Chapter 3

A Data Model for Bridging Heterogeneity

In Chapter 1, I discussed the need for a database technology that provides medical decision-support applications with automated access to clinical data in a manner that does not compromise the applications' portability across provider sites. I asserted that the technology must consist of an abstract query model and a general method to translate queries to a variety of clinical database implementations. In Chapter 2, I reviewed previous research related to the development of an abstract query model and a general query-translation method, and concluded that no existing database technology fully meets the needs discussed in Chapter 1, although several components of the solution are in place.

3.1 Components of the TransFER Technology

In this chapter and the next chapter, I describe a novel technology, TransFER, for supporting the data-access needs of medical applications across heterogeneous database systems. TransFER combines and builds on previous research results in semantic data modeling and query-transformation techniques and comprises several components. In Section 3.2, I introduce and formalize a semantic data model, called **FER**, that combines the features of the functional (F) and the entity-relationship (ER) models. I use the FER data model to represent the contents of different relational databases in a uniform way. In Section 3.3, I discuss a high-level query language for the FER model, called **ReFER**, that is based on the domain calculus. I use ReFER to represent queries against different relational databases in a uniform way. I posit that the FER data model and the ReFER query language can serve as the foundations of a standard query model for clinical medicine, the *FER query model*. In Chapter 4, I describe a formal

methodology for translating queries formulated in the abstract FER query model into queries that can be executed by disparate relational database implementations. The translation methodology uses a language that is based on the relational algebra to represent mappings between constructs of the FER query model and the corresponding constructs of an implemented query model. The mappings constitute part of an attribute grammar that formally specifies the translation of ReFER queries. The implementation of the attribute grammar performs translations of ReFER queries automatically.

3.2 The Functional–Entity–Relationship Data Model

Any abstract query model for clinical medicine must be based on an abstract model of data. The features of the data model are critical because they largely determine the kinds of information that can be represented conveniently, the semantic level at which queries can be formulated, and the range of implemented database environments to which queries can be exported. To address the shortcomings of existing data models with respect to these features, I have developed a data model that combines the modeling constructs of the functional model [Shipman 1981] and the extended entity–relationship model [Engels, et al. 1992]. The new data model, called FER, seeks to support concurrently the data-access needs and portability requirements of medical applications by meeting two important design criteria:

1. *Expressiveness*: The constructs of the data model should allow the schematic description of all relevant medical data in a clear and intuitive way. Although "intuitiveness" is a subjective criterion, it implies a correspondence between machine-readable descriptions of stored medical data and human cognitive models of medical information [Sowa 1984]. Operationally, an expressive data model allows the representation of familiar semantic concepts without the use of abstruse modeling constructs. Expressiveness is an important feature of a standard query model for clinical medicine, because it allows a large and diverse population of application developers to understand and use the query model without extensive training.

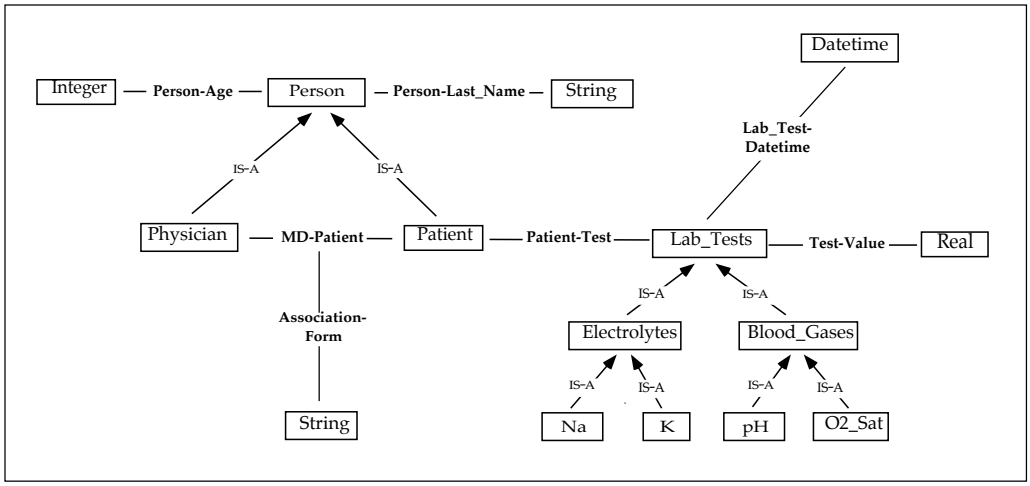
2. *Abstractness*: The constructs of the data model should be sufficiently general to allow a reference schema encoded in the data model to

subsume the heterogeneous relational database implementations that may be used to store and manipulate clinical data at provider sites. Site-specific database implementations may vary in terms of table and field names, schema structure, vocabulary standards, data semantics, and dialects of SQL. Abstractness is an important criterion because it allows the standard query model of clinical information to represent the contents of legacy clinical databases that have been implemented in significantly different ways due to local modeling decisions, product availability, and performance exigencies.

3.2.1 Constructs of the FER model

Consistent with the design goal of abstractness, the FER model includes a minimal set of high-level data-modeling constructs. In this section, I introduce the FER data model informally, referring to the hypothetical FER schema in Figures 3.1 and 3.2 where appropriate. I describe the FER model formally in Section 3.2.3.

The FER data model consists of five primary constructs and two derived constructs. The primary constructs of the model are **Entity Types**, **Entities**, **Relationship Types**, **Relationships**, and **IS-A Connections**.



1. **Entity Types**: Entity types denote *sets of objects* in the domain of discourse. An FER schema may describe entity types by specifying the associations that the members of those types may have with members of other entity types. For example, the Lab_Tests entity type in the schema of Figure 3.1 denotes the set of clinical laboratory tests. The schema specifies that members of the entity type Lab_Tests may be associated with members of the entity type Real via the Lab-Value association, and with members of the entity type Patients via the Patient-Lab association. In addition to the *user-defined entity types* defined for a particular FER schema (such as Person, Patient, and Lab_Tests in Figure 3.1) there are four *built-in entity types*: String, Integer, Real, and Datetime. The member objects of the built-in entity types are infinite and immutable, whereas the member objects of the user-defined entity type are finite and determined by the *state* of a database.

2. **Entities**: Entities denote *individual objects* in the domain of discourse. The FER model adopts an *object-oriented* perspective of entities, so that entities are characterized completely by their unique identifiers. This property implies that identifiers distinguish entities that may be identical in terms of the other entities with which they are associated. Examples of entities consistent with the schema of Figure 3.1 include the physician Jonas Salk, the lab test Serum_Sodium_#135579, and the integer 7.

3. **Relationship Types**: Relationship types denote and describe *sets of binary associations* between members of entity types (that is, between entities). A relationship type describes a set of associations by specifying the pair of entity types to which the associated entities belong. For example, the FER relationship type Lab-Value in Figure 3.1 describes a set of associations between Lab_Tests entities and Real entities. The relationship type of the FER model differs from the relationship construct of most ER models in two ways. First, FER relationship types may themselves participate in other FER relationship types. For example, in Figure 3.1, MD-Patient denotes a relationship type that associates members of the Patient entity type with members of the Physician entity type; MD-Patient also itself participates in a

relationship type named "Association-Form" with the String entity type. The semantics of Association-Form are that the association between a physician and a patient constitutes a domain *object* representing a physician-patient relationship (in the colloquial meaning of the term "relationship"); in each such relationship, the physician may assume the role of an "attending" physician or a "consulting" physician. Because a physician may be an attending physician for one patient and a consulting physician for another, the association form must be specific to a physician-patient relationship rather than to a physician. Allowing relationship types to participate in relationship types in this way enhances the expressiveness of the data model. The other way in which FER relationship types are distinct from those of the ER model is that the FER relationship type construct subsumes the relationship construct *and* the attribute construct of the ER model. In Figure 3.1, for example, the association between a Lab_Tests entity and the value of that lab test (an instance of Real) is modeled as a relationship type, rather than as an integer-valued attribute. It is likely that the value of a lab test would be modeled as an attribute in an ER schema. The FER model has no separate construct for modeling attributes; the model provides only the relationship type as a means for modeling both relationships and attributes.

4. **Relationships**: Relationships denote instances of associations between entities. Relationships also are characterized completely by a unique identifier and distinguished from each other on the basis of the identifier alone. Examples of relationships consistent with the schema of Figure 3.1 include the Person-Last_Name relationship <SSN138-58-4432,"Connors"> and the Lab-Value relationship <Serum_Sodium_#135579, 142.0>.

5. **IS-A Connections**: IS-A connections relate pairs of entity types and denote the semantics of generalization and specialization. In the FER model, IS-A semantics imply set subsumption and relationship inheritance. For example, in the schema of Figure 3.1, the IS-A connection between the Lab_Tests entity type and the Electrolytes entity type implies that each member of Electrolytes is also a member of

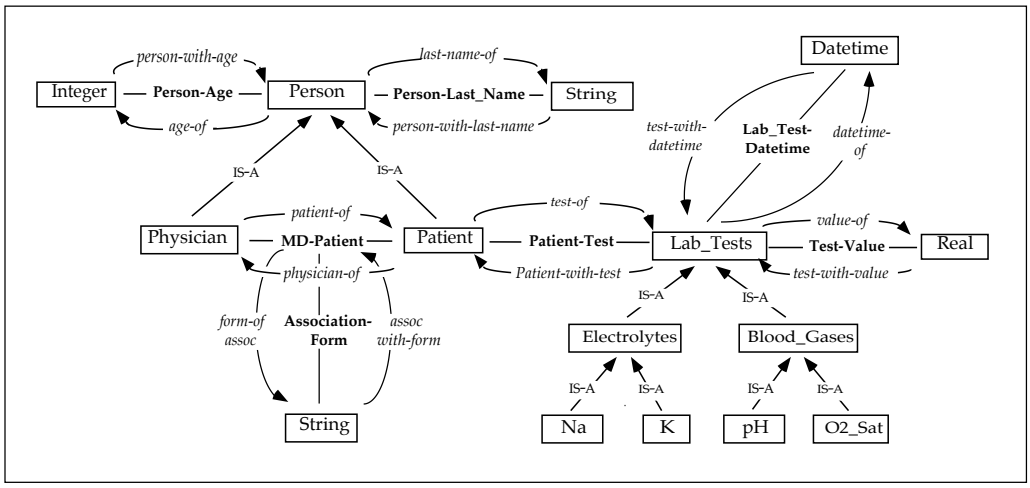
Lab_Tests and that the entity type Electrolytes inherits all relationship types in which Lab_Tests participates (specifically, Patient-Lab and Lab-Value).

The derived constructs of the FER model are **Relationship Functions** and **Relationship Predicates**.

1. **Relationship Functions**: Relationship functions are operational constructs for mapping across FER relationship types. Whereas relationship types represent static bi-directional associations between members of entity types, Relationship Functions provide operational unidirectional mappings from the members of one entity type to the set of associated members of another entity type. Each binary relationship type in an FER schema defines a pair of relationship functions. Figure 3.2 shows the schema of Figure 3.1 with all relationship functions displayed. For example, the relationship type Patient-Lab defines the pair of relationship functions:

   lab-of: Patient → Lab_Tests
   patient-with-lab: Lab_Tests → Patient.

   Subtypes specified by the IS-A connection inherit each of the relationship functions of their supertypes. For example, the relationship function "value-of" may be applied to a member of the entity type Electrolytes because Electrolytes is a subtype of the entity type Lab_Tests.


Integer
Person-Age
Person
Person-Last_Name
String
person-with-age
age-of
last-name-of
person-with-last-name
IS-A
IS-A
Physician
MD-Patient
Patient
Patient-Test
Lab_Tests
Test-Value
Real
patient-of
physician-of
test-of
Patient-with-test
value-of
test-with-value
Datetime
test-with-datetime
Lab_Test-Datetime
datetime-of
IS-A
IS-A
Electrolytes
Blood_Gases
IS-A
IS-A
IS-A
IS-A
Na
K
pH
O2_Sat
form-of assoc
Association-Form
assoc with-form
String

Because all relationship functions and relationship predicates are derived from a specified relationship type, they need not be strictly among the minimal set of FER modeling constructs. However, relationship functions and relationship predicates figure prominently in the FER query language and are among the primary constructs of the functional model, from which the FER model derives. I have, therefore, included them among the core set of FER modeling constructs.

3.2.2 Comparison to previous semantic data models

The FER model is a hybrid of two similar semantic data models, the functional model (FM) [Shipman 1981] and the extended entity-relationship model (EERM) [Hohenstein and Gogolla 1989]. I selected these models as the precursors of the FER model because they both possess features consistent with the design criteria of an abstract data model for clinical information.

Both models consist of a small set of abstract modeling constructs. The constructs of FM include only *types*, *entities*, and *functions*, whereas the constructs of EERM consist of *entity types*, *relationship types*, *instances of entity types and relationship types*, and *attributes*. Both models support the description of data in an implementation-independent way by suppressing operational details such as the types of identifiers used to distinguish objects or the data structures used to associate objects. FM and EERM also both support the inheritance of functions and relationships (respectively) by subsumed types and entity types (respectively).

Both models include query languages that combine the features of declarative set-based data-manipulation languages, such as SQL, with functional expression-based languages, such as LISP. The query language for the EERM is called the **EER calculus** (EERC) and the query language for the FM is called **DAPLEX**. The detailed syntax and semantics of DAPLEX and the EERC are described elsewhere [Hohenstein and Engels 1990, Shipman 1981]. The example queries in Figure 3.3 provide an introduction to the syntax and semantics of each query language. Both queries in Figure 3.3 are intended to retrieve the names of all patients treated by a physician who is the same age as the patient. Note that DAPLEX and the EERC both rely on function applications to navigate between associated entities. For example, the navigation from Physician to Patient is accomplished via the function application "Physician-Of(Patient)" in DAPLEX. Functional

navigation obviates the need for users to specify complex join expressions based on the identifying attributes of entities, and thereby increases the intuitiveness of both query languages. Also, DAPLEX and EERC queries specify declaratively the target entities to be retrieved rather than specifying procedures to retrieve them. Declarative query formulation allows underlying implementations to structure and manipulate data in a variety of ways, thereby increasing the abstractness of both query languages.

```
DAPLEX:   FOR EACH Patient
            SUCH THAT FOR SOME Physician-Of(Patient)
                  Age-Of(Physician-Of(Patient)) = Age-Of(Patient)
            PRINT Name-of(Patient)

EERC:     Name-of (pt)  |  ( pt : Patient ) AND
                              ∃ ( md : physician-of(pt))
                                    ( age-of (md) = age-of(pt) )
```

schema is specified is undesirable for an abstract conceptual model, because underlying implementations may vary in their representations. Therefore, the FER model adopts the approach of the functional model and represents all associations using a single construct — the relationship type.

2. The FM makes a precise distinction between types and functions, specifying that types denote sets of entities, the properties of which are described by functions, whereas functions denote featureless operations that return the properties of entities. The minimal nature of these constructs creates difficulties when modeling certain real-world semantics because the association between an entity and one of its properties may itself have a property. For example, the "diagnosis" association between a patient and a disease may have a diagnosing physician associated with it. The FM cannot simultaneously model "diagnosis" as a function that returns the diseases associated with a patient and as an entity that is itself associated with a physician. Typically, FM schemas circumvent this inadequacy by modeling associations such as "diagnosis" as types, thereby losing the semantics that "diagnosis" also represents an association. This modeling decision also sacrifices the ability to use "diagnosis" as a function in FM queries. Therefore, the FER model adopts the approach of the EER model and represents associations as relationship types, which both may have their own properties *and* define relationship functions that may be used in FER queries. This dual role of relationship types as associations *and* as entities proves particularly useful for modeling the temporal semantics of clinical data, because the ability to ascribe properties to relationship types allows all associations to be attributed with temporal tags.

3.2.3 Formal semantics of the FER data model

Although the modeling constructs of the FER data model are intended to be intuitive and clear, it is useful to specify formally each construct, so that the meanings of FER schemas and operations on FER schemas are unambiguous. In the following sections, I formally specify the semantics of entity types, IS-A

connections, relationship types, relationship functions, and relationship predicates.

3.2.3.1 Entity types and entities

Given an FER schema and a database state, each user-defined entity type "E" in the schema defines a set E = { e_1,..., e_n}, where each e_i , $1 \le i \le n$, represents a unique object in the domain of discourse. The objects e_1,..., e_n are called entities. For notational convenience, I will henceforth refer to both the entity type "E" and the set of entities E using the (unquoted) symbol E. For example, in Figure 3.1, the entity type Patient defines the set Patient, which contains all of the entities represented in a particular database that are patients. The FER data model defines two varieties of entity types — user-defined entity types and built-in entity types. User-defined entity types represent finite sets of arbitrary objects; the membership of user-defined entity types may change depending on the state of a database. The built-in entity types are Integer, Real, String, and Datetime. Each built-in entity type defines a set of entities that is infinite and independent of the state of the database:

Integer: the set of integers; for example, 44456, -55, 0

Real: the set of real numbers; for example, 56.0, -0.0006, 789.3

String: the set of character strings consisting of zero or more ASCII characters; for example, "Jim", "", "%%%**$-U"

Datetime: the set of temporal denotations of the form "YYYY/MM/DD HHMM" in which HHMM represents military (24 hour) time; for example, "1995/03/15 1730"

3.2.3.2 IS-A connections

User-defined entity types may be related by the binary relation IS-A. The semantics of the IS-A relation over two entity types, E and E', are that E' *subsumes* E. More precisely,

$$\text{IS-A}(E, E') \Rightarrow \text{if } e \in E, \text{ then } e \in E' .$$

Notice that the semantics of the IS-A relation are transitive, owing to the transitivity of set membership. I define the binary relation SUBTYPE(E, E′) to denote the existence of a *chain* of one or more IS-A connections from E to E'. The transitivity of the IS-A relation implies

$$\text{SUBTYPE}(E, E') \Rightarrow \text{if } e \in E, \text{ then } e \in E'.$$

I use this property later to show that inheritance holds over entities related by a chain of IS-A connections.

3.2.3.3 Relationship types and relationships

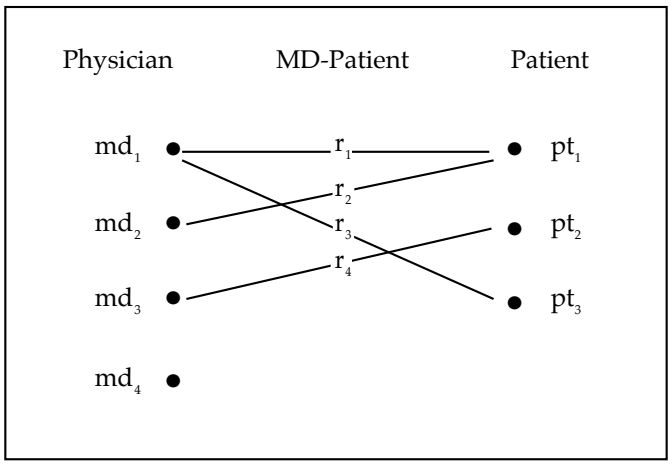
Given an FER schema and a database state, each relationship type "R" in the schema defines a set $R = \{ r_1,..., r_m\}$, where each r_i , $1 \le i \le m$, represents a distinct association between two objects that is represented in the database. The associations $r_1,..., r_m$ are objects themselves and are called relationships. As with entity types, I will henceforth refer to both the relationship type "R" and the set of relationships R using the (unquoted) symbol R. For example, in Figure 3.1, the relationship type MD-Patient defines the set MD-Patient, which contains all of the pairwise associations between physicians and patients that are represented in the database.

Each relationship type R is *typed* by an ordered pair of entity types <E, E′>, which denotes the entity types that *participate in* the relationship type. For example, the relationship type MD-Patient in the FER schema of Figure 3.2 is typed by the ordered pair of entity types <Physician, Patient>. Formally, if <E, E′> participates in the relationship type R, then participant_types(R) = <E, E′> and $\text{participant_type}_1(R) = E$ and $\text{participant_type}_2(R) = E'$. Each instance of a relationship type (that is, each relationship) is a unique object that associates an ordered pair of entities <e, e′>. Formally, if the relationship ρ associates the entities <e, e′>, then participants(ρ) = <e, e′>, $\text{participant}_1(\rho) = e$, and $\text{participant}_2(\rho) = e'$. It is important that the entity pair associated by a relationship be ordered because the semantics of a relationship may be very different if the positions of the associated entities are reversed. For example, if e and e′ are instances of the entity type Person and $e \neq e'$, then the semantics of the relationship Lord-Vassal(e, e′) are quite different from the semantics of the relationship Lord-Vassal(e, e′). Lastly, there exists the type constraint

$$\forall \rho \in R,\ \text{participant}_1(\rho) \in \text{participant_type}_1(R) \text{ AND } \text{participant}_2(\rho) \in \text{participant_type}_2(R),$$

which states that relationships can associate only entities of the appropriate entity types, as specified by the relationship type to which the relationship belongs.

A graphical example of entity types, entities, relationship types, and relationships appears in Figure 3.4, which depicts the state of a database for part of the FER schema defined in Figure 3.2. In the database state shown, the entity type Physician defines a set of four entities $\{md_1, md_2, md_3, md_4\}$ and the entity type Patient defines a set of three entities, $\{pt_1, pt_2, pt_3\}$. The relationship type MD-Patient defines a set of four relationships $\{ r_1, r_2, r_3, r_4 \}$, each of which associates a pair of Physician and Patient entities.



For example, the relationship type MD-Patient in Figure 3.2 defines the relationship functions "physician-of" and "patient-of" such that

$$f_{\text{MD-Patient}} = \text{patient-of} : \quad \text{Physician} \rightarrow \{ \text{Patient} \}$$
$$g_{\text{MD-Patient}} = \text{physician-of} : \quad \text{Patient} \rightarrow \{ \text{Physician} \}.$$

Because relationship types are directional, each relationship function ϕ has a "source" entity type, source(ϕ), and a "target" entity type, target(ϕ). Informally, source(ϕ) and target(ϕ) denote the entity types of the instances that the function ϕ maps "from" and "to," respectively. For example, source(patient-of) = Physician and target(patient-of) = Patient. Formally, for each pair of relationship functions f_R and g_R defined by the relationship type R, where participant_types(R) = <E, E'>,

$$\text{source}(f_R) = \text{target}(g_R) = E \text{ and}$$
$$\text{target}(f_R) = \text{source}(g_R) = E'.$$

As defined, relationship functions are applied to entities of the source entity type and return sets of entities of the target entity type. The semantics of this operation are defined as follows: Given the relationship functions f_R and g_R defined by the relationship type R, where participant_types(R) = <E, E'>,

(1) the expression $f_R(e)$ is valid iff $e \in \text{source}(f_R)$ and
the expression $g_R(e')$ is valid iff $e' \in \text{source}(g_R)$

(2) $f_R(e) = \{ e' : \exists r \in R$ such that participants(r) = <e, e'>
$g_R(e) = \{ e' : \exists r \in R$ such that participants(r) = <e', e>.

Informally, this definition simply states that the argument to a relationship function must be an entity of the appropriate entity type and that the result of applying a relationship function is the set of entities that are associated with the argument entity by some relationship. For example, in the sample database state of Figure 3.4, physician-of(pt_1) is a valid expression because $pt_1 \in$ Patient, and physician-of(pt_1) = $\{md_1, md_2\}$ because there exist the relationships $r_1, r_2 \in$ MD-Patient such that participants(r_1) = <md_1, pt_1> and participants(r_2) = <md_2, pt_1>.

Any entity type E that is subsumed by another entity type E' via a chain of IS-A connections inherits all of the relationship functions that apply to E'. By "inherits

relationship functions," I mean that the instances of the subsumed entity type are valid arguments to any relationship function that applies to the subsuming entity type. Formally,

$$\text{SUBTYPE}(E, E') \Rightarrow \text{if } e \in E \text{ then } \forall\, \phi \text{ such that source}(\phi) = E', \phi(e) \text{ is valid.}$$

For example, by the FER schema of Figure 3.2, the relationship function "patient-with-lab" may be applied to an instance of the entity type Electrolytes to return the set of patients who have had a particular electrolyte lab test performed.

3.2.3.5 Relationship predicates

Each relationship type R defines a relationship predicate p_R that determines if a pair of entity instances are related by an instance of the relationship type R. The value of a relationship predicate may be the Boolean TRUE or FALSE and the name of the relationship predicate is the same as the name of the relationship type. Specifically, if participant_types(R) = <E, E'>, then

$$p_R = \text{"R"} : \quad E \times E' \rightarrow \text{Boolean.}$$

For example, the relationship type MD-Patient in the schema of Figure 3.2 defines the relationship predicate MD-Patient, which takes an instance of the entity type Physician and an instance of the entity type Patient and evaluates to TRUE if the instances are associated (that is, if a specified physician is treating a specified patient), and to FALSE if they are not associated. The semantics of this predicate are defined as follows: Given the relationship predicate p_R defined by the relationship type R, where participant_types(R) = <E, E'>,

(1) the expression $p_R(e, e')$ is valid iff $e \in$ participant_type$_1$(R) and $e' \in$ participant_type$_2$(R)

(2) $p_R(e, e')$ = TRUE iff $\exists\, r \in R$ such that participants(r) = <e, e'>, else $p_R(e, e')$ = FALSE.

For example, in the sample database state of Figure 3.4, MD-Patient(md_2, pt_1) is a valid expression because $md_2 \in$ Physician and $pt_1 \in$ Patient, and MD-Patient(md_2, pt_1) = TRUE. However, MD-Patient(pt_1, md_2) is not a valid expression, and MD-Patient(md_2, pt_3) = FALSE. Note that it is not necessarily the case that $p_R(e, e')$ =

p_R (e', e) for any pair of entities e and e' because participants(r) defines the *ordered* pair of entities that are associated by the relationship r.

Any entity type E that is subsumed by another entity type E' via a chain of IS-A connections inherits all of the relationship predicates that apply to E'. By "inherits relationship predicates," I mean that instances of the subsumed entity type validly may appear in the same position within relationship predicates as instances of the subsuming entity type. Formally,

$$\text{SUBTYPE}(E, E') \Rightarrow \text{if } e \in E \text{ and } e'' \in E''$$
$$\text{then } \forall\, p_R \text{ such that } \text{participant_type}_1(R) = E' \text{ and } \text{participant_type}_2(R) = E''$$
$$p_R(e, e'') \text{ is valid}$$

and (the symmetrical case)

$$\text{SUBTYPE}(E, E') \Rightarrow \text{if } e \in E \text{ and } e'' \in E''$$
$$\text{then } \forall\, p_R \text{ such that } \text{participant_type}_1(R) = E'' \text{ and } \text{participant_type}_2(R) = E'$$
$$p_R(e'', e) \text{ is valid.}$$

For example, by the FER schema of Figure 3.2, the relationship predicate "Person-Last_Name" may be applied to an instance of the entity type Patient and an instance of the entity type String to determine if a specific patient has a specific last name.

The formal definitions of the syntax and semantics of the FER data model serve three purposes:

1. They specify the meaning of the FER data model precisely.

2. They provide a foundation for the precise specification of the query language that accompanies the FER data model.

3. They provide a precise reference point for assuring that the translation of queries formulated with respect to the FER data model is accurate.

I describe the query language that accompanies the FER data model in the next section, and I describe the process for accurately translating queries from this language to SQL in Chapter 4.

3.3 The ReFER Query Language

Because the FER data model is a hybrid of the functional model and the EER model, it is not surprising that the query language for the FER model is largely derived from DAPLEX and the EERC. This section discusses the ReFER query language, pointing out the desirable features that it has inherited from its precursors, and describing added features that enhance its functionality and usefulness.

3.3.1 Desiderata

The general design criteria of an abstract query language are identical to the criteria for an abstract query model:

1. *Expressiveness*: The language should support the retrieval of relevant clinical information in an intuitive manner. "Relevant" queries in the context of clinical decision-support applications encompass requests for a wide range of database objects:

    - A single entity along with its associated entities (for example, a particular patient and all her medication and lab orders)

    - A set of entities based on complex selection criteria (for example, all the diuretic medications ordered in the past week for patients with congestive heart failure)

    - The aggregated properties of a set of entities (for example the average length of stay for all patients admitted in July)

    An expressive query language must have constructs to support all these queries, including predicates to determine associations among entities and aggregation functions to abstract sets of values. An expressive

language also must support the formulation of queries in an "intuitive" manner. For query formulations to correspond to human cognitive models of information retrieval, low-level query constructs, such as explicit relational joins and pointer-chain navigation, should be avoided.

2. *Abstractness*: The constructs of the query language should be sufficiently general to support a variety of implementations at specific provider sites. In other words, the same abstract query constructs should subsume diverse implemented data models, diverse database schemas, and diverse query languages. As a negative example, a query language that requires the specification of a relational join to retrieve the values of patients' laboratory tests (as shown in Figure 3.1) is insufficiently abstract, because a specific implementation of the schema in Figure 3.1 may represent patients' names in the same relation as their laboratory test results (which would require no join operation).

The query languages of the EER data model (EER calculus [Hohenstein and Gogolla 1989]) and of the functional data model (DAPLEX [Shipman 1981]) fulfill the expressiveness and abstractness criteria. The primary mechanism for achieving expressiveness and abstractness in the EER calculus and in DAPLEX is the use of *functions* for navigating database schemas. For example, the term

"age-of (patient-of(physician))"

in the EER calculus represents the ages of all of the patients treated by a physician. The term specifies function applications to associate physicians with patients and patients with ages. This method of manipulating data enhances the intuitiveness of a query language, because it obviates the need to specify the *means* by which associations among entities are made at the time a query is evaluated. The user needs to know only that an association may exist. Also, functional data manipulation enhances the power of the query language by supporting the definition of *arbitrary functions*, such as those for aggregation:

"AVG (age-of (patient-of (physician)))".

Functional data manipulation also enhances the abstractness of the query language by obviating the need to specify *implementation-specific* details

regarding the means by which associations between entities are made. In general, associations may be represented in three (implementation-specific) ways:

1. The association may be stored explicitly. For example, the patient relation may have a stored attribute "age."

2. The association may be derived from stored values. For example, the patient relation in the database may have a stored attribute "date-of-birth," which can be compared to the current date to generate the age.

3. The association may be computed by a process external to the database. For example, the query engine may consult a database at the state Census Bureau to retrieve the age.

At the level of a site-independent query language, however, the means by which a specific database represents an association is inconsequential. It is only important that the database represents the association in *some* way. Data manipulation via function application, therefore, imparts implementation-independence to the query language.

The syntax and semantics of the FER query language that I have defined are based on the syntax and semantics of the EER calculus and of DAPLEX. I describe the query language, called **ReFER** (Retrieval language for the FER model) in the following sections.

3.3.2 ReFER syntax and semantics: an overview

The basic syntactic form of a ReFER query is defined by the following grammar rules:

(1) Query ::= '[' Termlist '|' Decllist 'AND' Predicate ']'
(2) Query ::= '[' Termlist '|' Decllist ']'

The symbol "Decllist" denotes a list of variable declarations, the symbol "Predicate" denotes constraints on the values of the declared variables, and the symbol "Termlist" denotes the set of entities that the query returns. Note that the Predicate clause is optional. Several sample queries based on the FER schema

of Figure 3.1 illustrate the level of abstraction and the intuitiveness of ReFER syntax and semantics:

Q1. [last-name-of(pt), age-of(pt) | (pt : Patient)]

Meaning: Retrieve the last name and the age of every patient in the database.

Q2. [last-name-of(pt) | (pt : Patient) AND age-of(pt) > 55]

Meaning: Retrieve the last name of every patient in the database who is older than 55.

Q3. [last-name-of(pt), last-name-of(physician-of(pt)) | (pt : Patient)]

Meaning: Retrieve the last name of every patient paired with the last name of each physician who is treating that patient.

Q4. [value-of(lyte), datetime-of(lyte), | (pt : Patient), (lyte : Electrolyte)
AND
Patient-Test(pt,lyte) AND
datetime-of(lyte) > "1995/10/20 0800" AND
last-name-of(pt) = "Tetrazzini"]

Meaning: Retrieve the value and the time of each electrolyte test performed on the patient named "Tetrazzini" since 8:00 a.m. October 20, 1995.

Q5. [last-name-of(pt) | (pt : Patient) AND
EXISTS [(tst : K) AND Patient-Test(pt, tst) AND
(value-of(tst) > 6.0 OR
value-of(tst) < 3.0)]]

Meaning: Retrieve the last name of every patient who has at least one potassium value that is greater than 6.0 or less than 3.0.

Q6. [last-name-of(md), AVG(age-of(patient-of(md))) | (md : Physician)]

Meaning: Retrieve the last name of every physician in the database paired with the average age of the patients treated by that physician.

The semantics of the ReFER language are based on a *domain-oriented calculus* [Pirotte 1978]. In a domain-oriented calculus, variables range over declared domains (or entity types in the case of the FER model). The results of queries are computed by iteratively binding variables to each of the entities over which they range, and evaluating the expressions containing the variables based on those variable bindings. For example, consider the ReFER query:

[last-name-of(pt) | (pt : Patient) AND age-of(pt) > 55].

This query returns the last names of all patients over the age of 55. The meaning of this query is defined as follows: The declaration "(pt : Patient)" declares the variable "pt" to range over all entities in the current database state that are members of the FER entity type Patient. The declaration effectively defines a *set of bindings* for the variable "pt," such that each element of the set binds "pt" to one of the entities in the entity type Patient. The predicate "age-of(pt) > 55" constrains this set of bindings to only those bindings that satisfy the inequality. Specifically, the predicate determines the age associated with each entity to which "pt" is bound, and tests whether this value is greater than 55. If the associated age is greater than 55, the predicate retains that binding of "pt"; if it is not greater than 55, the predicate discards that binding of "pt." For each binding of "pt" that remains after the predicate is applied, the term "last-name-of(pt)" determines the last name associated with the entity to which "pt" is bound. The union of the last names thus determined for each binding of "pt" that satisfies the predicate constitutes the result of the query.

I define the syntax and semantics of the ReFER symbols "Termlist," "Decllist," "Predicate," and "Query" more formally in the following sections. The formalization is based on a specification of the EER calculus [Hohenstein and Engels 1990].

3.3.3 Declarations

In grammar rules (1) and (2) for ReFER queries, Decllist denotes a list of declarations for the variables appearing in a ReFER query. The syntax of Decllist is defined by the following grammar rules:

(3) Decllist ::= Decl ',' Decllist
(4) Decllist ::= Decl
(5) Decl ::= '(' Variable ':' Entity_Type ')'

Examples of valid declaration lists are

"(pt : Patient)"
"(pt : Patient), (md : Physician)" and
"(pt1 : Patient), (pt2 : Patient)"

Each declaration associates a variable with an FER entity type. Informally, the semantics of a declaration (v : E) are that the declared variable v "ranges" over the members of the entity type E that exist in the current database state. Formally, the declaration δ = (v : E) defines a *set of bindings*, B(δ), such that for each entity e $\in$ E, there is a binding b $\in$ B(δ) of the form <v=e> that *assigns* e to the variable v. For example, if the entity type Patient consists of the entities {*p1*, *p2*} in a database state, the declaration δ = (pt : Patient) defines the set of bindings B(δ) such that

B(δ) = { <pt = *p1*>, <pt = *p2*> }.

Note that the set of bindings that a declaration defines must be finite, otherwise a query may return an infinite result. This "safety" property [Hohenstein and Gogolla 1989] implies that variables cannot be declared over the built-in entity types of the FER data model, such as Integer and String, because these entity types are infinite. The following query, therefore, is not valid:

[num | (num : Integer) AND num > 55].

In general, the elements of binding sets defined by declaration lists may be *tuples* of variable assignments, such that each element of a tuple specifies the assignment of a separate declared variable. Tuple-valued binding sets result from declaration lists that contain multiple declarations because the meaning of

such declaration lists is the *Cartesian product* of the binding sets defined by each declaration. Formally, given the declarations δ_1 and δ_2, if $B(\delta_1) = \beta_1$ and $B(\delta_2) = \beta_2$, then $B(\delta_1,\delta_2)= \beta_1 \times \beta_2$. For example, assume that the FER entity type Patient consists of the entities {*p1*, *p2*} and the FER entity type Physician consists of the entities {*m1*, *m2*}. The declaration list "(pt : Patient), (md : Physician)" defines the binding set

{ <pt = *p1*, md = *m1*>,
<pt = *p1*, md = *m2*>,
<pt = *p2*, md = *m1*>,
<pt = *p2*, md = *m2*> }.

3.3.4 Terms

Terms in the ReFER query language denote *multisets* of entities, that is, sets of entities in which duplicates may exists. Syntactically, terms may appear in a ReFER query either in the Termlist that specifies the results or in the Predicate that specifies the constraints. For example, the query

[last-name-of(pt) | (pt : Patient) AND age-of(pt) > 55]

includes the terms "last-name-of(pt)" and "age-of(pt)" (recall that String and Integer are entity types in the FER model so that the values of a patient's last name and age are consistent with the formal definition of terms). The syntax of Terms is defined by the following grammar rules, each of which is accompanied by an example:

(6) Term ::= Constant
Example: "55"
(7) Term ::= Variable
Example: "pt"
(8) Term ::= Function '(' Term ')'
Example: "age-of(pt)"
(9) Term ::= AggFunction '(' Term ')'
Example: "AVG(age-of(pt))"
(10) Term ::= Term ArithFunction Term
Example: "age-of(pt) + age-of(md)"

The symbol "Function" may be any of the relationship functions defined for a particular FER schema, such as "age-of" or "last-name-of." The symbol "AggFunction" may be any of the aggregation functions MIN, MAX, AVG, SUM, or COUNT. The symbol "ArithFunction" may be any of the arithmetic functions +, –, *, or /.

The meaning of a term in ReFER is a multiset of entities, the elements of which depend on the binding of the declared variables appearing in the term. For example, assuming a query includes the declaration "(pt : Patient)", the term

"physician-of(pt)"

denotes the set of physicians associated with the patient entity that is bound to the variable "pt." The value of a term, therefore, can only be determined in the context of a particular binding. I formally denote the value (that is, meaning) of a term by the "value" function

ν[Term, b]

in which "Term" is a term according to the grammar rules listed above and "b" is an element of the binding set defined by the declaration list of the query in which "Term" appears. For example, the value of the variable "pt" in the context of a binding that assigns the entity *p1* to "pt" is simply the value *p1*:

ν[pt, <pt = *p1*>] = *p1*

A definition of the value function "ν[]" exists for each syntactic form of a term, and together these value functions completely specify the meaning of terms in ReFER. Each definition of a value function in the following list is preceded by the number of the grammar rule to which the definition corresponds followed by the "meaning" designator *m*.

(6*m*) ν[Constant, b] = the value of the constant

For example, ν[55, b] = 55 and ν["Smith", b] = "Smith" regardless of the bindings specified by b.

(7*m*) ν[Variable, b] = the entity assigned to the variable in the binding b

For example, ν[md, <md = *d1*>] = *d1*.

(8*m*) ν[Function(Term), b] = Function (ν[Term, b])

This recursive definition specifies that the value of an FER relationship function applied to a term is the FER relationship function applied to the value of the term. The meaning of an FER relationship function applied to a term was defined in Section 3.2.3.4.

For example,

ν[age-of(physician-of(pt)), b] = age-of(ν[physician-of(pt), b]).

Informally, the value of the term "age-of(physician-of(pt))" is the multiset of the ages associated with the physicians who are treating the patient entity bound to "pt." Note that if two physicians are both treating the patient bound to "pt" and the two physicians have the same age, that age will appear twice in the value of the term "age-of(physician-of(pt))."

(9*m*) ν[AggFunction(Term), b] = AggFunction (ν[Term, b])

This definition specifies that the value of an aggregation function applied to a term is the result of applying the aggregation function to the value of the term. Because the values of terms are always *multisets* of entities, aggregation functions may always be applied to the values. In ReFER, the aggregation functions MIN, MAX, AVG, SUM, and COUNT have the same semantics as their counterparts in SQL [Date 1989a].

For example,

ν[AVG(age-of(physician-of(pt))), b] = AVG(ν[age-of(physician-of(pt)), b]).

The value of the term is the average of the multiset containing the ages of the physicians associated with the patient bound to "pt." Note how this example is different from

ν[AVG(age-of(pt)), b] = AVG(ν[age-of(pt), b]),

which equals the average of the set of ages of each patient. If each patient has only one age, then the expression is equivalent to v[age-of(pt), b], because the AVG function will always be applied to a set containing one element.

(10*m*) v[$Term_1$ ArithFunction $Term_2$, b] = v[$Term_1$, b] ArithFunction v[$Term_2$, b]

This definition specifies that the value of an arithmetic function applied to two terms is the result of applying the arithmetic function to the values of the terms. In ReFER, the arithmetic functions +, -, *, and / have the same meanings as their counterparts in SQL [Date 1989a]. For these arithmetic operations to be meaningful, each subterm v[$Term_1$, b] and v[$Term_2$, b] must evaluate to a set consisting of one element. This constraint can be checked at the time the query is compiled based on the cardinality constraints specified for the relationship types in the FER schema. For example,

v[age-of(pt) + age-of(md), b] = v[age-of(pt), b] + v[age-of(md), b].

Given the bindings of "pt" and "md" in b, the value of the term is equal to the value of the age associated with the Patient entity bound to "pt" plus the value of the age associated with the Physician entity bound to "md," (assuming that only one age may be associated with each entity).

In the ReFER query language, individual terms that conform to the grammar rules (6) – (10) may be concatenated into Termlist clauses according to the following two grammar rules

(11) Termlist ::= Term ',' Termlist
(12) Termlist ::= Term

Term list clauses are used to denote the results of ReFER queries when more than one type of entity is requested. For example, the query

[last-name-of(pt), age-of(physician-of(pt)) | (pt : Patient)]

requests the name of each patient and the ages of that patient's physicians. The query includes the Termlist clause "last-name-of(pt), age-of(physician-of(pt))." The meanings of Termlist clauses also are defined in the context of a specific

binding of the variables that appear in the term list. Formally, the value of a Termlist clause in the context of a binding is the *Cartesian product* of the values of each element of the Termlist clause in the context of the same binding:

(11*m*) ν[Term, Termlist, b] = ν[Term, b] $\times$ ν[Termlist, b].

If a Termlist consists of a single term, as in the grammar rule (12), then

(12*m*) ν[Termlist, b] = ν[Term, b].

For example, if b = <pt = *p1*> and the Patient entity *p1* has the last name "Smith" and is treated by two physicians who are aged 40 and 50, then

ν[last-name-of(pt), age-of(physician-of(pt)), b] =

ν[last-name-of(pt), b] $\times$ ν[age-of(physician-of(pt)), b] =

{ "Smith" } $\times$ { 40, 50 } =

{ <"Smith", 40>, <"Smith", 50>}.

3.3.5 Queries without predicates

Given the formal definitions of Decllist and Termlist, I can now formally define the meaning of ReFER queries of the form

(2) Query ::= '[' Termlist '|' Decllist ']'

(recall that the Predicate clause is optional). If Termlist and Decllist conform to the syntactic and semantic specifications that I described above, then I define the value (that is, meaning) of the query "[Termlist | Decllist]" as

(2*m*) $$\bigcup_{b \in B(\text{Decllist})} \nu[\text{Termlist}, b]$$

where the symbol $\cup$ denotes a *multiunion*, that is, a union that retains duplicate values. Informally, this definition states that the value of a query is the multiunion of the values of the Termlist that are computed for each binding in the binding set defined by the Decllist. For example, the value of the query

[last-name-of(pt), last-name-of(md) | (pt : Patient), (md : Physician)]

is (conceptually) computed by (1) generating a binding set consisting of the Cartesian product of all the Patient and Physician entities, (2) determining for each pair of Patient and Physician entities in the binding set the last names that corresponds to the Patient entity and to the Physician entity, and (3) returning the multiunion of all such pairs of last names.

3.3.6 Predicates

The role of the predicate in a ReFER query is to constrain the binding set defined by the Decllist to only those bindings that satisfy the logic of the predicate. Conceptually, one may think of the predicate in a query as a "filter" on the binding set defined by the declaration list in the query. For each binding in the binding set, the assignment for each variable is substituted into the corresponding variable of the predicate; if the predicate evaluates to TRUE under the substitution specified by a binding, the binding is retained in the binding set, otherwise the binding is discarded. The bindings that remain are used to compute the value of the query.

The syntax of Predicate is defined by the following grammar rules, each of which is accompanied by an example:

(13) Predicate ::= $Term_1$ Operator $Term_2$
Example: "age-of(pt) > age-of(md)"

(14) Predicate ::= Relationship '(' $Term_1$ ',' $Term_2$ ')'
Example: "MD-Patient(md, pt)"

(15) Predicate ::= 'EXISTS' '[' Decllist 'AND' Predicate ']'
Example: "EXISTS [(md:Physician) AND MD-Patient(md,pt)]"

(16) Predicate ::= 'NOT' Predicate
Example: "NOT age-of(pt) > 55"

(17) Predicate ::= $Predicate_1$ 'AND' $Predicate_2$
Example: "age-of(pt) > 55 AND age-of(pt) < 80"

(18) Predicate ::= $Predicate_1$ 'OR' $Predicate_2$
Example: "age-of(md) >60 OR age-of(md) < 35"

The symbol "Operator" may be one of =, <>, >, ≥, <, or ≤. The syntax and semantics of the symbol "Term" are defined in Section 3.3.4. The symbol "Relationship" may be any of the relationship types defined for a particular FER schema, such as MD-Patient or Person-Last_Name. Because predicates filter binding sets by testing one binding at a time, the meaning of a predicate is defined as the *truth value* of the predicate in the context of a particular binding. I formally denote the truth value of a predicate by the function

$$\tau[\text{Predicate}, b]$$

in which "Predicate" is a predicate according to grammar rules (13) – (18) and "b" is an element of the binding set defined by the declaration list of the query in which "Predicate" appears. For example,

$$\tau[\text{last-name-of(pt)} = \text{“Smith”}, \langle \text{pt} = p1 \rangle] = \text{TRUE}$$

if the string associated with the Patient entity *p1* by the relationship type Person-Last_Name is "Smith." Otherwise,

$$\tau[\text{last-name-of(pt)} = \text{“Smith”}, \langle \text{pt} = p1 \rangle] = \text{FALSE}.$$

A definition of the truth value function "τ[]" exists for each syntactic form of a predicate, and together these functions completely specify the meanings of predicates in ReFER. Each definition of a truth value function in the following list is preceded by the number of the grammar rule to which the definition corresponds.

(13*m*) $\tau[\text{Term}_1 \text{ Operator } \text{Term}_2, b] = \text{TRUE iff}$
$$\nu[\text{Term}_1, b] \text{ Operator } \nu[\text{Term}_2, b]$$

This definition specifies that a comparison operator holds over two terms under the binding b only if the operator holds over the *values* of the two terms under the binding b. In ReFER, the comparison operators =, <>, >, ≥, <, and ≤ have the same meanings as their counterparts in SQL [Date 1989a]. For example,

$$\tau[\text{ age-of(pt)} > \text{age-of(md)}, b] = \text{TRUE iff } \nu[\text{age-of(pt)}, b] > \nu[\text{age-of(md)}, b].$$

(14*m*) $\tau[\text{ Relationship}(\text{Term}_1, \text{Term}_2), b\ ] = \text{TRUE iff}$
$$\text{Relationship}(\nu[\text{Term}_1, b], \nu[\text{Term}_2, b]).$$

This definition specifies that a relationship predicate holds over two terms under the binding b only if the relationship predicate holds over the *values* of the two terms under the binding b. The meaning of an FER relationship predicate applied to a pair of terms are defined in Section 3.2.3.5.

For example,

τ[MD-Patient(md,pt), b] = TRUE iff

MD-Patient(v[md, b], v[pt, b])

Note that for the comparison operator and the relationship predicate specified in (13) and (14) to be meaningful, both of the subterms appearing in the predicates must evaluate to a set consisting of one element given the binding b. This constraint can be checked at the time the query is compiled based on the cardinality constraints specified for the relationship types in the FER schema.

(15*m*) τ[Exists[Decllist AND Predicate], b] = TRUE iff
$\exists$ b'$\in$ B(Decllist) such that τ[Predicate', b'] = TRUE

where Predicate' = Predicate with each free variable substituted according to the assignments specified in the binding b. The binding b assigns values to each of the *free variables* in Predicate (that is, the variables not declared in Decllist_1); the "Exists" predicate tests whether Predicate holds for *any* of the bindings defined by Decllist. For example,

τ[Exists[(md:Physician) AND age-of(md) > age-of(pt)], <pt = *p1*>] =
TRUE iff
$\exists$ b $\in$ B((md:Physician)) such that τ[age-of(md) > age-of(*p1*), b] = TRUE.

(Note the substitution of the value *p1* for the variable "pt" in the predicate.) The "Exists" predicate determines if there are any physicians older than the patient *p1*.

(16*m*) τ[NOT Predicate , b] = TRUE iff τ[Predicate, b] = FALSE

For example,

τ[NOT MD-Patient(md,pt) , b] = TRUE iff
τ[MD-Patient(md,pt), b] = FALSE.

(17*m*) τ[Predicate$_1$ AND Predicate$_2$, b] = TRUE iff
τ[Predicate$_1$, b] = TRUE AND τ[Predicate$_2$, b] = TRUE

(18*m*) τ[Predicate$_1$ OR Predicate$_2$,b] = TRUE iff
τ[Predicate$_1$, b] = TRUE OR τ[Predicate$_2$,b] = TRUE

3.3.7 Queries with predicates

Given the formal definition of Predicate, I can now formally define the meaning of "complete" ReFER queries of the form

(1) Query ::= '[' Termlist '|' Decllist 'AND' Predicate ']'

Given that the predicate in a ReFER query "filters" the binding set defined by the declaration list, the value of a query that includes a predicate is

(1*m*) $$\bigcup_{b \in B(\text{Decllist, Predicate})} \nu[\text{Termlist, b}]$$

where B(Decllist, Predicate) = { b : b $\in$ B(Decllist) and τ[Predicate, b] = TRUE}. Informally, the definition states that the result of a ReFER query is the multiunion of the values of the term list computed for each binding that is specified in the declaration list and that fulfills the logical condition of the predicate.

3.3.8 Subqueries: terms revisited

The ReFER query language supports the use of subqueries. Subqueries are terms that have the syntactic form of complete queries, but that include one or more

free variables, that is, variables that are not declared in the declaration list of the subquery. The syntactic form of a subquery term is

(19) Term ::= [Termlist | Decllist AND Predicate]

The value of a subquery term is defined by the value function that corresponds to the grammar rule (19):

(19*m*) ν[[Termlist | Decllist AND Predicate], b] =

$$\bigcup_{b' \in B(\text{Decllist, Predicate}')} \nu[\text{Termlist, } b']$$

where Predicate' = Predicate with each free variable substituted according to the assignments specified in the binding b.

For example,

$$\nu[\ [\text{last-name-of(md)} \mid (\text{md} : \text{Physician}) \text{ AND } \text{MD-Patient(md,pt)}\ ], \langle \text{pt} = p1 \rangle] = \bigcup_{b' \in B(\ (\text{md} : \text{Physician}),\ \text{MD-Patient(md,}p1\text{)}\ )} \nu[\text{last-name-of(md)}, b']$$

Note the substitution of the value *p1* for the variable "pt" in the predicate. The value of this term is the value of the query with the variable "pt" substituted by the constant *p1*, as specified in the binding.

In ReFER queries, subqueries may appear anywhere terms may appear, provided that they conform to the semantic constraints that exist for certain terms (such as being single-valued if they appear as the argument to a comparison operator). This feature allows one to formulate certain queries more concisely than would be possible in relational query languages, such as SQL. For example, consider the complex ReFER query

```
[last-name-of(pt),
 MIN( [value-of(lab) | (lab : pH) AND Patient-Lab(pt,lab) ] )
        | (pt : Patient) AND
          AVG( [value-of(sat) | (sat : O2Sat) AND Patient-Lab(pt, sat) ] ) < 95 ]
```

The query requests the last name and the lowest pH value of any patient whose average oxygen saturation is less than 95 percent. Notice that the Termlist clause consists of two terms:

1. "last-name-of(pt)"
2. "MIN([value-of(lab) | (lab : pH) AND Patient-Lab(pt,lab)])."

The Predicate clause compares two terms:

1. "AVG([value-of(sat) | (sat : O2Sat) AND Patient-Lab(pt,sat)])"
2. "95"

The terms that appear as arguments to the aggregation functions MIN and AVG are both subqueries containing the free variable "pt." Therefore, for each binding of "pt," the values of the subqueries can be computed according to the meaning of subquery terms and the results can be aggregated according to the meanings of MIN and AVG. The subqueries, therefore, behave as any other terms (such as "age-of(physician-of(pt))") in that they contain variables whose values are specified by a particular binding and they evaluate to sets of entities. This property makes ReFER subqueries a powerful language construct that has no counterpart in SQL because SQL subqueries cannot be the arguments to aggregation functions [Date 1989a]).

Although certain ReFER constructs may be more expressive than their SQL counterparts, the true advantage of the ReFER query language with respect to sharing applications across heterogeneous databases is the language's abstractness. Abstractness allows one to translate queries formulated in ReFER into various equivalent SQL queries that are specific to different relational database implementations. I describe the means by which the TransFER methodology performs this translation in the next chapter.