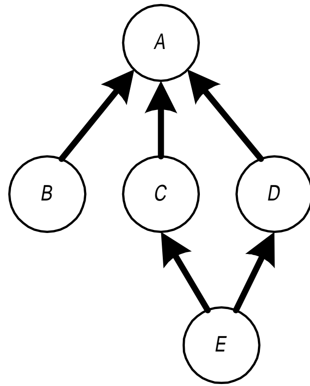

FIGURE 2 A directed acyclic graph.

DAGs are less intuitive than Euler diagrams. For example, in Figure 2, *B* is shown "outside" *A* even though every object instance within *B* must also be contained in *A,* and information about subtype overlapping is lost. This is a consequence of depicting subtype connections by arrows rather than spatial containment. However this disadvantage is more than offset by the fact that DAGs may be used to conveniently represent subtype patterns of arbitrary complexity, while still allowing plenty of space around each node for details to be attached.

Note that Figure 2 depicts only the subtype connections. Figure 1 contains the additional information that *B* overlaps with *C*, *C* overlaps with *D*, and *B* and *D* are mutually exclusive. Hence if DAGs are used, some other means must be used to convey whether or not subtypes are mutually exclusive. One way of doing this in ORM is to attach an exclusion symbol "⊗" via dotted lines to the relevant subtype links. The absence of such a symbol indicates that the populations of the types may overlap. For example, with this additional symbol, Figure 3 now conveys the exclusion/overlap information in Figure 1.

Traditionally, Euler diagrams are viewed existentially (i.e. something exists in each region). From this viewpoint, Figure 1 makes further claims about totality. For example, *A* is a proper supertype of the union of *B*, *C* and *D*; and *E* is a proper subtype of the intersection of *C* and *D*. However Euler diagrams may also be viewed hypothetically (no existential claims are made). From the hypothetical viewpoint, Figures 1 and 3 are equivalent. With DAGs, further symbols (e.g. "/") may be used to indicate totality constraints. As we see later, a proper treatment of subtyping entails that the relevant exclusion and totality constraints are typically implied.

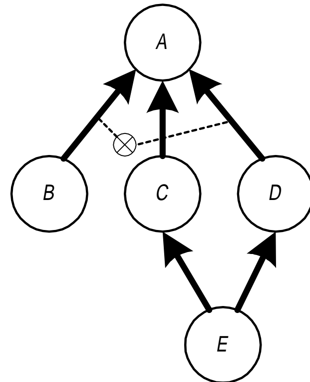

Figure 3 B and D are mutually exclusive.

When should subtypes be used?

Some modelers like to use subtypes purely for taxonomic reasons. For example, suppose we wish to classify employees according to their marital status. We might set this out using subtypes as shown in Figure 4. Here the exclusion "–" and totality "/" symbols are superimposed to indicate that both apply. In other words we have a *partition* (the subtypes are exclusive, and their union equals the supertype).

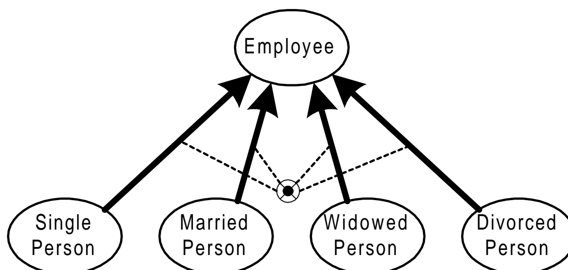

FIGURE 4 A partition.

Here we have four possible marital states. As the number of states grows, subtype displays consume even more space. Imagine using this method to display employee rank in a company that has ten or more ranks. Usually, taxonomic information is more efficiently conveyed via some classifying predicate. For example, the taxonomy in Figure 4 may be displayed in ORM as set out in Figure 5. For the reader unfamiliar with Object-Role Modeling, some brief points about the method are now given.

ORM is a conceptual modeling method that views the application world simply in terms of objects that play roles, either individually (e.g. smokes) or within a relationship (e.g. in Figure 5 the object type Employee plays the role of having, and the object type MaritalStatus plays the role of being held). Unlike ER, no use is made of attributes, so there is no need to agonize over whether some feature is to be modeled an attribute or not.

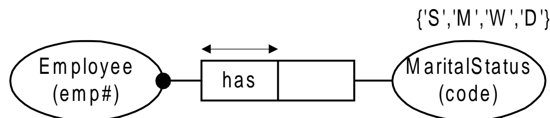

FIGURE 5 The same partition.

ORM allows facts to be verbalized naturally, and its role-based notation allows fact types of any length to be populated with instances for validation with the domain expert as well as facilitating the declaration of many constraints and rules. Since its object types are conceptual domains, ORM diagrams reveal the semantic glue that binds an application together. Once an ORM model has been developed, various abstraction mechanisms may be applied to hide detail as desired. One such mechanism generates an ER view. Because of its obvious advantages over ER, we use ORM for the rest of our discussion. However the reader should have little trouble in relating the discussion to ER as well. Perhaps the most well known version of ORM is NIAM, which was developed in Europe in the early 70s. The version of ORM discussed here is FORM (Formal ORM) which is supported by the InfoModeler CASE tool from Asymetrix. A detailed treatment of ORM is found in [1].

In Figure 5 the named ellipses depict object types, with their reference schemes abbreviated in parenthesis. Employees are identified by their employee number (emp#), while each marital status is identified by a code: the possible codes (e.g. 'S') are listed in braces. Such a list is called a *value constraint*, since it indicates which values may be used to reference objects of that type.

If we also want to record full names (e.g. 'Single') for marital status this may be added as a 1:1 fact type (e.g. MaritalStatus has MaritalStatusName). Notice how the approach facilitates schema evolution. Here we simply added a fact type; if in ER we had modeled marital code as an attribute of employee, and then later decided to store the corresponding names, a more drastic change is required.

In ORM, each role is shown as a box attached to the object type that plays it. A logical predicate is depicted as a named sequence of one or more contiguous roles, with the name written in or beside the first role. If desired, alternative predicate readings may be provided to allow facts to be read in other directions.

The arrow-tipped bar over the left role in Figure 5 is a *uniqueness constraint*, which may be verbalized as: each Employee has at most one MaritalStatus. If the fact type is populated with instances, entries in the left column will be unique, but not so for the right column. Read from left-to-right, this is a many:1 fact type. The black dot on Employee is a *mandatory role constraint,* indicating that this role must be played by each employee referenced in the database population. In words, this constraint is: each Employee has some MaritalStatus. With InfoModeler, facts and constraints may be entered graphically or textually, with automatic conversion between the two representations.

Let us assume we are using ORM or at least a version of ER that supports value constraints. Since taxonomies are more compactly depicted with predicates, we argue against introducing subtypes merely to display a taxonomy (compare Figures 4 and 5). Just as for other object types, we recommend the following design guideline: *don't introduce a subtype unless there's something specific you want to say about it*. There are two main cases in which this might arise:

1. A role is played only by that subtype
2. A constraint or operation is different for that subtype

The first case is by far the most important, and we illustrate it by means of example. Suppose we are modeling a hospital system for which output reports like the sample shown in Table 1 are required. Here a "?" denotes a simple null value (value is unknown), whereas "–" is a special null value (an actual value cannot exist because of some other entry on this row: this corresponds to Codd's I-mark).

TABLE 1 A sample report.

Patient Nr	*Name*	*Sex*	*Phone*	*Prostate status*	*Pregnancies*	*Ante-natal visits*
101	Adams A	M	2052061	OK	–	–
102	Blossom F	F	3652999	–	5	6/20/90 7/15/90 2/15/95
103	Jones E	F	?	–	0	?
104	King P	M	?	benign enlargement	–	–
105	Smith J	M	2057654	?	–	–

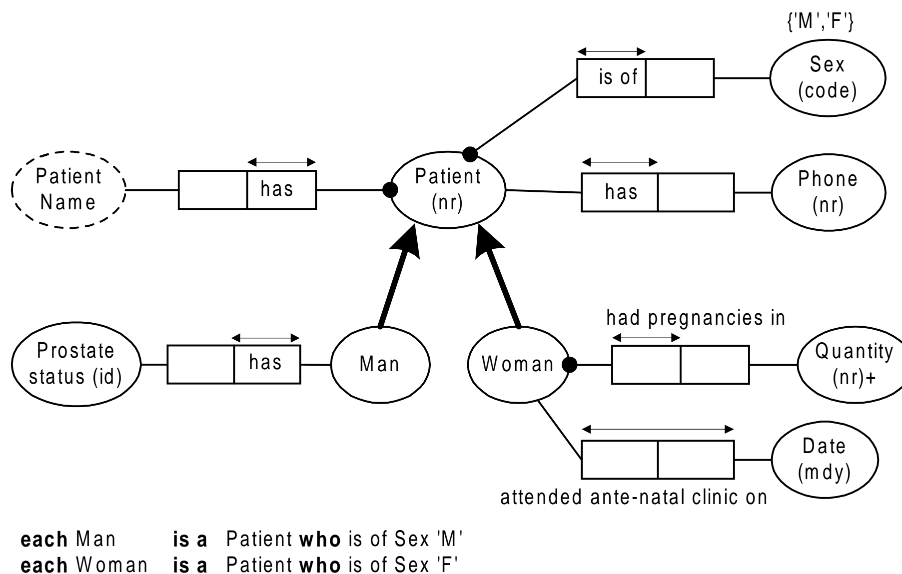

FIGURE 6 An ORM schema for Table 1.

An ORM schema for this report is shown in Figure 6. The broken ellipse for PatientName indicates this is a *value type* (in this case a character string) and hence needs no reference scheme. The solid ellipses denote *entity types*. The "+" on Quantity indicates that the values which refer to Quantity are actual numbers and hence may be added etc. In contrast, patient numbers and phone numbers are not to be used arithmetically.

Notice how the simple null values are modeled in terms of *optional roles*. If a role played by a non-leaf node has no mandatory role dot it is optional. For example, it is optional whether a patient has a phone. The inapplicable-nulls however are modeled using subtypes. It is part of the modeler's job to extract the rules that determine when a value is applicable here. If the modeler is familiar with the application, a reasonable guess may be made. But since infinitely many rules satisfy any finite data set, there is in principle no way of automatically deriving the correct rule with certainty just from the sample data. Fortunately, the domain expert knows the rule so we need only to check the rule with this person.

Let us agree that the rules in this case are: prostate status is recorded only for men; number of pregnancies and dates of ante-natal visits are recorded only for women. This requires us to introduce subtypes for men and women. In general, *if an optional role is played only by some well-defined subtype we should introduce the subtype and attach its specific roles*. In doing this, we should provide *subtype definitions* which enable membership in the subtype to be determined by conditions on roles played by its supertype(s).

Figure 6 includes appropriate subtype definitions in the FORML language used by InfoModeler. Notice that these definitions are not simply "is a" connections: they also include the condition that determines membership in that subtype. Most subtype definitions are fairly simple like this; but in general the schema path and conditions involved in the definition may be long and complex. For example, consider defining a subtype for patients who contracted in the 1980s any disease known to have originated from Africa. An expressive formal language such as FORML is required for such cases.

In our example, the introduction of the subtypes Man and Woman as special cases of Patient is referred to as *specialization*. The inverse process whereby a common supertype is introduced is known as *generalization*. For example, we might start with just Man and Woman and then decide to introduce Patient because of their common information (name, sex, phone number). Although each process arrives at a subtype graph, with specialization the subtypes usually inherit the identification scheme of their (top) supertype, while with generalization a new identification scheme is often introduced for the supertype to provide global reference. For further details on context-dependent reference and related issues, see [1, 3].

Sometimes a constraint or operation defined for an object type takes a more specific form for some well-defined cases. In this case, subtypes might be introduced merely to facilitate declaration of the more specific versions of the constraints or operations. Ignoring non-monotonic approaches to default reasoning, we should ensure that the *specialized rules* (on the subtypes) *are merely stronger versions of the general rules* (on the supertype(s)). This implies that the rules should be consistent.

Consider the schema fragment in Figure 7. In general, employees have use of zero or more company cars. Recent employees have use of at most one company car. Managers have use of at least one and possibly two company cars (the "1-2" is a frequency constraint). Let us define a RecentManager to mean a manager who is a recent employee (excluding newly appointed managers who are a long-time employees). In this case, RecentManager inherits both the constraints of its direct supertypes, so each recent

manager has use of exactly one company car. Notice how the constraints are simply strengthened as we move down to more specialized cases. An optional role may become mandatory, and the frequency of a role may become more constricted, but we may not do the reverse.

In displaying the more specialized rules, broken lines were used to indicate that the car-usage predicate was repeated. To simplify layout, an object type (e.g. CompanyCar) may also be duplicated using double ellipses to indicate repetition. To reduce clutter on a schema diagram, display of such repeated predicates may be toggled off.

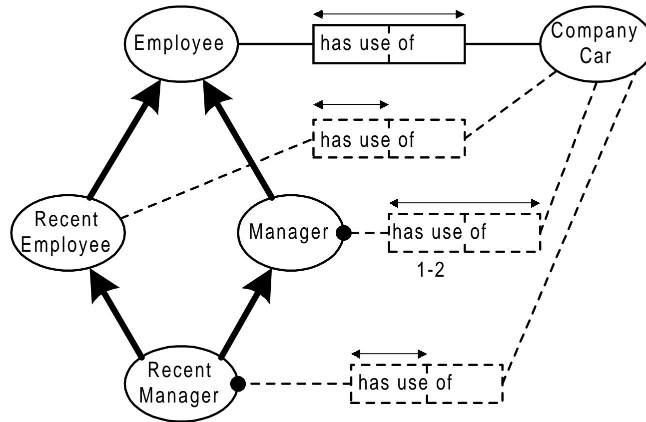

FIGURE 7 Constraint strengthening.

When the data model is augmented by binding operations to some of its object types, these operations may be specialized as well. When constraints and operations take on "different shapes" in this way, this is often referred to as "polymorphism". This term is sometimes used in a different sense (e.g. [3]).

What subtype constraints should be declared?

Subtype constraints may be static or dynamic. Dynamic constraints specify restrictions on initialization of and migration between subtypes. An extensive discussion of these is provided by Ross [5]. Instead of using subtypes, some of these cases may be specified using a transition table or graph. For example, instead of saying a person cannot migrate from a married person to a single person, we merely exclude the transition from the marital status 'M' to 'S'.

Static constraints on subtypes are more fundamental. The only proper way to treat these is to provide *formal subtype definitions*. This point is rarely recognized in practice. It is a common misconception that the declaration is complete once subtype links (is-a connections) and exclusion and totality constraints are declared. To see that this is nonsense, consider the populated schema fragment shown in Figure 8 (reference schemes are omitted for simplicity). The population satisfies all the declared constraints, but there is still something wrong. Can you spot the problem?

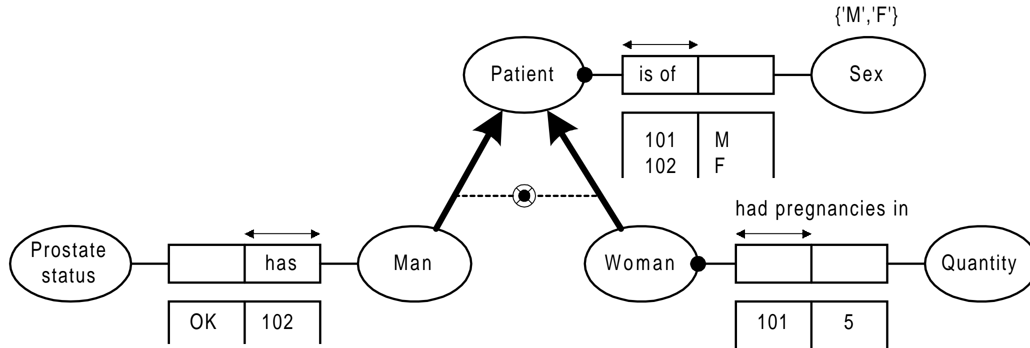

FIGURE 8 What is wrong with this?

It wasn't hard to spot, was it? Prostate status has been recorded for women and pregnancies for men! The only way to avoid such errors is to declare the subtype definitions (see Figure 6) and enforce them. Moreover, the partition constraint between the subtypes is implied by these definitions in conjunction with the constraints on the sex fact type. Since each person has one of two sexes, the subtyping is exhaustive; and since each person has only one sex, the subtyping is exclusive. For this reason, display of implied totality and exclusion constraints may be toggled off as desired to avoid clutter.

How should subtypes map to a relational DBMS?

Subtypes are not directly supported in the relational model, though Codd's RM/T proposal provides limited support, and the proposed SQL3 standard includes primitive support in the form of sub-tables. Fortunately there are ways of mapping subtypes to even current relational systems. We now sketch the basic ideas, limiting our attention to static aspects.

There are two basic aspects to the mapping any conceptual schema: grouping fact types into tables; and mapping the associated business rules (constraints and derivation rules). Let's look at the grouping aspect first. Ignoring tuning by controlled denormalization, let us assume we want a redundancy-free, fully normalized relational schema. Since all the fact types in a correct ORM schema are elementary, this task is fairly trivial. Redundancy is nothing other than repetition of an elementary fact, and a relational table stores one or more elementary fact types. So to avoid redundancy we simply map each fact type to only one table, in such a way that its instances appear only once. To achieve this, fact types with compound uniqueness constraints map to separate tables, while fact types with functional roles (these have a simple uniqueness constraint) attached to the same object type are grouped into the same table, keyed on the object type's identifier. There are a few finer points to the grouping, but this gives the basic idea.

Now let us consider mapping of the fact types played by object types in a subtype graph. First, determine whether or not the subtypes inherit the primary identification scheme of their supertypes. If they do, there are three basic ways in which the fact types may be grouped into relational tables: absorption; separation; partition.

Absorption effectively absorbs the subtypes back into their top supertype before grouping. This reduces the number of tables and speeds up many queries and updates which otherwise would have involved a join., but generates nulls and complicates access to a single subtype. *Separation* groups functional fact types of each node into a separate table for that object type. This reduces nulls but requires joins for queries involving attributes from more than one subtype. The *partition* option is used only if the subtype graph forms a partition. In this case, common fact types on the supertype(s) are pushed down to the leaf subtypes, then a separate table is formed for each of these subtypes.

Combinations of these three approaches may also be used. When the primary identifier of a subtype is context-dependent rather than inherited, special care is required to map the subtype links. For further details on these issues, see section 8.4 of [1].

Assuming that a grouping decision for subtype mapping has been made, we now have the task of mapping the relevant subtype constraints. Fortunately this is relatively straightforward. We illustrate the basic ideas by mapping the conceptual schema in Figure 6. Using absorption, we obtain the relational schema shown in Figure 9. The functional fact types are absorbed into a single Patient table, while the many:many fact type about ante-natal visits is mapped to a separate table. For simplicity, the domains on which the attributes are based are omitted. Notice how the constraints are mapped. Keys are underlined, the value constraint on sex codes is displayed next to the relevant attribute, and optional columns are shown in square brackets. The dotted arrow denotes a subset constraint (in this case, a foreign key).

The mapping of the subtype constraints is captured in the three numbered qualifications. Qualification 1 declares that a non-null value for prostate exists only if the value of sex is ‘M’. Qualification 2 declares that a non-null value for pregnancies exists if the value of sex is ‘F’ (this captures the mandatory role) and only if the value of sex is ‘F’ (this captures the subtype constraint). Qualification 3 declares that each value in AnteNatalVisit.patientnr must be a value of Patient.patientnr for which the value of sex is ‘F’: for simplicity we omit the referential action to be taken on violation of this constraint.

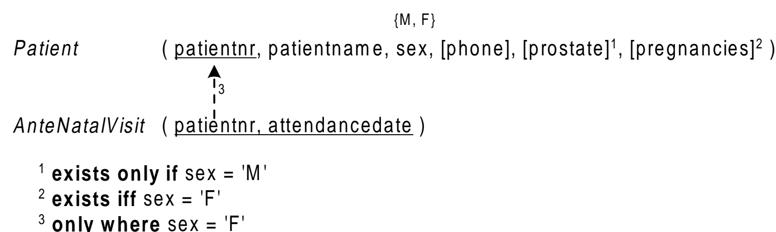

FIGURE 9 The relational schema mapped from Figure 6.

The generic notation in Figure 9 may be mapped to the language in the chosen DBMS. For example, in SQL-92 the first two qualifications map to the following constraints on the Patient table:

```
constraint prostate_recorded_only_for_males
        check (     prostate is null or sex = 'M' );
constraint pregnancies_recorded_iff_patient_is_female
        check (     pregnancies is null and sex <> 'F' or
                    pregnancies is not null and sex = 'F' )
```

Since the subtype constraint underlying qualification 3 is stronger than a subset constraint, the following assertion is used instead of declaring a foreign key clause:

```
create assertion each_antenatal_visit_is_by_a_female
        check ( not exists  ( select patientnr from AnteNatalVisit
                              except
                              select patientnr from Patient
                              where sex = 'F') )
```

Although included in the SQL92 standard, assertions are not yet supported in all SQL systems, so it may be necessary to generate alternate code for this constraint (e.g. as a triggered procedure). Since mapping large and complicated schemas is a lengthy and error-prone task, it is important to have a CASE tool that automates as much of the mapping as desired. The next release of InfoModeler should provide a good example of what is possible in this regard. The interesting and creative task is for the modeler to arrive at a clear, conceptual model of the application through communication with the subject matter experts. Although methods like ORM make this task easier, this initial step from the informal to the formal is one that can't be automated, so humans will remain indispensable in the modeling process. Just as well!

References

1. Halpin, T.A. 1995, *Conceptual Schema and Relational Database Design*, 2nd edn, Prentice Hall, Sydney, Australia.

2. Halpin, T.A., Harding, J. & Oh, C-H 1992, 'Automated support for subtyping', *Proc. Third Workshop on the Next Generation of CASE Tools*, eds B. Theodoulidis & A. Sutcliffe, Paris, France, pp. 151-61.

3. Halpin, T.A. & Proper, H.A. 1995, 'Subtyping and polymorphism in Object-Role Modelling', *Data and Knowledge Engineering*, North-Holland (to appear).

4. Ross, R.G. 1994, 'Prescriptions for subtyping in database design', *Data Base Newsletter*, vol. 22, no. 5, Database Research Group, Boston MA.

5. Ross, R.G. 1994, 'Representation schemes for type hierarchies, *Data Base Newsletter*, vol. 22, no. 6, Database Research Group, Boston MA.

6. Ross, R.G. 1995, 'Declarative rules for type hierarchies, *Data Base Newsletter*, vol. 23, no. 1, Database Research Group, Boston MA.

This paper is made available by Dr. Terry Halpin and is downloadable from www.orm.net.