A FORMAL MODEL FOR BRIDGING HETEROGENEOUS RELATIONAL DATABASES IN CLINICAL MEDICINE

A DISSERTATION
SUBMITTED TO THE PROGRAM IN MEDICAL INFORMATION SCIENCES
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

By
Walter Sujansky
January 1996

A FORMAL MODEL FOR BRIDGING HETEROGENEOUS RELATIONAL DATABASES IN CLINICAL MEDICINE

A DISSERTATION
SUBMITTED TO THE PROGRAM IN MEDICAL INFORMATION SCIENCES
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

By
Walter Sujansky
January 1996

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Russ Altman, Principal Advisor
(Section on Medical Informatics)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Jeffrey Ullman
(Department of Computer Science)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Mark Musen
(Section on Medical Informatics)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy

Arthur Keller
(Department of Computer Science)

Approved for the University Committee on Graduate Studies

Dean of Graduate Studies and Research

Abstract

Decision-support applications in medicine require patient-specific data to provide advice in the context of clinical cases. The emergence and expansion of on-line clinical databases provides the potential for decision-support applications to access patient-specific data automatically. Because the means by which data are represented and retrieved vary widely among clinical databases, decision-support applications that automatically access patient data currently cannot be shared among health-care institutions . The goal of my research has been to overcome the *representational heterogeneity* of clinical databases so as to provide medical decision-support applications with a *uniform interface* to patient-specific data. I developed and implemented a formal methodology, called TransFER, to bridge the representational heterogeneity of relational databases and to increase the portability of medical decision-support applications.

The TransFER methodology supports the definition of a global *reference schema* of clinical information against which applications may formulate requests for patient data. The reference schema is specified using a semantic data model, called FER, that is a hybrid of the functional data model ("F") and the entity-relationship data model ("ER"). The FER data model includes a declarative query language called ReFER. A formal *mapping language* based on the relational algebra allows database administrators to specify the correspondence between the global reference schema and the local relational database at their site. A *translating compiler* at each site uses the encoded mappings to translate automatically queries that are specified against the global model to semantically equivalent queries that can be executed by the local relational database .

I implemented a prototype of this model and conducted an experiment to test the validity and generality of the TransFER methodology. The experiment entailed that three groups design a relational database schema to represent a single clinical data set and that I specify a reference schema for the same data set using the FER data model. My hypothesis was that the TransFER methodology could map the FER reference schema effectively to the three independently designed relational database schemas and that the TransFER methodology could translate automatically queries formulated against the reference schema to queries that could be executed by all three schemas relational databases. The results of the evaluation indicated that (1) the FER data model and ReFER query language are well-suited for representing clinical database queries, (2) the mapping language is effective at mapping FER schemas to heterogeneous relational database schemas, and (3) the translating compiler is capable of correctly translating ReFER queries to equivalent SQL queries. The primary shortcoming of the TransFER approach is that automatically generated query translations are currently less efficient than manually generated translations, but the evaluation results indicate that formal query-optimization techniques that I have developed hold promise for eliminating this difference.

Acknowledgments

The ideas and work represented in this document would not have come about without the support of many people. My principal advisor, Russ Altman, guided me through the most difficult parts of this undertaking with his tireless intellectual and moral support. Russ's many insights into my work and endless creative suggestions helped me turn a collection of raw ideas into a scholarly document. Russ's spirited encouragement and fine sense of humor made it fun to do so. I am also indebted to Jeff Ullman, who contributed significantly to the quality of the technical content of this dissertation. Professor Ullman patiently assimilated and rigorously examined my ideas to help me place my work on a formal foundation that would have been otherwise unattainable. Mark Musen's valuable perspectives on the fields of medical informatics and artificial intelligence encouraged me to generalize and hone my research, and Arthur Keller's knowledge of database theory and database literature helped me place my work in an appropriate context.

Gio Wiederhold has contributed to my research ideas and my academic development from the time I began graduate school. Gio taught me much of what I know about databases, provided me with generous support, and extended me his friendship. He and Thierry Barsalou stoked my interest in database technology, and their innovative work on object–relational models influenced my own research. I was also fortunate to work with Peter Ludemann, who taught me the “art of Prolog” and saved me countless programming hours.

I am grateful for the superior research environment that was available to me at the Medical Information Sciences Program at Stanford. Through his vision and administrative wizardry, Ted Shortliffe has developed and sustained an academic program that has exceptionally talented people and the highest intellectual standards. I will surely miss the superb state-of-the-art computing environment that Tom Rindfleisch and his staff provided. My fellow students have been a consistent source of creative input and technical assistance, especially those who volunteered their time to design and build the databases that I needed to evaluate my work: Keith Campbell, Bill Detmer, Doug Fridsma, Vatsala Sarathy, and Huma Waheed. I still owe you pizza. Blackford Middleton and Doug Owens were also instrumental in formulating the clinical queries for the evaluation. Darlene Vian has been a dear friend and consistent supporter who has probably helped me in ways that I don't even know. I also appreciate the editing talents of Lyn Dupre and Debbie Fife, who taught me to express my thoughts succinctly and to reunite my infinitives correctly.

Finally, I must thank the special people in my life. I am grateful to my father for teaching me the value of hard work and perseverance. The camaraderie of Matt Daley always made the good times great and the difficult times tolerable. I have also been very lucky to have the affection and companionship of Ulrike; her patience, warmth, and cheerfulness have sustained me much more than she knows. And the greatest debt is to my mother, Eva, who has supported me in many ways, unwaveringly and unconditionally, throughout my graduate education and always. It is to her, with great love, that I dedicate this dissertation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Database Heterogeneity and Decision Support

The practice of clinical medicine demands that physicians perform complex information-processing tasks. To care for a single patient adequately, physicians must assimilate a large number of patient parameters, analyze those parameters in the context of their medical knowledge, build a mental model of the patient's condition to direct further diagnostic and treatment steps, and update this model continuously as the patient parameters change. Given that a typical physician cares for numerous patients concurrently, the information-processing demands of clinical medicine quickly approach the limits of human cognitive abilities.

Recognizing this problem, researchers in the field of medical informatics have devoted significant effort to the development of computational tools that can assist in or assume certain of the clinical information-processing tasks and thereby extend the abilities of physicians. One variety of these tools, *clinical decision-support (CDS) systems*, assists physicians by simulating the expertise of clinical consultants. Typically, these systems collect the clinical data associated with a single patient, process these data in the context of a large store of medical knowledge, and generate a recommendation, reminder, or critique intended to enhance the patient's clinical care. Although CDS systems vary widely with respect to the clinical contexts in which they are useful, the types of advice they provide, and the technical methods they employ, all share a common feature: Just like a human clinical consultant, a CDS system must acquire the data relevant to a specific patient before it can generate any advice. In colloquial terms, the system must first "learn the facts of the case."

The increasing prevalence of electronic medical record systems has created the potential for CDS applications to learn the facts of a case without human intervention. This mode of data access is often advantageous because it relieves users from tedious data entry and allows applications to operate quickly and unobtrusively. However, the complexity and heterogeneity of electronic medical record systems impedes the widespread use of CDS applications that access clinical databases automatically. Because the schemas, vocabularies, and query languages of clinical databases vary significantly across healthcare facilities, one must customize the database interfaces of CDS applications at each facility where the applications are deployed. The goal of my research has been to simplify and to standardize the way in which CDS systems acquire the facts of a case despite the complexity and heterogeneity of the electronic databases that store those facts. By providing all CDS systems with a single model of clinical data and a principled way to use that model to access heterogeneous clinical databases, I hope to eliminate the need to customize CDS applications, thereby facilitating their widespread deployment.

In this dissertation, I describe a novel technology that *automatically translates database queries to a variety of equivalent formulations* so that a single query can retrieve data directly from a variety of clinical databases. The technology includes an abstract database schema and a high-level query language that allows the developers of CDS systems to formulate queries without knowing the complex and heterogeneous features of specific databases. The key element of the technology is a mapping language that allows experts at each database site to encode a formal mapping between the *abstract database schema* that is used by CDS-system developers and the *implemented database schema* that describes the local database. A query-translation algorithm applies the mapping to automatically translate any query that appears in a decision-support application to an equivalent query that is specific to the schema, clinical vocabulary, and query language of the local database. I assert that the formal properties of this query-translation methodology distinguish it from existing methods for customizing database queries and establish it as an important new technology for sharing decision-support applications in medicine and other domains.

1.1 Expert Systems and Patient Data

Although medical informatics researchers have paid relatively little attention to the acquisition of patient data by CDS systems, the way in which a decision-support system learns the facts of a case has significant implications for that system's acceptance as a standard clinical tool and for its portability to multiple clinical environments. A comparison of two prominent CDS systems illustrates this point.

1.1.1 A case study

The MYCIN system [Buchanan and Shortliffe 1984] is a clinical consultation tool designed to assist in the diagnosis and treatment of patients with microbial infections before definitive culture results are available. MYCIN acquires all patient data by directly prompting the user of the system to enter specific information. For example, Figure 1.1 displays a sample dialogue between the MYCIN interface and a physician user.

What is the age of the patient?> **55**
Does the patient have a fever?> **Y**
What is the morphology of the organism?> **ROD** ...

Figure 1.1: A sample dialogue between the MYCIN system and a physician user. Each question is a query posed by the MYCIN interface, and each statement (in boldface) is a response entered by the physician.

MYCIN was designed to prompt for only those data that are contextually relevant and to avoid asking for the same information twice. Nevertheless, the system still requires that the physician user manually enter many facts of a case; often these facts are already recorded elsewhere, perhaps even in electronic form. Although formal evaluations proved MYCIN as proficient as infectious-disease specialists in generating differential diagnoses and formulating treatment plans [Buchanan and Shortliffe 1984], the system was never widely used in day-to-day practice. The data-entry burden that MYCIN places on users may be partially responsible for the system's lack of acceptance.

The HELP system [Kuperman, et al. 1991] is another clinical consultation tool; it is designed to identify and report aspects of a patient case that require attention, but that may be overlooked by caregivers. HELP and MYCIN differ in many ways, but one prominent distinction is that the HELP system acquires as many as possible of the patient data it needs from an online clinical database, rather than from the user. HELP collects patient data by issuing queries to which a clinical database can respond and by assigning the results of queries to internal variables that the HELP system can process. Figure 1.2 shows an excerpt from a HELP program that retrieves and subsequently processes the value of a patient parameter.

```
DATA:      recent_serum_sodium  = Query [ last(Serum_Sodium) ]

LOGIC:     If recent_serum_sodium > 155
             Then Alert_physician
```

Figure 1.2: Excerpt from a HELP system program. The first command instructs the system to retrieve the value of a patient parameter from the clinical database, and the second command instructs the system to compare the retrieved value to a reference value in order to reach a medically relevant conclusion.

In addition to eliminating unnecessary data reentry, automatic access to clinical data allows the HELP system to operate in the background, constantly assessing a patient's condition and level of care, and appearing only when a clinical recommendation is warranted. Although the HELP system has been in use at the Latter-day Saints (LDS) Hospital in Salt Lake City for over 20 years, and formal evaluations have proved it effective in improving care and reducing costs [Bradshaw 1988, Evans, et al. 1990], similar systems are in place today at only a handful of other institutions. One impediment to the dissemination of HELP technology is the difficulty of implementing a comprehensive clinical database. Perhaps a greater obstacle is the tremendous investment of resources required to develop, encode, and evaluate the complex clinical criteria that systems like HELP use to analyze patient data. This obstacle exists because the extensive set of clinical criteria implemented in the existing HELP system—despite being encoded in small, modular units and applying general medical knowledge—cannot be shared by any other institution without significant manual translation. This lack of portability results from strong assumptions encoded into the criteria regarding the implementation of the clinical database—assumptions currently required for HELP programs to access patient data automatically. These assumptions include the way the database is organized, the query language the database understands, and the terms used in the database to represent patient parameters. Because the assumptions are specific to the clinical database implementation at the LDS Hospital and are not likely to hold at other institutions, the encoded clinical criteria cannot be used elsewhere.

1.1.2 The data acquisition tradeoff

MYCIN and HELP are examples of technologies that, despite extensive development and proven clinical effectiveness, have not been deployed widely due (at least partially) to the way in which they acquire patient data. It is instructive to note that the data-acquisition weaknesses of each system are the strengths of its counterpart. MYCIN is less acceptable to physicians because it requires them to reenter data that are already recorded, a flaw overcome by the HELP system through its ability to retrieve data automatically from a clinical database. The HELP system cannot be used at most institutions, however, because, in order to automatically retrieve data, it must make restrictive assumptions about the way in which clinical databases are implemented, a limitation avoided by MYCIN through its reliance on the user for all data (which entails the less restrictive assumptions that the "database" speaks English and understands medical terminology). The two systems effectively illustrate the extremes of a tradeoff: automated data acquisition versus portability.

Although MYCIN and HELP have demonstrated some utility despite the limitations imposed by this trade-off, other kinds of CDS systems may be rendered completely infeasible. For example, several centralized agencies are currently developing standard protocols for directing cost-effective clinical care, which they intend to disseminate widely [Kelly and Swartwout 1990, VanAmringe and Shannon 1992]. Some investigators have suggested that decision-support systems could serve as a vehicle for the widespread distribution and unobtrusive application of these protocols [Bronzino, et al. 1989, van der Lei and Musen 1991]. Under such a model, standard protocols would be encoded as computer programs and downloaded from the publishing source to CDS systems at numerous provider sites. The CDS systems would subsequently execute the protocols based on local clinical data. Directed by the logic of a protocol, each CDS system would access patient data from the local clinical database and use it to determine the clinical context of a patient's case. Based on that context, the system would apply further protocol knowledge to generate advice regarding the diagnostic and therapeutic steps appropriate for the case. Without *both* the database-querying capabilities needed to establish clinical contexts automatically *and* the portability needed to implement the system at a variety of provider sites, the essential objectives of such a system would be compromised.

To make feasible future systems such as the protocol-dissemination tool, as well as to facilitate the wider use of existing systems such as MYCIN and HELP, the trade-off between data acquisition and portability must be eliminated. Developers of decision-support systems can overcome this trade-off only by designing their systems to query clinical databases in a manner that is independent of the databases' specific implementation features. Until now, accessing clinical databases in this manner has been impossible given existing technology. The fictitious scenario in the following section illustrates some of the difficulties.

1.2 Query Models

Sam is a clinical protocol designer at a large research institute near Washington, D.C. He is currently working on a protocol for diagnosing and managing patients with infectious gastroenteritis. The protocol consists of a flowchart that suggests diagnostic tests based on the nature and duration of symptoms, and that recommends therapeutic interventions based on the results of these diagnostic tests. Because all clinical protocols are distributed and applied via decision-support systems, Sam is encoding his protocol as a computer program, using a standard programming language for representing medical decision logic. The language has been adopted universally, so that the programs he writes can be run at any provider site in the United States, provided that the site stores patient data electronically.

One part of Sam's protocol checks whether a patient is taking antibiotics, because antibiotic use can predispose to gastrointestinal infections. If the patient is taking antibiotics, the protocol will recommend their discontinuation. Sam has almost completed this portion of the protocol and is looking forward to going home for the evening. All that remains is the formulation of a database query to inform the protocol whether the patient is, in fact, taking any antibiotics. Sam pauses for a moment to gather his thoughts before hitting the next keystroke.

1.2.1 Query-model components

What does Sam need to know to formulate this simple query? The answers to this question lie at the heart of the trade-off between data acquisition and portability. To formulate an effective query, Sam must know:

1. The abstract *model* of data representation that applies to the clinical database (for example, can the data be thought of as represented by unstructured text files, by uniform tables, or by a tree structure?)

2. The *schema* of the specific data that are represented in the clinical database (for example, are the names of patients and the names of the medications taken by those patients represented in a single file, which the query can directly access, or are they represented in different files, which the query must compare?)

3. The *language* for specifying queries that can be processed by the clinical database (for example, does the database require low-level commands that instruct it precisely where to find the names of all drugs associated with a patient name, or can it process high-level commands that only describe which patients and drugs to retrieve?)

4. The *format* of the data that are represented in the clinical database (for example, are the names of drugs represented as the full trade names used by clinicians, as the abbreviated names used by the hospital formulary, or as the numerical codes used by a national drug-coding system?)

Together, these four components constitute the *query model* that is used to formulate data-retrieval requests. Informally, a query model is the model of data storage and information retrieval that is in the mind of a database programmer when he encodes the conceptual notion of an information request into the executable commands of a formal query language. Figure 1.3 illustrates the components of a query model and how the components fit into a conceptual whole.

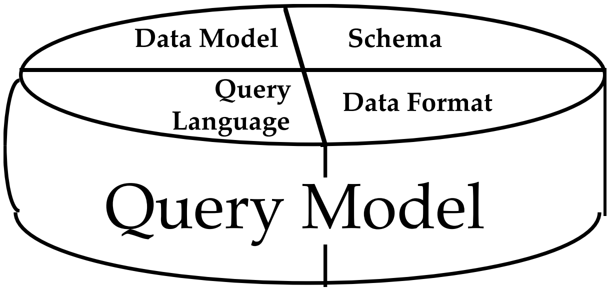


Figure 1.3: The components of a query model: an abstract data model for conceptualizing the data, a schema for organizing the data, a format for representing the data, and a query language for retrieving the data.

1.2.2 The query-model mismatch

Sam cannot formulate the query he has in mind without making a commitment to a specific query model, that is, without specifying each component of the query model he intends to use. However, the provider sites that will receive Sam's protocol have implemented their clinical databases using a variety of query models, which are likely to differ from Sam's query model and from one another. Although they all represent similar information, the clinical databases at provider sites implement different data models, assume different database schemas, process different query languages, and employ different data formats. Therefore, there necessarily will be a query-model *mismatch* at some sites when Sam's protocol is downloaded, and the mismatch may prevent Sam's protocol from accessing patient data correctly and from functioning as intended.

Until now, there has been only one way to bridge this mismatch and thus to confer automated data acquisition and portability upon Sam's protocol: For almost every provider site, the antibiotic-medication query had to be translated manually from Sam's query model to the query model of the local implemented database. Performing this translation entailed understanding the differences between the assumed and the implemented query models with respect to each of the four query model components, and translating all portions of the query such that the meaning of the whole was preserved. Depending on the degree of query-model mismatch, this process required careful, skilled, and extensive labor. Assuming N different query models, the translation needed to be performed N times by Sam prior to disseminating his protocol, or once at each of N sites when Sam's protocol was downloaded. In either case, the total workload was the same.

1.2.3 Bridging query models

Fortunately for Sam, he is using a new technology that allows him to encode database queries so that they can be translated *automatically* to a variety of query models. The technology available to Sam consists of three components:

1. An *abstract query model* that allows clinical data requests to be formulated in a general way, independent of the implemented query model at any provider site

2. A set of *mappings* that relates each element of a local implemented query model to its counterpart in the standard abstract query model

3. A *translating compiler* that uses the mappings to translate queries formulated in the abstract query model to equivalent queries in a local implemented query model

The abstract query model comprises the same four components as shown in Figure 1.3. The mappings encode the correspondence between the abstract query model and a specific implemented query model. These mappings are encoded such that a translating compiler can use them to translate any abstract query to an equivalent implementation-specific query. Figure 1.4 illustrates how a translating compiler uses the set of mappings to allow Sam's protocol to retrieve patient data from an arbitrary clinical database implementation.

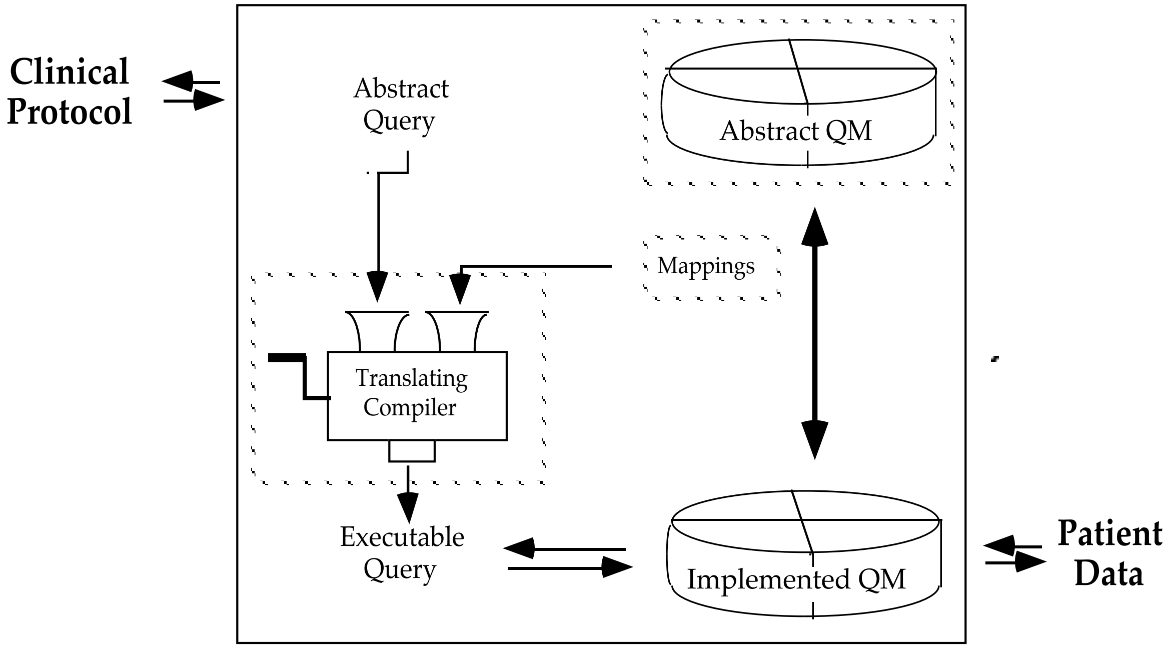


Figure 1.4: Schematic of the query-translation technology. A translating compiler uses encoded mappings between query models to translate any query formulated in the standard abstract query model (QM) to an equivalent query that can be executed by an implemented QM. The hatched boxes denote the three primary components of the query-translation technology.

Because Sam is using the abstract query model to encode his query, he does not need to make commitments to any implemented query model—commitments that will constrain the environments in which the query can be executed. The only commitments made in the abstract query model concern features of databases that are common to all the databases with which clinical applications may interface. For example, the model safely assumes that the concepts *patients*, *medications*, and *prescriptions* are represented in each clinical database (in some, probably implementation-specific, way). The model also assumes that a query to retrieve all the medications prescribed for a particular patient can be formulated in the query language of each clinical database (again, in some implementation-specific way). Sam is not reluctant to use a query model that makes such general assumptions about clinical databases, because he knows that each clinical database can satisfy them. In addition, Sam knows that there is a way for his abstract query to be translated at each client site into an equivalent query that is consistent with the implemented database there, and that returns the requested results to his protocol. The translation is possible because personnel at each site have specified and represented mappings between the query model implemented at their site and the standard query model that Sam's protocol understands. A translating compiler uses these machine-readable mappings to translate his queries automatically into equivalent queries that can be executed directly by the local clinical database. Sam does not need to worry about how the translation of his specific query is performed when he writes the protocol, and the administrators of each local database do not need to worry about how the translation is performed when they download the protocol. In fact, as long as each provider site keeps its mappings up to date, Sam can write and the site can download an arbitrary number of protocols with no additional overhead. Comfortable in this knowledge, Sam quickly encodes the antibiotic query and leaves for home.

1.3 Addressing the Problem: The TransFER Approach

The scenario described in Section 1.2.3 depends on the existence of two critical technologies:

1. A standard abstract query model that is sufficiently general to subsume implemented query models with respect to each of the query-model components, and that is sufficiently rich to include most of the modeling concepts, medical objects, and query operations needed by clinical decision-support systems

2. A model and a mechanism for translating automatically queries formulated with respect to the standard query model into queries consistent with an arbitrary clinical database implementation, such that the only inputs to the translation process are the query itself and a representation of the mappings between the two query models

The first technology allows clinical expert systems to specify the database queries that they require without limiting the database implementations that they can access. The second technology allows CDS systems to execute the database queries within them across environments with disparate database implementations by performing automated query translation. I define the *standard query model framework* as the research paradigm concerned with specifying a standard query model and with developing techniques to automatically translate queries based on this model. The specific goals of this framework are to develop more general and expressive abstract query models and more powerful query-translation methods. As these goals are realized, shared decision-support systems will be able to retrieve a wider array of patient data automatically from a wider variety of clinical databases, and the tradeoff between data acquisition and portability will be gradually eliminated. Figure 1.5 shows the logical conclusion of this evolution: a general technology that allows CDS applications to access patient-specific data in a uniform way from disparately implemented databases.

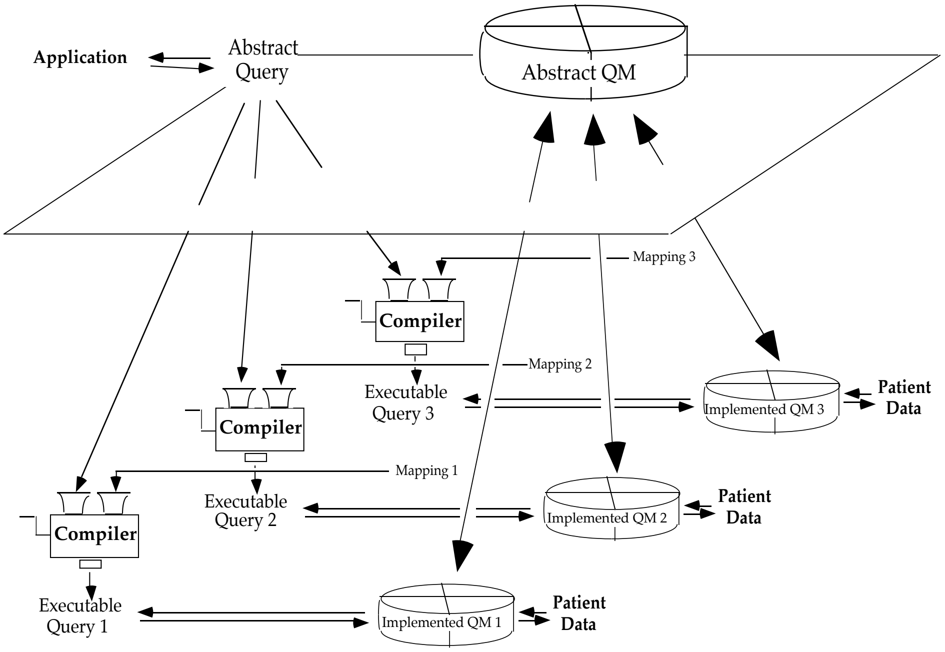


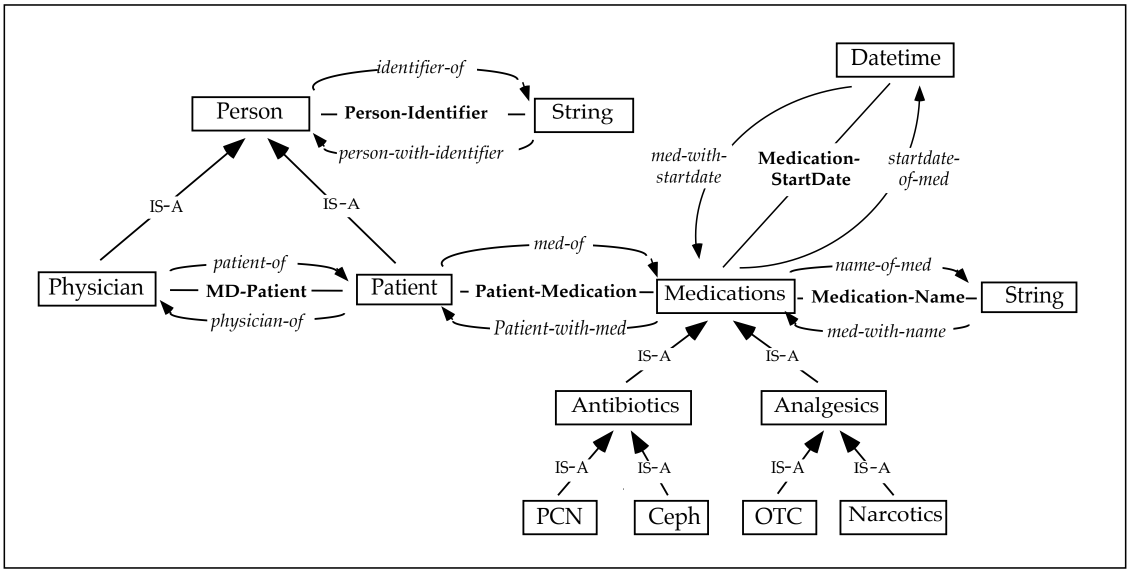
Figure 1.5: A model for supporting automated data access and portability. Application developers formulate queries using a common abstract query model. At each provider site, mappings between the local implemented query model and the abstract query model are entered into a translating compiler. The compiler converts the abstract queries appearing in the application program to executable queries that can be processed by the local clinical databases. Note that each site uses a copy of the same translating compiler.

I have developed an instantiation of the standard query model framework, called **the TransFER methodology**. TransFER provides access to heterogeneous databases implemented in the *relational* model by automatically translating queries formulated with respect to a standard query model to equivalent queries formulated in the relational query language SQL. Although the relational model is not ubiquitous among existing clinical databases, I have constrained the problem to relational databases because (1) the physical data independence provided by the relational model has steadily increased the use of this model in commercial database management systems, and (2) the well-studied mathematical properties of the relational model support a formalization of the query-translation process that is essential to its correctness and efficiency. If the TransFER methodology proves successful, I contend that nonrelational database sites will be no worse off than they are currently with respect to sharing decision-support applications, whereas the significant and increasing number of clinical database sites that have relational interfaces will benefit substantially from the automated query translation that TransFER provides.

The TransFER methodology consists of four general components:

1. A **semantic data model**, called FER (Functional Entity–Relationship model), supports the modeling of clinical data in an abstract fashion that can subsume many site-specific database implementations. The formal constructs of the FER model are entity types, relationship types, relationship functions, and IS-A connections. Figure 1.6 shows a representative FER schema for a small subset of the clinical domain.

2. A **high-level query language**, called ReFER, corresponds to the FER data model and allows developers of CDS applications to specify queries that are independent of any site-specific database implementation. A sample ReFER query appears in Figure 1.7. Note that this query is formulated with respect to the objects of the FER schema in Figure 1.6.


Person
Person-Identifier
String
identifier-of
person-with-identifier
IS-A
IS-A
Physician
MD-Patient
patient-of
physician-of
Patient
Patient-Medication
med-of
Patient-with-med
Medications
Medication-Name
String
name-of-med
med-with-name
Datetime
med-with-startdate
Medication-StartDate
startdate-of-med
IS-A
IS-A
Antibiotics
Analgesics
IS-A
IS-A
IS-A
IS-A
PCN
Ceph
OTC
Narcotics

the correspondence between the abstract FER schema and a specific relational implementation of it. For example, if each entity and relationship type happens to be modeled as a separate table in the relational database, then the mapping view for each entity and relationship type is simply the corresponding table itself:

view(Medications) => table(`PRESCRIPTIONS`)
view(Person-Identifier) => table(`MEDICAL_RECORD_NUMBERS`)
and so forth.

If there is not a one-to-one correspondence, however, the mapping expressions must combine or decompose tables to create views that accurately represent the intended semantics of the FER entity and relationship types. For example, if a database contains separate tables for inpatient and outpatient prescriptions, the mapping view for the entity type Medications (which represents both inpatient and outpatient prescriptions) must specify a *union* of the inpatient and outpatient tables to create a set that contains all of the prescribed medications represented in that database:

```
view(Medications) =>  table(INPATIENT_PRESCRIPTIONS)
                       UNION
                      table(OUTPATIENT_PRESCRIPTIONS)
```

The effect of specifying a union in the view is that any ReFER query that references the entity type Medications will be translated to a relational query that searches both the `INPATIENT_PRESCRIPTIONS` and `OUTPATIENT_PRESCRIPTIONS` tables.

4. A **query-translation module** performs the translation of ReFER queries by combining in a principled way the ERA mapping views that correspond to the FER objects appearing in each query. For example, translation of the query in Fig. 1.7 to a site-specific relational query entails combining the mapping views for Patient, Medications, Person-Identifier, Medication-Name, and Medication-StartDate as specified for a particular site. If the ERA mappings are correctly specified, *the resulting relational expression is semantically equivalent to the input ReFER*

query. The translation module subsequently optimizes and translates the expression into a site-specific database program consisting of one or more SQL statements. Figure 1.8 shows the automatically generated output of the TransFER query translator for the query in Figure 1.7, based on a relational database schema that was used for testing purposes.

```
Begin DB Program {

SELECT  ENTITY_TYPE_NAMES.ent_type_name
FROM    Therapy T1, Patient, Therapy T2, MEDICATION_CODES MC1,
        FER_TYPE_HIERARCHY FTH1, ENTITY_TYPE_NAMES, Therapy T3,
        MEDICATION_CODES MC2, FER_TYPE_HIERARCHY FTH2
WHERE   T1.start_datetime > "1995/10/28 0800" AND
        Patient.mr_number = "MRN-188640" AND
        (T1.mr_number = Patient.mr_number AND
         T1.mr_number = T3.mr_number AND
         T1.med_code = T3.med_code AND
         convert(varchar(50),T1.start_datetime) =
               convert(varchar(50),T3.start_datetime)) AND
        (T3.mr_number = T2.mr_number AND
         T3.med_code = T2.med_code AND
         convert(varchar(50),T3.start_datetime) =
               convert(varchar(50),T2.start_datetime)) AND
        (ENTITY_TYPE_NAMES.ent_type = FTH1.ent_type) AND
        (FTH1.desc_ent_type = MC1.ent_type) AND
        (MC1.med_code = T2.med_code) AND
        (FTH2.ent_type = "Antibiotics") AND
           (FTH2.desc_ent_type = MC2.ent_type) AND
        (MC2.med_code = T3.med_code))

} End DB Progam
```

Figure 1.8: Output of TransFER query translator. The query shown in Figure 1.7 is translated to this database program based on mappings specific to a relational database. The database management system of the relational database can directly execute the SQL query produced by the translation.

To summarize, sharing applications using the TransFER model entails the following steps:

1. A global reference schema of clinical data is specified using the FER semantic data model. The schema represents the medical concepts and the associations among concepts typically stored in clinical information systems.

2. Based on the FER reference schema, local database administrators use the ERA mapping language to specify mappings between each entity

type and relationship type of the global FER schema and the relational tables and fields of their local database schemas.

3. Developers of CDS applications (for example, the authors of clinical alert rules) formulate all data requests as ReFER queries against the FER reference schema.

4. When CDS applications are distributed to local database sites, the TransFER compiler automatically applies the local ERA mappings to translate the ReFER queries into site-specific SQL queries. These queries can retrieve the appropriate data locally when the applications are run.

In this dissertation, I formally present the elements of the TransFER methodology and I describe an evaluation of the methodology's practical utility. The evaluation is based on a prototype implementation of TransFER that automatically translates queries formulated in the ReFER query language to database programs that can by executed by heterogeneous relational databases. The results of the evaluation show that TransFER is an effective realization of the standard query model framework for databases implemented in the relational model. Specifically, the evaluation shows that

1. The FER data model and ReFER query language are sufficiently expressive to represent queries that typically appear in CDS applications

2. The ERA mapping language is sufficiently powerful to encode the mappings between FER schemas and heterogeneous relational database implementations

3. The TransFER compiler is sufficiently robust to translate ReFER queries into SQL queries that are site-specific and semantically equivalent.

1.4 Guide to the reader

In the remainder of this document, I describe in much more detail the research problem that I addressed, the query-translation methodology that I developed, and the practical evaluation that I performed.

In Chapter 2, I create a framework for my research by characterizing the types of applications that can benefit from a standard model of database access and by describing previous research related to the development of such a model. Chapter 3 introduces and formally defines the semantic data model and the high-level query language that I propose as the standard model of database access for shared applications. Chapter 4 describes the formal methods that I have developed to translate queries formulated with respect to the standard model of database access to equivalent queries that can retrieve data from a variety of legacy clinical databases. In Chapter 5, I present the results of an evaluation that I performed to assess the practical feasibility of TransFER as a model for sharing clinical decision-support applications across heterogeneous provider sites. Chapter 6 summarizes my work, discusses the implication of the evaluation, and proposes several areas of future research.

Chapter 2

The Shoulders of Giants

The task of providing medical decision-support applications with access to patient data that are stored in disparate clinical information systems is a necessarily complex undertaking. Clinical data consist of numerous, varied patient parameters that are richly interrelated. Clinical queries relevant to decision support may involve complicated search criteria and may return complex data objects. Implemented query models for clinical data vary along several dimensions. In order to develop a useful standard query model for medical applications, one must accommodate the complexity and heterogeneity inherent in the electronic representation of clinical data. Doing so requires an understanding of previous research results in medical informatics, in database systems, and in artificial intelligence. My objectives in this chapter are (1) to create a framework for my research by characterizing the kinds of applications that can benefit from it and by defining the terms and concepts that I shall use to describe it, and (2) to review previous research from a number of disciplines that addresses the same problems I am addressing from a different perspective or that constitutes a component of the solution that I propose.

Section 2.1 presents a short glossary of relevant terminology that appears in the remainder of the chapter. Section 2.2 briefly reviews several classes of decision-support applications in medicine, focusing on the various ways in which the applications acquire patient-specific data. Section 2.3 describes current research initiatives by the medical informatics community to standardize the representation of clinical data and to increase the portability of decision-support applications that require automated access to clinical data. Section 2.4 describes current research initiatives in the database and artificial intelligence communities that address the issues of standardization and interoperability that are relevant to medical applications, although in a domain-independent way. In Section 2.5, I outline the individual components required for the query model mapping technology that I introduced in Chapter 1, and I review previous research relevant to each of the components.

2.1 Terminology

The following glossary of database terms provides a reference point for further discussion. Although there may not be universal agreement regarding the definitions of these terms, the glossary was derived from several texts [Gardarin and Valduriez 1989, Ullman 1989, Wiederhold 1983] and represents a consensus among several database researchers.

2.1.1 Basic definitions

1. **Database**: A large, persistent, and shared representation of the data objects and of the relationships among data objects that are relevant to the functioning of some enterprise and that can be accessed by a computer.

2. **Database Management System (DBMS)**: A software system that reliably manages a database and provides access to large collections of data objects efficiently. Note that this is a very generic definition: One may describe everything from the Unix operating system to an object-oriented database with a natural language interface as a DBMS.

3. **Database System**: A database and a corresponding DBMS.

4. **Data Model**: A set of constructs with associated composition rules used to specify the structure of and the constraints upon data in a database. A data model provides an *abstract mathematical formalism* for describing data, which allows users to see information not as raw bits, but in more abstract and intuitive terms.

5. **Data Definition Language (DDL)**: A specification language that allows users to describe the data in a database using a specific data model. Data definition languages are used to specify database schemas (see glossary item 6).

6. **Database Schema**: A complete specification of the object types relevant to an enterprise, expressed using a particular DDL. The information captured in a database schema depends on features of the data model in which it is specified. Specifically, a database schema is *not* necessarily a relational database schema, but may be specified with respect to data models that are more or less expressive.

7. **Data Manipulation Language (DML)**: A language interface provided by the DBMS to specify operations on the database, such as queries and updates. DMLs correspond to particular data models and, as such, may vary in expressiveness and level of abstraction.

2.1.2 Levels of database description and data manipulation

Database systems may be described and manipulated at different levels of abstraction. For example, a database system may have a low-level query language that specifies in assembly language the physical locations of requested data, or a high-level query language that specifies in first order logic the abstract properties of requested data. Different levels exist for two reasons:

1. Humans and computers require different degrees of detail in describing and manipulating data.

2. Higher levels of abstraction provide *independence* from lower-level implementation details that may change with time or may vary across database systems.

Four levels of database description typically exist [Batini, et al. 1992, Sheth and Larson 1990]:

1. *External level*: A description of a *subset* of the database corresponding to a program's or a user's view of the enterprise. An external view typically is available to “client” applications as a circumscribed window on the enterprise data. For example, the portion of a clinical database used by the pharmacy staff comprises an external-level schema of the entire clinical database.

2. *Conceptual level:* An *implementation-independent* description of the entire database corresponding to a global view of the modeled enterprise. A database schema specified at the conceptual level is a *conceptual schema.* For example, a data description of the entire patient record specified using an abstract data model would constitute a conceptual schema. Conceptual-level description hide many implementation details that are of no interest to the users of a database system.

3. *Logical level*: An *implementation-dependent* description of the data that can serve as the interface to an operational database system. A database schema specified at the logical level is specific to the data model used by the DBMS at a particular site and is called a *logical schema*. Importantly, a single conceptual schema may be represented by various logical schemas.

4. *Physical level*: A *representation-dependent* description of the database that corresponds to the physical storage structures and access methods used to store and manipulate data in secondary memory. Data descriptions and data manipulations specified at the logical level typically are translated by the DBMS to corresponding representations and operations at the physical level, so that they can be efficiently implemented and executed by the operating system.

Having established these terms, I now can describe previous research in sharing decision-support applications across heterogeneous database systems.

2.2 Accessing Clinical Data

A review of prominent medical expert systems developed over the past 20 years reveals that decision-support applications acquire patient-specific data in one of four ways:

1. *Via direct interaction with the user*. The system prompts the user for all relevant clinical parameters, sometimes offering synonym dictionaries and phrase-completion facilities to assist in the process. This is especially the case for diagnostic systems that require the entry of complex clinical findings. Examples of such "input-driven" systems are MYCIN [Buchanan and Shortliffe 1984]nd QMR [MILLER AND MASARIE

1990]. A notable feature of input-driven systems is that no patient-specific data are stored between consultation sessions.

2. *Via the loading of special-purpose data files*. The system initially captures patient data via user interaction and stores the data in specially formatted files that are managed by the operating system. During subsequent consultation sessions, the patient-specific files are loaded into the memory space of the application and inference proceeds. An example of such a "file-driven" system is ONCOCIN [Shortliffe 1981]. A notable feature of file-driven systems is that the stored patient data cannot be shared easily among other applications and that all queries and updates are performed by the decision-support system.

3. *Via access to an online clinical database*. The decision-support system queries a general-purpose database system that captures, stores, and retrieves clinical data. Typically, the clinical database supports a number of enterprise-wide functions, including direct patient care, administration, and clinical research. Queries to the database are executed via an application programmer interface (API) that is specific to the clinical database system and that is available to all client applications. Examples of such "database-driven" systems are clinical reminder and alert applications integrated with the following clinical information systems: HELP [Kuperman, et al. 1991], COSTAR [Barnett, et al. 1982], TMR [Stead and Hammond 1988], and the Regenstrief Medical Record System [McDonald 1984]. A notable feature of database-driven systems is that the APIs used to access patient data are complex and very specific to the local clinical database implementation [Adams 1986]; as a result, decision-support applications cannot be shared across multiple databases because each database's API is significantly different. These applications suffer from the portability problem discussed in Chapter 1.

4. *Via access to abstracted data sets*. The system queries a longitudinal database that stores clinical data abstracted from the medical record systems of numerous provider sites. The data usually are related to a single clinical syndrome, such as rheumatism or coronary artery disease. Data can be pooled from multiple sources because a common

> set of clinical parameters and a common descriptive vocabulary are specified *a priori* and are enforced during the data-abstraction process. An example of such a “databank-driven” system is ARAMIS (The American Rheumatism Association Medical Information System) [Blum 1981]. A notable feature of databank-driven systems is that, although they can access data that are primarily stored in a variety of medical record systems (including clinical databases), the data must first be abstracted into a uniform and standard representation, a generally laborious process.

A standard query model of clinical data and an automated query translation mechanism for clinical queries would benefit decision-support systems in each of the four categories by providing a direct, programmatic, and uniform interface to various clinical database implementations. Input-driven systems could acquire data directly from electronic medical records rather than requiring users to enter all patient parameters manually. File-driven systems could delegate certain data-manipulation tasks to database management systems, which are typically optimized to perform complex operations upon large data sets. Database-driven systems could access online databases using a standard API, facilitating their porting to clinical environments that have different database implementations. Databank-driven systems could exploit the query translation mechanism to pool data from disparate sources without laborious manual abstraction. In order to realize these benefits, of course, the abstract query model must be sufficiently powerful to provide decision-support systems with access to all data that they require, and the query-translation mechanism must be sufficiently sound to ensure that data retrieved from different sources have the same meaning.

2.2.1 Clinical practice guidelines

There is a class of the database-driven systems that can benefit particularly from a standard query model and an automated query-translation mechanism. These systems are applications that implement clinical practice guidelines. Clinical practice guidelines (CPGs) are systematically developed statements to guide practitioners' decision making in specific clinical circumstances [Field and Lohr 1992]. The intent of CPGs is to improve the quality and to control the costs of health care by specifying *protocols* for providers to follow in managing particular classes of patients; protocols may include the sequence in which actions should be taken and the clinical criteria for various medical interventions. Protocols can improve the quality of care by recommending medically warranted actions that physicians have overlooked, and protocols can control the costs of care by advising against medically unnecessary or inappropriate actions that physicians have ordered. CPGs are typically formulated by central, authoritative bodies based on analyses of medical literature, and subsequently are made available for distribution to practitioners. For example, the American Medical Association recently published a directory of 1500 CPGs developed by more than 45 physician organizations and other groups [Toepp and Kuznets 1993].

Observers have remarked that computer technology may play an important role in the dissemination and implementation of CPGs by allowing CPGs to be incorporated into the routine information management tasks of physicians. For example, the Committee on Clinical Practice Guidelines of the Institute of Medicine (IOM) has recommended the incorporation of CPGs into computerized decision aids, asserting, "Information and decision support systems are crucial elements in long-term strategies for promoting the application of guidelines. As information and decision-support strategies are used in tandem with other kinds of implementation approaches, they can simultaneously present information and shape and control its uses." [Toepp and Kuznets 1993]. As discussed in Section 1.1.2, however, centrally formulated CPGs can be disseminated conveniently via computer applications only if the applications have *both* the database-querying capabilities needed to establish clinical contexts automatically *and* the portability needed to implement guidelines at a variety of provider sites. The IOM has recognized portability as an important factor in the electronic application of CPGs and has called for standards to permit guidelines to be used on many different types of computer-based equipment and systems. The standards identified by the IOM include "rules of syntax, database structures, and communication links that support multiple settings of care" [Dick and Steen 1991].

2.3 Clinical Data Standards

Consistent with the needs recognized by the IOM, a number of national organizations already have developed or are currently developing standards for the representation of clinical data and decision-support knowledge. These standards bodies include ASTM, HL7, IEEE and ACR/NEMA [McDonald, et al. 1991]. Most standardization initiatives focus on the specification of standard message formats and communication protocols for the electronic interchange of clinical data and on the development of standard programming languages for the electronic interchange of decision-support logic.

2.3.1 Medical data interchange standards

The HL7 message standard [HL-7 Working Group 1989] specifies the syntactic structure of ASCII messages that transmit data regarding medical events, such as admissions, orders, and laboratory results, among clinical information systems. The HL7 standard defines the types of messages that may be transmitted, the records that make up each message type, and the set of data elements in each record. For example, the message specification for "Lab Results" consists of a header record signifying the identity of the patient being tested, followed by multiple detail records, each consisting of the data elements TEST NAME, TEST RESULT, and TEST RESULT UNITS. Although HL7 has been adopted by over 100 commercial vendors as a standard format for the exchange of clinical data, it is strictly a communication protocol and is inadequate as the foundation for a standard query model of clinical information. Specifically, HL7 lacks a *static* conceptualization of clinical data that specifies the set of clinical data objects and the relationships among the data objects in a manner that is independent of any *dynamic* data transmission [McLinden, et al. 1990]. For example, HL7 includes no explicit representation of the relationship between a lab test order and the lab test result corresponding to the order; *HL7 only specifies the syntactic positions of the two objects in a transmitted message*. As a result, the applications that send and receive HL7 messages must be aware of the meanings of and the relationships among data objects, which creates the potential for inconsistent and incorrect interpretations. In order to provide a uniform query interface to clinical information systems, a clinical data standard must explicitly and formally represent the set of data objects and the relationships among the data objects that are represented in clinical database systems; otherwise, there is no way to ensure that communicating applications agree upon the semantics of exchanged data.

The IEEE Medical Data Interchange Standard (MEDIX) [Harrington, et al. 1990], another model for data exchange between automated healthcare systems, addresses the need for a static conceptualization of clinical data by specifying a *reference model* of the health care environment. The MEDIX reference model provides an object-oriented conceptual schema of clinical data objects and their relationships that can be used by applications to confer unique meanings to transmitted messages [Ostler, et al. 1990]. In order to provide a common schema of clinical data across all health care systems, the reference model is specified at an abstract, implementation-independent level. For example, the objects specified in the reference model include patients, physicians, and services provided. It is only within implemented information systems that the conceptual reference model is mapped into specific hardware and software realizations or computational models. The MEDIX approach to defining clinical data standards is compatible with the requirements for application portability that were articulated by the IOM committee on clinical practice guidelines (see Section 2.2.1) because the MEDIX reference model provides a standard, abstract view of the structure and the meaning of clinical data that is independent of specific clinical database implementations. The encapsulation provided by the object-oriented reference model dictates that the implementation of data objects be hidden from client applications and that all operations upon data objects, including queries and updates, be specified by sending messages to abstract objects of the reference model. According to the MEDIX architecture, an “agent process” must exist at each database site that transforms the object-based messages sent to the reference model to non–object-based operations, such as SQL queries, that are supported by local database implementations. The MEDIX architecture represents a powerful and general step toward providing decision-support applications with uniform access to patient data across database implementations. However, before application portability can be realized, there must exist a methodology for conveniently specifying and performing the *transformation* of MEDIX-level operations to local database operations. Such a methodology, which has not been addressed by the MEDIX standards committee, will be instrumental in providing MEDIX interfaces to disparately implemented clinical databases. The development of this methodology is the focus of my research.

2.3.2 Medical knowledge-representation standards

In addition to standards for the representation and communication of clinical data, the development of portable clinical practice guidelines requires standards for the representation of the medical logic that evaluates data. A recent initiative toward standardizing the representation of medical decision-support logic is the Arden Syntax [Hripcsak, et al. 1994]. The Arden Syntax is a programming language that represents medical knowledge in discrete, independent programs called Medical Logic Modules (MLMs), which query a database for relevant clinical data and subsequently process the data in order to generate advice based on encoded medical logic. Typically, MLMs generate alerts, reminders, and critiques for providers based on the automated recognition of noteworthy patterns in a clinical database. The paradigm of pattern recognition and alert generation is appropriate for implementing certain clinical practice guidelines.

Although other languages exist for encoding data-driven medical knowledge [Adams 1986], these languages are specific to the particular clinical computing environments in which they were developed. The Arden Syntax is distinct in that it is specifically designed with portability in mind, that is with the ability to be implemented in numerous, diverse clinical computing environments. Portability is achieved by standardizing the constructs for representing medical logic, such as data types and control structures, and by *underspecifying* the constructs for formulating database queries, such as database structure and query language syntax. The designers of the Arden Syntax intentionally chose to underspecify the query constructs because, after reviewing a number of clinical information systems, they concluded that no single syntax for database queries could be devised that would be general enough to accommodate the diversity in terminology, in database structure, and in query languages that exists among clinical information systems [Hripcsak, et al. 1994]. Rather than attempting to specify a complete query syntax, therefore, the designers of Arden specified a syntax for only those constructs determined to be common to many clinical information systems. These constructs consist of aggregation operators and temporal constraints (see Figure 2.1). The remaining constructs required to completely specify database queries are encoded informally at the time an MLM is authored (usually in natural language), and are implemented in a formal manner that is consistent with the query language, database structure, and medical terminology of a specific institution only later. As a result, the Arden standard specifies the division of queries into two disjoint components: an institution-specific component, which is informally specified and must be *manually* translated into the formal syntax of an institution's local query language, and a more general component, which is formally specified in the Arden Syntax and must be *merged* with the institution-specific component during compilation of MLMs. The syntax and semantics of the query model are described elsewhere [American Society for Testing and Materials 1992].

Figure 2.1 shows a database query formulated in the Arden Syntax. The query is intended to retrieve the name and dosage of the diuretic medication prescribed to a patient most recently within the last week.

```
(drug, dose) := read last { select Drug_name, Drug_dosage
                            from MEDICATIONS
                            where Patient_name = "Williams" AND
                                  Drug_Class = "Diuretic"}
                where it occurred within the past 7 days
```

Figure 2.1: A database query encoded in the Arden Syntax The text appearing within the braces represents the institution-specific component of the query, in this case specified in SQL. The boldfaced text represents the formal Arden constructs of the query, consisting of the aggregation operator "last" and the temporal constraint "where it occurred. . ." (boldfaced for clarity only).

Note that, although the institution-specific component of the query in Figure 2.1 is specified in SQL, this is an arbitrary (albeit convenient) notation. Another author could specify the institution-specific logic of the same query using natural language or machine code. These alternative representations all would conform to the Arden standard, because the institution-specific component of a database query is completely outside the scope of the Arden Syntax. While this feature is an effective short-term expedient to accommodate the diverse database implementations and medical vocabularies in use at various sites, an informal query model actually compromises the portability of MLMs in the long run. Without a standard database schema and query syntax, the query model can have no well-defined semantics. Without precise semantics, database queries are interpreted subjectively at "client" institutions and may be implemented inconsistently when the queries are manually translated. A codeveloper of the Arden Syntax remarked upon this difficulty following a recent experiment in sharing Arden programs:

> "Although standards for representing clinical decision logic can be of great assistance in sharing the work of many, sharing may be delayed until common standards exist not only in the description of the logic, but in all aspects of the medical information system" [Pryor and Hripcsak 1994].

Furthermore, without a standard database schema and query syntax, there can be no formal foundation upon which to build compilers to automatically transform MLM queries to data-manipulation operations that can be executed by diverse clinical information systems. Thus, although the Arden standard provides a uniform syntax for encoding the medical *logic* that makes up clinical practice guidelines, it does not provide a uniform model of medical *data* that would allow MLMs to interface with diverse clinical information systems in a consistent and portable manner. Until standard models of clinical data and of clinical-data retrieval are incorporated into languages for sharing medical logic, the portability envisioned by the IOM for electronic CPGs will not be realized.

As demonstrated by the advent of HL7, MEDIX, and Arden, standards bodies in medical informatics are specifying *parts* of the standard environment for developing portable decision-support applications, but the parts also must be coordinated and integrated before a functioning environment can exist. Specifically, standard languages for encoding decision-support logic, such as the Arden Syntax, must be integrated with standard models for representing and retrieving clinical data, such as the MEDIX reference model. In addition, powerful and sound methodologies must be developed to translate automatically standard representations of database queries into implementation-specific query operations so that diverse databases sites can conveniently and cost-effectively integrate portable decision-support applications into their local environments.

2.4 Contributions of Database and Artificial Intelligence research

Clinical medicine is one of many domains that will benefit from the development of standard representations for enterprise data and decision logic. Accordingly, domain-independent research in the fields of database systems and artificial intelligence (AI) has addressed the sharing of data and knowledge among independent software agents. In the field of database systems, this research has focused on the development of heterogeneous database systems, and in the field of artificial intelligence, this research has focused on the development of sharable knowledge-representation formalisms. It is instructive to review research results from both fields when evaluating the technology available to support portable knowledge-based applications in medicine.

2.4.1 Heterogeneous database systems

Heterogeneous database systems (HDBSs)[1] are computational models for the integration and aggregation of data from disparately implemented databases [Sheth and Larson 1990]. For example, an HDBS might provide uniform access to electronic patient records in a hospital computing environment that uses a MUMPS hierarchical database for storing patient demographic data and a Sybase relational database for storing patient laboratory results. HDBSs are sometimes confused with distributed database systems (DDBSs) because both provide a unified view of and common interface to data that is physically stored in different locations. However, DDBSs are much more integrated and coordinated than are HDBSs. Typically, the constituent databases of a DDBS implement the same data model and query language and run the same distributed database management software. Also, the fragmentation of data in DDBSs is usually *induced* to reap the efficiency and autonomy advantages of distributed computing. The constituent databases in HDBSs, in contrast, existed prior to the establishment of the HDBS and are coordinated much more loosely. The HDBS model, therefore, is more applicable than the DDBS model for providing medical decision-support applications with a uniform interface to disparate clinical

---

[1] The terms "heterogeneous database systems" and "federated databases" are used synonymously in database literature.

databases. Specifically, HDBSs have the following characteristics [Sheth and Larson 1990]:

1. *Representational heterogeneity*. The constituent databases in an HDBS may use different data models, different query languages, different terminologies, and different schema structures to represent the same real-world semantics. In other words, although the data stored at multiple sites may have identical real-world semantics, the data representations and the data-access methods at the sites may differ.

2. *Local autonomy*. Each constituent database has the right to control access to its own data by the HDBS and it has the ability to access and manipulate its own data independently of the HDBS. Most decisions regarding the representation and manipulation of data are made by local database administrators to accommodate local system needs (such as functionality, performance, and cost). The membership of the database in a heterogeneous database system is incidental to the primary purpose of the database.

3. *Bottom-up integration*. Whereas DDBSs *induce* the distribution of data that were previously integrated to achieve efficiency benefits, HDBSs *integrate* data that were previously distributed to achieve interoperability benefits. Bottom-up integration implies that the HDBS must provide an interface to diverse, preexisting information systems without requiring extensive modifications of preexisting software.

These three features describe accurately the situation faced by developers of portable decision-support applications that require access to clinical data at multiple provider sites: Preexisting clinical database implementations are extremely diverse, are maintained to meet local functionality requirements, and are very costly to modify. In this environment, the goal of an HDBS is to provide *database transparency* to portable applications, that is, to provide a global and consistent database interface for applications, as if the data were not distributed and all of the database management systems were of the same type. HDBS research is predicated upon the belief that heterogeneity at the level of constituent database systems will (and should) persist, despite standardization efforts. This certainly is a realistic model for clinical databases in the United States in the foreseeable future.

To provide database transparency in the face of representational heterogeneity, HDBS researchers have focused upon several research issues that are relevant to the development of a uniform query model of clinical information. These issues include:

1. *Common data models and common query languages*. The specification of a single data model to represent the global schema of distributed data and of a single query language to specify site-independent operations upon distributed data are essential to providing database transparency to applications that use HDBSs. Experimental HDBS systems typically have used simple semantic data models and set-oriented query languages, such as the functional model and the DAPLEX query language [Smith, et al. 1986] and the entity–relationship model and the GORDAS query language [Ceri and Pelagatti 1984]. Simple data models that represent "atomic" facts, such as entities and the relationships among entities, are easier to translate to the more complex data models that are implemented by constituent database systems. Set-oriented query languages, such as the relational algebra, provide greater potential for query decomposition and query optimization than record-at-a-time languages, such as CODASYL. I have applied both of these principles in my work.

2. *Schema integration*. Database research on integrating the schemata of constituent database systems into a global schema have yielded valuable insights into the resolution of representational heterogeneities among data in different systems [Agarwal 1991, Ceri and Pelagatti 1984]. Techniques for integrating inconsistent schemas have included the definition of auxiliary data structures to specify mappings across naming conflicts or units-of-measure conflicts; these types of mappings are a necessity for HDBSs in the clinical domain, where numerous competing coding schemes are used to record clinical data.

3. *Query translation*. HDBS researchers also have explored techniques to conveniently and faithfully translate queries formulated in a global

data-manipulation language to equivalent queries formulated in the data-manipulation languages of constituent databases. These techniques typically use mapping information that encodes the correspondence between the global schema and a constituent database schema. For example, this technique has been used in the MULTIBASE query translation facility [Smith, et al. 1986].

4. *Query optimization*. Because the global data model and global data-manipulation language may be more powerful than their counterparts in constituent database systems, additional processing that is external to the local database system may be required to execute global queries. For example, the generalization and specialization constructs available in the functional data model have no counterparts in the relational model used by many clinical database systems. Operations to implemente the generalization/specialization constructs must be specified outside of the database query language and integrated with the queries in an efficient manner. HDBS research has addressed the optimization of such hybrid queries.

Although research in heterogeneous database systems has been ongoing for over a decade and is addressing many of the issues relevant to database interoperability in the medical domain, prominent standardization efforts in medical informatics, such as the MEDIX reference model and the Arden Syntax, have not adopted explicitly the HDBS paradigm or HDBS research results. My own research is based closely upon the results in global data-model development, schema integration, and query translation that originated in HDBS research, as I describe in Section 2.5.

2.4.2 Standards for knowledge representation

Researchers in knowledge-based systems have recently addressed the issues of sharing and reusing domain knowledge among knowledge-based applications to reduce the development costs of such applications [Musen 1992]. The challenges inherent in this objective and the techniques under development to achieve it also are relevant to the task of sharing decision-support applications that require access to clinical databases.

The largest project in this field, the Knowledge-Sharing Effort (KSE), is a collaborative research project to identify and address impediments to the sharing of knowledge among applications [Fikes, et al. 1991]. The specific objective of the effort is to find ways of preserving existing knowledge bases (KBs) so that they can be shared, reused, and built upon by many applications. The central problems addressed by this research are the heterogeneity of knowledge-representation (KR) languages and the heterogeneity of domain terminologies.

The heterogeneity of knowledge representation languages prevents KBs from being shared and reused because the syntax used to represent domain knowledge in one application environment may be completely meaningless within another environment, although the semantics of the represented domain knowledge are similar. The KSE is addressing this problem in two ways: (1) the definition of a *standard* knowledge-representation language, called the Knowledge Interchange Format (KIF) [Genesereth and Fikes 1991], to encode knowledge in an abstract, implementation-independent way; and (2) the development of tools to translate KIF knowledge bases into the knowledge representation languages of various application environments [Fikes, et al. 1991].

As with HDBS research, the work of the KSE is predicated upon the assumption that diversity among knowledge-representation languages will and should persist, which implies that interoperability will depend upon techniques to faithfully translate encoded knowledge among disparate systems.

The second problem addressed by the KSE is the heterogeneity of domain terminologies. Heterogeneity among the terms used to denote and relate objects in the domain of discourse prevents KBs from being shared and reused because different knowledge-based applications may disagree upon the meanings of symbols (terms) that appear in shared KBs; the applications may, therefore, interpret encoded knowledge inconsistently. The inconsistent use of a shared KB implies that at least one application using the KB is interpreting the knowledge incorrectly. Inconsistencies in the interpretation of shared knowledge will occur if applications associate the same term with different meanings or use different terms to denote the same meaning. The KSE is addressing this problem by developing a *standard* language to represent domain terminologies (that is, to represent the set of applicable terms and their meanings) and a method of translating terminologies encoded in the standard language into various KR languages. Also, the KSE is developing a set of tools to verify the consistency between the domain terminology appearing in a shared KB and the domain terminology appearing in an application that will use the shared KB [Fikes, et al. 1991].

Of the two problems being addressed by the KSE, the representation of domain terminologies in a standard, implementation-independent way and the translation of domain terminologies to specific application environments are the most relevant to the objective of providing decision-support applications with a uniform interface to clinical databases. Domain terminologies for knowledge-based systems are quite similar in content and in purpose to conceptual schemata for database systems. Specifically, a domain terminology provides a *conceptualization* of the entities and of the relationships among the entities that may exist in some universe of discourse [Fikes, et al. 1991]. In the field of artificial intelligence, such a conceptualization is called an *ontology*. The standard ontologies envisioned by the KSE define sets of terms that denote classes, relations, functions, and object constants for some domain of discourse. Standard ontologies also include a logical axiomatization to constrain the interpretation of the terms [Gruber 1993]. By this definition, an ontology is similar to a query model in that both consist of the following components:

1. A formal abstract model for representing the properties of objects in a domain

2. A definition of the objects classes and of the relations and functions that may be defined over the members of those classes in a particular domain

3. A specification of the object constants that may be members of the defined object classes

If an ontology also includes a formal query language, then the ontology is equivalent to a query model, as defined in Section 1.2.1.

In the KSE, ontologies are translated from the standard ontology language, which is based on KIF, to the various KR formalisms of implemented knowledge-based environments, such as LOOM [Yen, et al. 1991]. Ostensibly, the translations preserve the semantics of the ontologies. Whereas the KSE is interested in translating between standard domain ontologies encoded in a variant of KIF and various implemented KR formalisms, HDBS researchers are interested in translating between standard conceptual database schemas, such as the MEDIX reference model, and the logical databases schemas of implemented database systems.

Given the similarities in content and in purpose between domain ontologies and conceptual database schemata, at least two results from the KSE's research on ontologies are applicable to the development of environments for portable decision-support applications in medicine: ontology design criteria and ontology translation methods.

Gruber has specified a set of design criteria for ontologies that are intended to support knowledge sharing and interoperability based on a shared conceptualization [Gruber 1993]:

1. **Clarity**. An ontology should effectively communicate the intended semantics to humans who design applications based on the ontology.

2. **Coherence**. An ontology should be internally consistent.

3. **Extensibility**. An ontology should support the addition of new terms (to support the needs of a specific site or when general domain knowledge increases) without requiring revisions of the existing definitions (and, by implication, of applications that use those definitions).

4. **Minimal encoding bias**. An ontology should be specified at a sufficiently abstract level to enable its incorporation into agents that may be implemented in different representation systems and styles of representation.

5. **Minimal ontological commitment**. Because an ontology serves a different purpose from that of a knowledge base, it should include the *minimal* set of domain knowledge to support the intended knowledge-sharing activities, that is, it should make as few claims as necessary about the world being modeled.

The methods used to translate standard ontologies into the specific KR formalisms of implemented knowledge-based systems also may be applicable to the translation of standard database queries into the specific query languages of implemented clinical database systems. The KSE translator operates much like a standard programming language compiler in that it has a *recognition component* to parse sentences based on a specification of the (standard) input grammar and a *generation component* to generate translated ontological statements from the internal representation of the input stream. Because a transformation must take place between the input and output ontologies, the KSE translator also has a *reformulation component* that applies rewrite rules to an internal representation of the parsed input to create a new internal representation that is consistent with the representational formalism of the desired output [Fikes, et al. 1991]. The reformulation component is a specialized type of theorem prover that derives output statements that are *equivalent* to the input statements, rather than implied by them. In this sense, it is comparable to standard rewrite systems, such as attribute grammars [Knuth 1968], that apply semantic rules to generate translated outputs. The reformulation component is distinct from attribute-grammar methods, however, in that it is goal-directed and applies heuristics to guide the choice of rewrite rules. The reformulation process, therefore, entails a search of the space of possible transformations to effect the desired translation, with possible backtracking. This model of ontology translation is also used by the SIMS system [Arens, et al. 1993], a prototype heterogeneous database system that applies rule-based techniques to integrate database schemas and perform query translation. Attribute grammars, conversely, support the *deterministic* modeling of transformation rules such that the rules always are applied in a predetermined sequence, which avoids heuristic search and backtracking. I have chosen attribute grammars as the formalism for representing and executing the translation of queries because deterministic modeling renders the translation process more efficient. However, it may be instructive later to compare the efficiency and effectiveness of the attribute-grammar and rule-based models for rewriting queries, and it may be useful to apply rule-based methods to optimize translated queries.

2.4.3 Combining database and knowledge-based research

Although database and AI researchers often have different perspectives on similar problems, the challenge of providing decision-support applications with a uniform interface to diverse database systems constitutes a problem that applies to both research fields. The fundamental basis for this duality is the fact that database systems *are* knowledge-based systems: Database systems represent knowledge about the world by implementing some computational model that supports both the storage of data items and the description of the stored items. This definition applies equally well to what AI researchers call domain ontologies. Sowa asserts that the primary difference between knowledge-based and database systems is the ratio of the number of data descriptors to the number of stored data items: database systems store many repetitions of a small number of simple data types, whereas AI systems have unique and complex descriptions for almost every data item [Sowa 1984] Of course, AI systems also provide complex inference techniques that allow data processing far more sophisticated than that supported by database query engines, but the inferences performed by knowledge-based and database systems differ more in sophistication than in nature; both types of systems provide information based on the value-added processing of data.

This philosophy has motivated the development of software modules known as *mediators* [Wiederhold 1992]. Mediators apply database and knowledge-based techniques to provide interfaces between applications and sets of autonomous databases. Mediators apply database techniques to locate, retrieve, and integrate data from heterogeneous database systems, and mediators apply knowledge-based techniques to perform value-added abstractions and simulations based on the retrieved data. One of the primary purposes of mediators is to insulate applications from the complex and changing features of the databases that they access. By creating a single uniform interface to multiple heterogeneous information sources, a mediator greatly simplifies the development of applications that need to access those data sources.

Given the similarities between database and knowledge-based systems at a fundamental level, the strategy of applying both database and knowledge-based techniques to mediate between applications and databases is appropriate to the current objective of interoperability in medical informatics. For example, the principles that AI researchers are discovering for the specification of robust sharable ontologies and the reasoning tools that AI researchers are developing for the validation of domain ontologies certainly are applicable to the development of a standard conceptual schema of clinical data, such as the MEDIX reference model. Also, the techniques developed by database researchers for the resolution of semantic discrepancies among database schemata and for the optimization of queries that are distributed among heterogeneous systems certainly are applicable to the development and implementation of portable languages for the representation of decision-support logic, such as the Arden Syntax. Because the task of providing medical decision-support applications with access to patient data that is stored in disparate clinical information systems is a necessarily complex undertaking, sound research results from all applicable fields should be brought to bear upon it. The result of this approach will be a *clinical information mediator* that applies database and AI techniques to insulate CDS applications from the heterogeneities and complexities of clinical database systems.

2.5 Components of the Solution

The technology that I propose to provide decision-support applications with access to patient data in heterogeneous clinical information systems consists of a standard abstract query model of clinical information and a means to translate standard abstract queries to implementation-specific operations. Previous research exists relating to each component of this strategy: An abstract data model and corresponding query language, a standard schema of clinical information represented in the abstract data model, and a means of translating queries against the standard schema to queries that can execute against diverse clinical databases. My work is based upon certain results of this past research.

2.5.1 An abstract data model and query language

As shown by HDBS research, database interoperability requires the use of a common data model that is sufficiently simple and abstract to translate conveniently to various implemented data models and a corresponding query language that can be used to formulate queries at an equally abstract level. *Semantic data models* [Hull and King 1987, Peckham and Maryanski 1988] provide appropriately abstract conceptualizations of domain data that can serve as common modeling environments for disparately implemented databases. Although semantic data models (SDMs) have been traditionally used for database design because they provide more powerful abstractions for the specification of database schemas than are supported by the relational, hierarchical, and network models, the abstract and expressive modeling constructs of SDMs also are well suited for the specification of conceptual schemas (see Section 2.1) that model domain data in an implementation-independent way. Most SDMs provide the following abstractions:

1. **Classification**: The designation of individual objects as members of an abstract object class. For example, the classification of several persons as members of the class "Patient."

2. **Aggregation**: The designation of multiple object classes as components of a more abstract object class. For example, the aggregation of the classes "Sodium test result" and "Potassium test result" to form the class "Electrolyte test results."

3. **Generalization**: The designation of one object class as the supertype of another object class. For example, the generalization that the object class "person" is a supertype of the object classes "Patient" and "Physician."

4. **Association**: The designation of individual objects as participants in an abstract object that denotes a relationship. For example, the association of two persons such that one is the physician of the other.

The best-known examples of SDMs are the entity–relationshp (ER) model [Chen 1976], including numerous variations of it [Elmasri, et al. 1985, Engels, et al. 1992], and the functional data model [Shipman 1981, Wilkinson, et al. 1989]. Other SDMs include extended relational models [Codd 1979, Elmasri and Wiederhold 1979], and data models based on semantic networks [Mylopoulos, et al. 1980, Sowa 1984]. I have based the conceptual model that I use on both the ER model and the functional model, combining the desirable features of each. I describe the features of this hybrid data model in Chapter 3.

The SDMs suited for supporting database interoperability have corresponding high-level query languages, because a query language is necessary to provide an operational interface to a variety of database systems. The functional data model and the entity–relationshp data model used in the HDBS research projects described in Section 2.4.1 both provide corresponding query languages and were likely selected as common data models for that reason. A third candidate is the conceptual graph (CG) data model, which also has its own query language. Although CGs have not yet been used as the conceptual data model in experimental heterogeneous database systems, they were recently approved by an ANSI standards committee as the normative modeling language for the Information Resource Dictionary System (IRDS) [ANSI-X3H4 1993]. The IRDS is a modeling environment that describes and manages an organization's information system resources and supports the integration of an enterprise's information environment and information models. As such, the IRDS separates the conceptual view of information (its *meaning*) from the way in which information is represented, stored, and processed, so that it can be used and applied in many different ways. Because the use of CGs in this capacity is quite new and yet to be proven, I have chosen to base my abstract query model on the entity–relationshp model and the functional model.

2.5.2 A Domain schema for clinical medicine

The specification of a standard query model for clinical information requires that a standard *schema* of medical data be specified using the chosen conceptual data modeling environment. Although numerous groups are currently pursuing the specification of a standard conceptual schema for clinical data, no such schema has yet emerged in the public domain. The MEDIX reference model described in Section 2.3.1 has yet to be published as a national standard. Although the Computerized Patient Record Institute (CPRI) has called for the specification of a composite clinical data dictionary to support clinical database interoperability, the CPRI is currently developing no such reference model of its own. An independent group of medical informatics researchers recently convened to begin work on a “standard canonical scheme of medical concept representations” [Evans, et al. 1994], but the work of the “Canon” group is still very nascent and is primarily oriented towards standardizing medical terminology rather than medical database schemas. Numerous proprietary efforts to define global conceptual schemas for clinical data also are under way, but these are not likely to be useful in supporting widespread interoperability among clinical databases unless they are published and accepted as public-domain standards. In short, there currently exists no comprehensive and publicly available schema of clinical information encoded in *any* modeling environment that could serve as the global schema in a standard query model for clinical medicine. The definition of a global schema for sharing decision-support applications, therefore, will have to proceed de novo; however, the process will be unhindered by the preexistence of a global schema that has been widely adopted but may be inadequate for the purposes of sharing applications (because it violates Gruber's criteria for shared ontologies, for example).

2.5.3 A query-translation methodology

An abstract data model and a standard global schema of clinical data are only part of the solution for providing applications with a uniform interface to clinical databases. The other part involves "operationalizing" the global schema such that queries issued against it can retrieve patient data that is physically stored in heterogeneous database systems. Unless the implementation of clinical databases is standardized, which is unlikely and may be undesirable, some transformation must take place between the abstract query model against which the queries of shared applications are formulated and the local query models in which clinical data actually are stored. The transformation may take one of two forms: data translation or query translation. Although I have chosen to pursue the strategy of query translation, I discuss both strategies for purposes of comparison.

2.5.3.1 User-mediated translation

In order to appreciate at a higher level of abstraction the data-translation and query-translation strategies that I will describe, it is instructive to consider how shared applications that require patient data achieve widespread portability today. The prevailing model entails the *manual* translation of patient data and of application queries by end users. In this paradigm, used by decision-support systems like MYCIN and QMR, the "global schema" of clinical information is the informal mental model of medicine shared by all professionally trained users. Applications issue queries against this "schema" by formulating questions in the corresponding "query language," English (presumed to be supported at every site). The "query engine" consists of a human end user, who services queries in one of two ways. If the requested data have already been translated by the user from its original representation in the patient chart to its “global conceptual” representation in the user's mind (that is, if the user “knows” the information), the query engine can respond to the query directly and immediately. This mode of query response involves **data translation** from the stored data format (the patient chart) to the shared data format (the user's mental model) prior to consideration of any queries.

Alternatively, if the data is not already known to the user, the English query must be translated into a search strategy against the patient chart to locate the data and to respond to the query. This mode of query response involves **query translation** from the shared formulation (an English question) to a formulation relevant to the stored format (a search procedure in the chart). These two methods of manually servicing portable queries are analogous to the data-translation and query-translation strategies that must be formalized in a computational model to automatically service queries with minimal human intervention.

2.5.3.2 Translation of data

Data translation involves the transformation of data from the various native formats in which they may be collected and stored to a common shared format in which they can be queried by portable applications. Data translation may take place manually or automatically. Manual data translation requires medical records personnel to manually abstract patient records into a format consistent with a standard clinical data set and with a standard coding scheme. The data that are manually “normalized” in this way are subsequently entered into a standard centralized database and all applications that require clinical data query this database. This is the model that many large clinical data banks, such as ARAMIS, employ for aggregating data from multiple sites. Although this method could be applied to normalize data into a standard format that portable applications can query, the costs and time requirements of manually translating each collected datum from a stored format to the shared format are too great to accommodate many decision-support applications.

An alternative method for translating data is to develop computer programs (often called “geteways”) that algorithmically translate data from the stored format in which they are captured to a shared format in which they can be accessed by portable applications. Gateways may convert data directly to the shared format [Marrs, et al. 1994], or they may translate the data first to an “interchange” format (such as HL7 [Rishel 1989]) that is standard across database implementations and that subsequently can be translated to the shared format. Although automating data translation reduces the costs and delay of translating data, this strategy still suffers from two problems. First, data translation entails the duplicate storage of data in both the original and the shared format, which increases both the cost of operating an information system and the chances of compromising data integrity. Second, the correspondence between the stored format and the shared format is represented only in the encoded algorithms of the translating programs, and this “procedural” representation is difficult to inspect, validate, and maintain. A procedural representation of the correspondence between disparate data formats increases the chances of incorrectly translating complex medical data and increases the costs of updating the translation algorithms.

2.5.6.3 Translation of queries

The alternative strategy for providing a common query interface to disparately stored data is to translate the queries rather than the data. This model entails the translation of queries that are issued against the shared format to equivalent queries that can be executed against the stored format. Again, query translation can be implemented either manually or automatically.

Manual query translation requires that database programmers reformulate each query represented in a portable application into an equivalent query that is consistent with the database system at their provider site. The programmers perform the translation based on their interpretation of the meaning of the original query and on their understanding of the meaning of the locally stored data. Manual query translation is the method proposed for sharing database queries that appear in medical logic modules encoded in the Arden Syntax [Hripcsak, et al. 1990]. The problems inherent in this method of query translation include the effort required to perform the translation and the potential for semantic errors introduced by the translation. Because there is no formal standard model of clinical data and because the structure of clinical databases is complex and highly variable, considerable effort is required to understand the meaning of a shared query and to implement the same meaning with respect to a local database environment [Hripcsak 1994, Pryor and Hripcsak 1994].

Automated translation of queries provides, in theory, the opportunity to remedy the deficiencies of manual query translation. Automated query translation requires that programmers represent only a *mapping* between the shared data format and their local data format. A query-translation and query-execution *engine* uses the mapping to automatically reformulate each query into an equivalent local query and to execute the local query. This strategy requires less manual effort and allows the systematic representation and application of mappings between query models. I base my approach to providing a uniform query interface to disparate clinical databases on this strategy, and I discuss two variations of the strategy that appear in current research systems and that have influenced my work.

1. *Automated query translation based on canonical mappings*. In this method, an entity–relationshp (ER) schema initially is designed and then algorithmically transformed into a corresponding relational schema, based on a unique and canonical set of schema transformations. Queries issued against the ER schema are automatically translated to equivalent relational queries by a compiler that performs query transformations based on the canonical mappings. Essentially, the mapping between any ER schema and its corresponding relational schema is *hard-coded* into the translating compiler. This method is used by two independent research groups to translate queries from an ER-based query language to an equivalent relational form [Hohenstein 1990, Markowitz and Shoshani 1993]. The advantage of automatically translating queries based on *canonical* mappings is that the mappings and the corresponding query-translation algorithm may be readily formalized and validated because they are invariable, regardless of the specification of the ER schema. One of the research groups [Hohenstein and Engels 1990] formally models the translation algorithm as an attribute grammar [Knuth 1968], which constitutes a declarative representation of the translation algorithm that can be formalized and validated.

The disadvantage of query translation based on canonical mappings between an ER schema and a relational schema is that it precludes the use of a single ER schema as an interface to different legacy databases; the structure of the legacy databases will not conform, in most cases, to the canonical structure required by the query-translation algorithm. The missing element is a flexible means of representing the mapping between a conceptual schema and a relational schema, such that the mapping may be both validated and used to drive query translations. Although I have borrowed the technique of representing mappings in an attribute grammar, I have enhanced it to be more flexible and to support various relational implementations of a single conceptual schema.

2. *Automated query translation based on procedural mappings*. In this method, the mappings between the conceptual database schema and various underlying database implementations are represented as procedural functions that physically import objects stored in underlying databases into corresponding objects stored in the conceptual environment, where they may be manipulated further as part of query processing. This approach characterizes the database-integration strategy of the Physician Workstation project at Hewlett-Packard Laboratories [Annevelink, et al. 1992]. This strategy is effectively a hybrid of the data-translation and the query-translation strategies: Procedural functions translate and import data from a stored format into the shared format, but the functions are only invoked when queries require data that the functions provide. Also, the mapping functions may specify that the underlying database not only retrieve requested objects, but also apply certain of the data operations included in conceptual-level queries (such as filtering results). Currently, the scope of operations that can be specified in this way is limited.

The advantage of query translation based on procedural mappings as opposed to canonical mappings is that procedurally specified mappings can accommodate a wide range of legacy database implementations and are not confined to “canonically” implemented versions of the conceptual schema. The disadvantage of this approach is that the query engine cannot perform certain optimizations because the mappings between query models are specified as *procedural* functions that cannot be decomposed, reordered, and recombined to achieve maximum efficiency [Annevelink, et al. 1992]. To maximize the potential for optimization, a *declarative* representation of query-model mappings is necessary. Because a query optimizer can inspect and manipulate declarative mappings based on an encoded understanding of the operations within the mappings, the optimizer can apply information about the semantics of the data, the state of the database, and the capabilities of the DBMS to determine an optimal sequence and allocation of the operations.

Although experimental methods exist for automatically translating queries from a semantic data model to a relational data model, none of these methods can map a single schema of clinical information to various legacy databases and use the mappings to generate optimized queries. Because portable decision-support applications require these features, I have borrowed from and enhanced the existing methods to develop a model of query translation that can deliver the needed functionality. I describe this methodology in the following two chapters.

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.

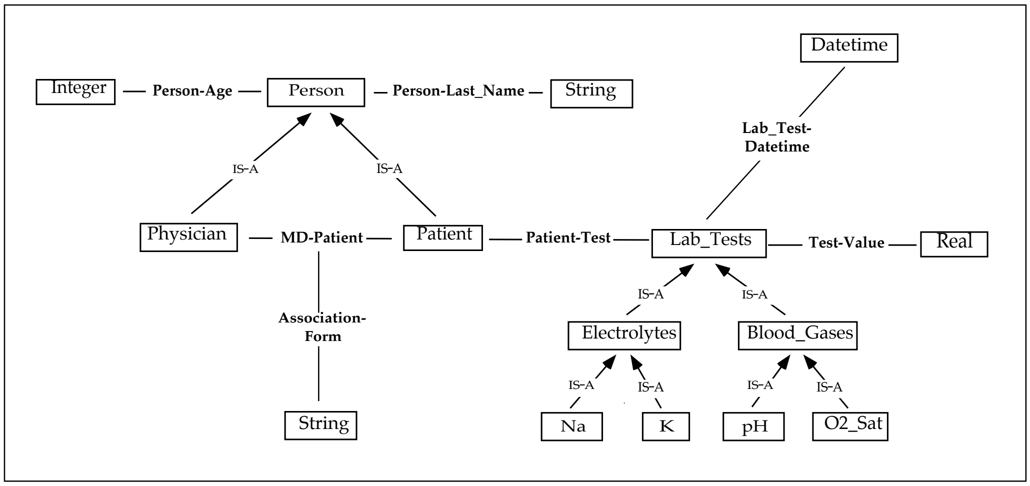
2. *Abstractness*: The constructs of the data model should be sufficiently general to allow a reference schema encoded in the data model to

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

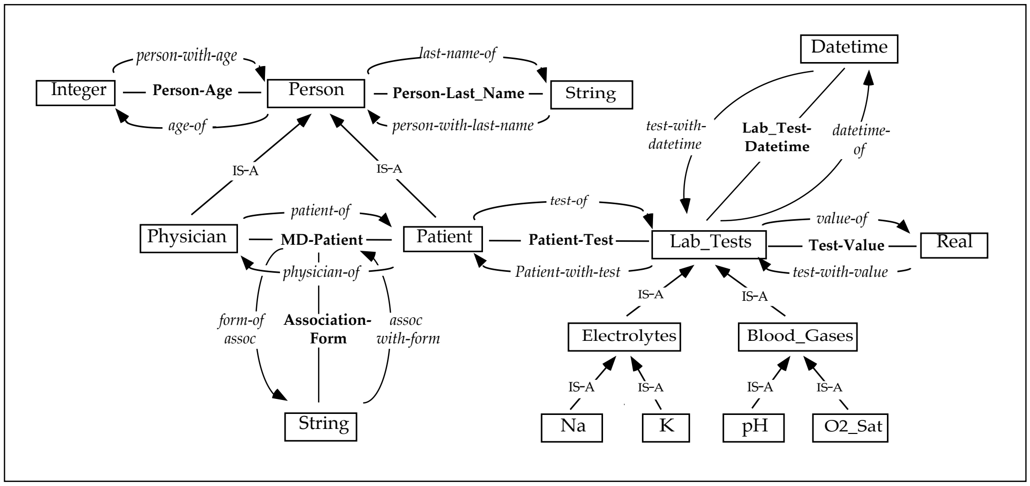
Lab_Tests and that the entity type Electrolytes inherits all relationship types in which Lab_Tests participates (specifically, Patient-Lab and Lab-Value).

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

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 = { $e_1, \ldots, e_n$}, where each e_i , $1 \leq i \leq n$, represents a unique object in the domain of discourse. The objects $e_1, \ldots, e_n$ 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

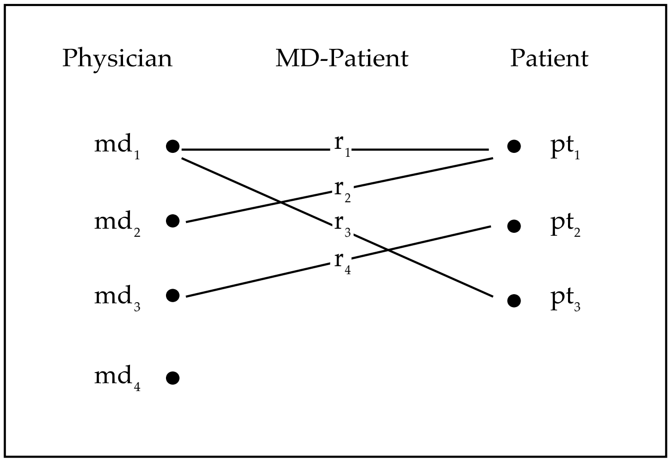
Given an FER schema and a database state, each relationship type "R" in the schema defines a set $R = \{ r_1, \ldots, r_m\}$, where each r_i, $1 \leq i \leq m$, represents a distinct association between two objects that is represented in the database. The associations $r_1, \ldots, r_m$ are objects themselves 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

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

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 $e \in \text{participant\_type}_1(R)$ and $e' \in \text{participant\_type}_2(R)$

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

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

$$\nu[\text{Term}, b]$$

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*:

$$\nu[\text{pt}, <\text{pt} = p1>] = p1$$

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 ν[age-of(pt), b], because the AVG function will always be applied to a set containing one element.

(10*m*) $\nu[\text{Term}_1 \text{ ArithFunction Term}_2\text{, b}] = \nu[\text{Term}_1\text{, b}] \text{ ArithFunction } \nu[\text{Term}_2\text{, b}]$

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 $\nu[\text{Term}_1\text{, b}]$ and $\nu[\text{Term}_2\text{, b}]$ 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,

$$\nu[\text{age-of(pt) + age-of(md), b}] = \nu[\text{age-of(pt), b}] + \nu[\text{age-of(md), b}].$$

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

{ "Smith" } × { 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 ∪ 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,

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

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

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

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

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,

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

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

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(ν[md, b], ν[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*) $\tau[\text{Predicate}_1$ AND Predicate_2, b] = TRUE iff
$\tau[\text{Predicate}_1$, b] = TRUE AND $\tau[\text{Predicate}_2$, b] = TRUE

(18*m*) $\tau[\text{Predicate}_1$ OR Predicate_2,b] = TRUE iff
$\tau[\text{Predicate}_1$, b] = TRUE OR $\tau[\text{Predicate}_2$,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,

ν[[last-name-of(md) | (md : Physician) AND
MD-Patient(md,pt)], <pt = *p1*>] =

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

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.

Chapter 4

An Automated Query-Translation Methodology

An expressive and abstract query model for clinical medicine is only a prerequisite to providing applications with access to heterogeneous clinical databases. The second component of the technology is a principled methodology for translating automatically queries specified using the abstract query model to semantically equivalent queries that can be executed by heterogeneous clinical databases. In this chapter, I describe the query-translation methodology of the TransFER model. In Section 4.1, I discuss the desirable features of query-translation methodologies in general and I introduce the framework of the TransFER query-translation methodology, which is based on the concept of attribute grammars and the relational algebra. In Section 4.2, I describe how database programmers can use a mapping language based on the relational algebra to encode site-specific mappings between an FER schema and a relational database. Section 4.3 provide a description and a detailed example of how the translation methodology applies the encoded mappings within the attribute-grammar framework to translate ReFER queries to site-specific SQL queries. In Section 4.4, I describe the theoretical basis of the optimization techniques used by the TransFER compiler to simplify translated queries.

4.1 Framework of the Methodology

An ideal query-translation methodology for the FER query model would satisfy the following criteria:

1. *Automation*: The methodology translates queries with no human intervention. Although human intervention is required to specify the

mappings between a particular clinical database and the abstract FER query model, subsequent use of the mappings to translate individual queries is completely automated. The human labor required to specify the mappings is amortized over the queries that are later translated automatically.

2. *Latitude*: The methodology is general and powerful enough to translate FER queries to the query models of a great variety of clinical databases, provided that the databases are semantically consistent with the FER query model. *Semantic consistency* means, informally, that the same semantic information is represented in the query model of the implemented database as is represented in the abstract query model, although the technical means by which the information is represented may differ. Differences between semantically consistent query models constitute *representational heterogeneity* [Barsalou and Gangopadhyay 1992]. Several types of representational heterogeneity exist [Agarwal 1991] and the ideal query-translation methodology is able to resolve all of them.

3. *Efficiency*: The methodology automatically generates queries that are as computationally efficient as queries translated manually by database programmers. Different formulations of semantically equivalent queries can dramatically influence the queries' computational efficiency, even given the query-optimization capabilities of many commercial relational query engines. Database programmers, who are familiar with the semantics of particular databases and with the costs of various query operations, are adept at writing queries that retrieve requested data with the minimum computation. The ideal automated translation methodology applies similar optimization principles to generate queries of comparable efficiency.

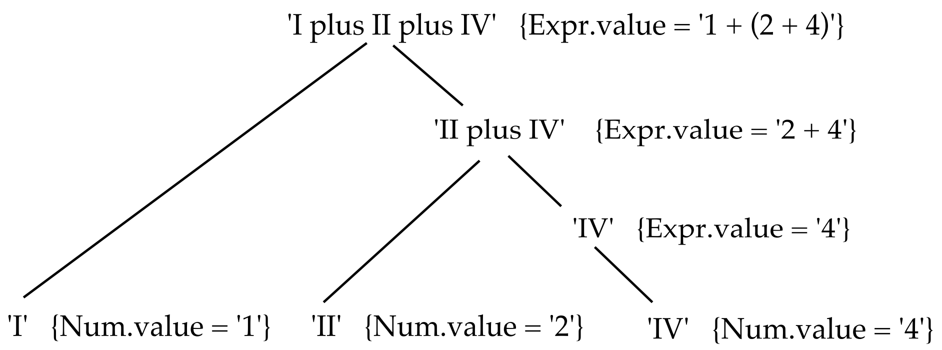
The query-translation methodology that I propose, called TransFER, attempts to fulfill each of these design criteria. The methodology provides automated query translation by implementing a syntax-directed translation engine that takes as input only a ReFER query and a set of database-specific mappings. The methodology provides an expressive database mapping language to accommodate a wide variety of clinical database implementations. The methodology optimizes the queries that it automatically generates by applying well-founded transformation techniques based on the theory of relational algebra.

The computational framework of the translation methodology consists of an *attribute grammar* for the ReFER query language (the language was described in Section 3.3). Attribute grammars, first introduced by Knuth [Knuth 1968], are context-free grammars (CFGs) in which each terminal and nonterminal symbol is associated with a set of attributes. Attribute grammars are useful for formally representing the transformation of one language to another because the set of attributes associated with the syntactic symbols of one representation system can denote the semantics of the symbols in a different representation system. When an expression is parsed by an attribute grammar, the values of the attributes associated with each symbol are assigned, usually based on the values of attributes associated with adjacent symbols in the parse tree. The rules for assigning values to attributes are called *semantic functions*. Together, the productions of a CFG and the semantic functions for deriving the attributes of each symbol in the CFG constitute an attribute grammar.

4.1.1 A sample attribute grammar

As a general example, Figure 4.1 shows a simple attribute grammar that recognizes expressions denoting the addition of Roman numerals using the operator "plus," such as "II plus IV." The purpose of the attribute grammar is to translate these expressions into conventional mathematical notation, such as "2 + 4," so that they can be computed by a standard calculator The boldfaced text in the upper panel of Figure 4.1 represents the CFG for the Roman numeral addition language. The plain text in the upper panel of Figure 4.1 represents the semantic functions for assigning a value to each attribute. Note that each symbol on the left-hand side of a grammar rule (that is, "Expr" and "Num") has a single attribute named "value," which denotes the value of the symbol in conventional mathematical notation. For terminal symbols (that is, leaves in the parse tree), such as "Num," the semantic function assigns a constant value to the attribute; for nonterminal symbols, such as "Expr," the semantic function derives a value for the attribute based on the values of attributes previously assigned to symbols on the right-hand side of the grammar rule. The lower panel of Figure 4.1 shows the parse tree for the expression "I plus II plus IV," with all attributes assigned. Note that the attribute value at the root of the parse tree ("1 + 2 + 4") has precisely the same semantics as the input string ("I plus II plus IV"), although it is represented in a completely different syntactic notation. The effect of the attribute grammar, therefore, is to translate any expression formulated in the Roman numeral addition language into a semantically equivalent expression in conventional mathematical notation. If one were to interface the attribute grammar in Figure 4.1 to a standard calculator, one could enter the expression "I plus II plus IV" and receive the result "7."

Expr$_1$: Num 'plus' Expr$_2$	Expr$_1$.value = Num.value '+' Expr$_2$.value
Expr : Num	Expr .value = Num.value
Num : 'I'	Num.value = '1'
Num : 'II'	Num.value = '2'
Num : 'III'	Num.value = '3'
Num : 'IV'	Num.value = '4'
etc...	



semantically equivalent expressions formulated in the relational algebra. By translating ReFER queries to relational algebra, one can retrieve the requested data from a relational database, just as by translating Roman numeral expressions to mathematical notation one can compute the result with a standard calculator. By customizing the attribute grammar to particular relational database implementations, one can generate relational algebra translations of ReFER queries that are site specific. The relational algebra that I use is a version of the standard relational algebra, which I have enhanced to have the expressive power of SQL[1]. I describe the algebra in Section 4.1.3.

Figure 4.2 shows an excerpt of attribute grammar for the ReFER query language and a parse tree generated by the attribute grammar for the simple ReFER query

"[name-of(pt) | (pt : Patient)]".

The translation of the query is performed by the semantic functions of the ReFER attribute grammar, just as the translation in Figure 4.1 is performed by the semantic functions of the attribute grammar for the Roman numeral addition language. The semantic functions corresponding to each symbol in the ReFER attribute grammar (shown in plain text in the upper panel of Figure 4.2) assign a relational algebra expression to the symbol at the time it is parsed, much the way that the semantic functions of the attribute grammar in Figure 4.1 assign a conventional arithmetic expression to each symbol at the time it is parsed. For terminal symbols, such as Entity and Relationship, the relational algebra expression corresponding to the symbol on the left-hand side of a grammar rule is constant; these expressions, in fact, represent the site-specific mappings for the entity types and relationship types of the FER schema. For example, the attribute grammar in Figure 4.2 assigns the relational algebra expression *P* to the entity type Patient and the relational algebra expression *N* to the relationship type Person-Name. Note that *P* and *N* are just place holders in the example of Figure 4.2; in an actual ReFER attribute grammar, they might be complex relational algebra expressions, such as

---

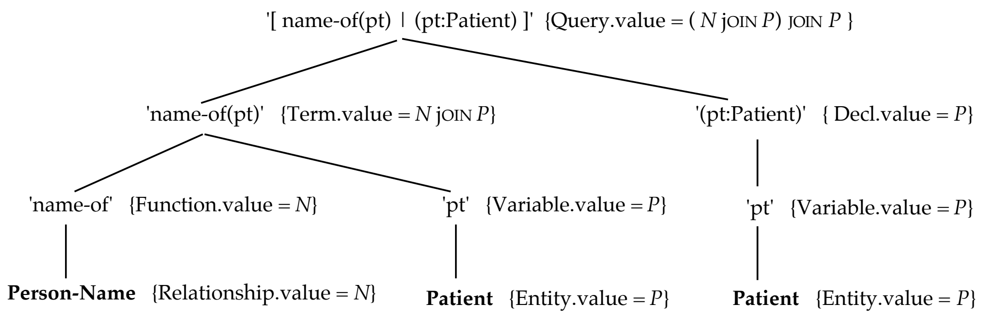
[1] I refer here to the ANSI standard SQL published in1989[Date 1989a], commonly referred to as "SQL89."

```
Project( [ LastName, FirstName ],
         Select( [ Ward = “Medicine” OR
                   Ward = “Surgery”],
                  Union (  InPatients, OutPatients ) ) ).
```

The specification of mapping expressions like *P* and *N* is discussed in Section 4.2. Note that variables in ReFER queries are assigned the values of the entity types with which they are associated in the declaration statement (that is, “pt” is assigned the value for “Patient”) and functions in ReFER queries are assigned the values of the relationship types to which they correspond in the FER schema (that is, “name-of” is assigned the value for “Person-Name.” The rationale for these assignments is discussed in Section 4.3.3.2. For nonterminal symbols, such as “Term” and “Query,” the corresponding relational algebra expressions must be constructed from the relational algebra expressions previously assigned to the symbols on the right-hand side of the grammar rules. The semantic functions of the attribute grammar construct relational algebra expressions by specifying relational algebra operations, such as JOIN and UNION. Note that the semantic functions for the nonterminal symbols “Query” and “Term” in Figure 4.2 are oversimplified because they do not specify the fields on which relational algebra expressions are to be joined. This omission is to simplify the diagram only. The semantic functions of the actual attribute grammar do specify the join fields, as described in Section 4.3.3.

Because the relational algebra is an *applicative* language [MacLennan 1987], the application of relational operators to relational expressions produces new relational expressions. By recursively combining relational algebra expressions as specified by the semantic functions, the attribute grammar constructs a single relational algebra expression in a bottom-up fashion. This relational algebra expression is semantically equivalent to the input ReFER query, just as the arithmetic expression constructed at the root of the parse tree in Figure 4.1 is equivalent to the input Roman numeral expression. The relational algebra expression subsequently can be translated to SQL via a straightforward transformation, as I discuss in Section 4.3.4. When the attribute grammar is interfaced to an SQL query engine, ReFER queries can retrieve data directly from relational databases.

Query : '[' Factor '|' Decl ']'	Query .value = Term.value JOIN Decl.value
Term$_1$: Function '(' Term$_2$ ')'	Term$_1$ .value = Function.value JOIN Term$_2$.value
Term : Variable	Term.value = Variable.value
Decl : '(' Variable ':' Entity ')'	Decl.value = Variable.value
Variable : <string>	Variable.value = Entity.value
Function : <string>	Function.value = Relationship.value
Entity : Patient	Entity.value = *P*
Relationship : Person-Name	Relationship.value = *N*



ReFER attribute grammar, all table names and field names referenced in the relational algebra expression that is produced by the translation will exist in the local relational database, assuming that the mappings have been specified correctly. Examples of mappings for entity types and relationship types appear in Section 4.2.

2. TransFER attribute grammars do not immediately evaluate the relational algebra expressions that they generate. Deferral of evaluation allows the resulting expressions to be transformed into more efficient forms (while retaining the same semantics) before the relational query is actually executed. For example, an optimizer may determine that joining the relational algebra expression *N* to the expression *P* twice in the translation result of Figure 4.2 is superfluous, and that the simplified expression (*N* join *P*) is equivalent to the original expression ((*N* join *P*) join *P*). Relational algebra is well suited for representing the translation of ReFER queries because numerous formal techniques already exist for optimizing relational algebra expressions. The use of these techniques is discussed in Section 4.4.

To summarize, the TransFER attribute grammar translates a ReFER query into a semantically equivalent relational query by (1) decomposing the ReFER query into its syntactic components, (2) translating each component into an appropriate relational algebra expression, and (3) combining the expressions into a single relational algebra query. Before I discuss how the semantic functions of the attribute grammar generate and combine relational algebra expressions to effect query translations, I introduce the relational algebra language in which those expressions are formulated. Recall that this language is used to represent the site-specific mappings for FER entity and relationship types (such as “Patient” and “Person-Name”) as well as the site-independent translations of ReFER query operations (such as “name-of(pt)”).

4.1.3 The extended relational algebra (ERA)

Because the goal of the TransFER attribute grammar is to translate ReFER queries into representations that are meaningful with respect to relational databases, it is natural to use a relational data-manipulation language (DML) to specify the relational semantics of ReFER query constructs. The most prominent DMLs for relational databases are the relational calculus and the relational algebra [Codd 1972]. Although variants of the relational calculus (usually SQL) constitute the query interfaces of most modern relational databases, I have chosen the relational algebra for representing the relational semantics of ReFER constructs. The relational algebra is preferable because it is an *applicative* language, which supports both the incremental construction of relational queries from modular subcomponents and the systematic optimization of the constructed queries. The incremental assembly of relational subcomponents is compatible with the framework for query translation that I described earlier in this section because it allows the relational semantics of FER entity types and relationship types to be specified in a context-free fashion and to be combined later as needed. The systematic optimization of relational algebra expressions is consistent with the efficiency criterion for query translation methodologies because it allows optimization procedures to be encoded and automatically performed. The relational algebra and the relational calculus are equally expressive [Codd 1972, Klug 1982], and conversion from one to the other is straightforward.

The relational algebra is a query language based on the notion of applying operators to relations. The original relational algebra and extensions to it have been described extensively [Codd 1970, Klug 1982, Maier 1983]. I have adopted this work to define an *extended relational algebra* (ERA) that I use to represent the relational semantics of FER schemas and ReFER queries. As opposed to the original relational algebra, ERA is equal in expressiveness to SQL[2] and can, therefore, utilize the full capabilities of modern SQL query processors.

Table 4.1 lists the operators of the ERA. The syntax for relational operators that I use in this document differs from the conventional syntax but the semantics of the operators are similar. Let <relation> be any well-defined relational algebra expression with respect to the relational data model, <attrs> be any set of field names corresponding to the columns in a relation, <consts> be a set of constant values, and <conds> be a set of conjoined conditions of the form "<attrs> = <attrs>" or "<attrs> = <consts>." Given these definitions, the relational algebra operations of the ERA are listed in Table 4.1.

---

[2] with the sole exception that the ERA does not include the SQL sorting operation "ORDER BY."

ERA Operator	Syntax
PROJECT	**Proj ([**<attrs>**],** <relation>**)**
MULTIPROJECT	**MProj ([**<attrs>**],** <relation>**)**
SELECT	**Sel ([**<conds>**],** <relation>**)**
JOIN	**Join ([**<conds>**],** <relation>, <relation> **)**
UNION	**Union (** <relation>, <relation>**)**
DIFFERENCE	**Diff (** <relation>, <relation>**)**
AGGREGATE	**Agg([** <arg-attrs>, <group-attrs>, <fxn>, <new-attrs>**],** <relation>**)**
RENAME	**Rename([**<attr-renames>**]**, <relation> **)**
EXTEND	**Extend([** <attr-defns>**]**, <relation> **)**
EXISTS	**Exists(** < relation> **)**

Table 4.1: The operators of the extended relational algebra. The definitions of <relation>, <attrs>, <conds>, <fxn>, <attr-renames>, and <attr-defns> are discussed in the text. Note that the syntax differs from the conventional syntax for relational operators (that is, π, σ, ∪, etc.), but is more convenient for expository purposes.

4.1.3.1 Project, Select, Join, Union, Difference

The operators PROJECT, SELECT, JOIN, UNION, and DIFFERENCE in Table 4.1 have the same semantics as specified by Codd for the original relational algebra [Codd 1972]. The correspondence between the syntax of the operators used in Table 4.1 and the conventional syntax is as follows:

ERA Syntax	Conventional Syntax
Proj ([<attrs>**],** <relation>**)**	$\pi_{<\text{attrs}>}$ <relation>
Sel ([<conds>**],** <relation>**)**	$\sigma_{<\text{conds}>}$ <relation>
Join ([<conds>**],** <relation>, <relation>**)**	<relation> $\bowtie_{<\text{conds}>}$ <relation>
Union (<relation>**,** <relation>**)**	<relation> $\cup$ <relation>
Diff (<relation>**,** <relation>**)**	<relation> $-$ <relation>

4.1.3.2 Multiprojection

The MULTIPROJECT operator is identical to the PROJECT operator except that duplicate rows are retained (the classic PROJECT operator eliminates any duplicate rows that result from "projecting out" certain fields in a table).

4.1.3.3 Aggregation

The AGGREGATE operator is a generalization of the relational aggregation operator introduced by Klug [Klug 1982]. The AGGREGATE operator in Table 4.1 partitions a relation into sets of tuples that have the same values of the attributes specified in <group-attrs>; within each partition, it applies the aggregation function <fxn> to the set of values specified in <arg-attrs>, and returns a new relation consisting of the Cartesian product of the result of the aggregation function (which receives the new attribute name <new-attrs>) and the value of <group-attrs> for that partition. For example, given the relation "Doctors" shown in Figure 4.3 and the aggregation function AVERAGE, the relational algebra expression

R = **Agg([** {Age}, {Specialty}, AVERAGE, {Avg-Age} **],** Doctors**)**

would generate the relation R shown in Figure 4.4 by taking the average of the "Age" values for each partition with the same value of "Specialty" and naming the resulting value "Avg-age."

Doctors =

Name	Specialty	Age
"Smith"	"Ortho"	50
"Jones"	"Ortho"	60
"Thomas"	"Cardio"	40
"Cross"	"Cardio"	50

Figure 4.3: A relational table named "Doctors."

R =

Specialty	Avg-Age
"Ortho"	55
"Cardio"	45

Figure 4.4: A relational table generated by applying the AGGREGATE operator to the relational table in Figure 4.3.

4.1.3.4 Rename and Extend

The RENAME operator simply renames the specified attributes that appear in the relation to which it is applied. The renaming specifications appear in a list of terms of the form "<old-attr-name> => <new-attr-name>." For example, given the relation shown in Figure 4.3, the relational algebra expression

S = **Rename([** Specialty **=>** Dept, Name **=>** Last-Name **],** Doctors**)**

would generate the relation S shown in Figure 4.5.

S =

Last-Name	Dept	Age
"Smith"	"Ortho"	50
"Jones"	"Ortho"	60
"Thomas"	"Cardio"	40
"Cross"	"Cardio"	50

Figure 4.5: A relational table generated by applying the RENAME operator to the relational table in Figure 4.3.

The EXTEND operator adds a column to the relation to which it is applied. The value of the column may be a constant or the value of any other column that already exists in the relation. The extension specifications appear in a list of terms of the form "<new-attr-name> **==** <new-attr-value>." For example, given the relation shown in Figure 4.3, the relational algebra expression

T = **Extend([** Degree **==** "MD"**,** Experience **==** Age**],** Doctors**)**

would generate the relation T shown in Figure 4.6.

T =

Name	Specialty	Age	Degree	Experience
"Smith"	"Ortho"	50	"MD"	50
"Jones"	"Ortho"	60	"MD"	60
"Thomas"	"Cardio"	40	"MD"	40
"Cross"	"Cardio"	50	"MD"	50

Figure 4.6: A relational table generated by applying the EXTEND operator to the relational table in Figure 4.3.

4.1.3.5 Exists

The EXISTS operator actually does not perform any transformation on the relation to which it is applied, but specifies that any JOIN operations subsequently applied to the Exists() expression are *semi-joins* [Ullman 1988 p. 60]. For example, given the relations Doctors and Fathers shown in Figures 4.3 and 4.7, respectively, the relational algebra expression

U = **Join([** Name = LName**],** Doctors**, Exists(**Fathers**))**

would generate the relation U shown in Figure 4.8.

Fathers =

LName	Child-Name
"Smith"	"Maggie"
"Smith"	"Chuck"
"Cross"	"Carlos"
"Cross"	"Martha"

Figure 4.7: A relational table called "Fathers."

U =

Name	Specialty	Age
"Smith"	"Ortho"	50
"Cross"	"Cardio"	50

Figure 4.8: A relational table generated by applying the JOIN operator to the relational table in Figure 4.3 and to an EXISTS operation over the relational table in Figure 4.7.

4.1.3.6 Virtual relations

An unusual but useful feature of the extended relational algebra is the notion of *virtual relations*. Virtual relations are named relations that have attributes and may appear in relational algebra expressions, but which do not actually exist as stored relations in a database. A virtual relation represents a pair of *functions* that simulate the existence of a stored relation at the time that the virtual relation is joined to an actual relation. For example, consider the virtual relation "Age-BirthYear," which has the following schema:

```
Age-BirthYear = field          functional definition
                 curr-age:     compute_age(birth-yr)
                 birth-yr:     compute_birth_year(curr-age).
```

The virtual relation represents mappings between birth years and current chronological ages. For example, on November 1, 1995, the age 50 maps to the birth year 1945 and the birth year 1960 maps to the age 35 (note that these mappings are not strictly precise because the exact birth date is not considered; the functions make the assumption that every person's birth date is January 1). Because Age-BirthYear is a virtual relation, the mappings between ages and birth years are not actually stored. Rather, the functions "`compute_age()`" and "`compute_birth_year()`" that correspond to the fields of the virtual relation compute the appropriate mappings when the virtual relation is joined to an actual relation. For example, the mappings are invoked in the following relational algebra expression, which derives the birth years of the physicians represented in Figure 4.3.

Proj([birth-yr **],**
    Join([Age = curr-age **],**
        Doctors,
        Age-BirthYear **)).**

The mechanism by which the mapping is simulated is that the value of the field to which "curr-age" is joined (that is, "Age") is substituted as the argument to the function "`compute_birth_year()`" and the result is assigned to the field "birth-yr." In this case, the result of joining the "Doctors" relation to the virtual relation “Age-BirthYear” is equivalent to extending the “Doctors” relation with the computed birth year:

```
Proj( [ birth-yr ],
      Extend( [ birth-yr == compute_birth_year(Age) ],
              Doctors ).
```

This is the case only because there is a one-to-one association between ages and birth years. In general, however, the functions defined for virtual relations may be multivalued, so that the mapping specified by the virtual relation cannot be implemented by extending each row with a single value. Therefore, the join is required.

The advantage of representing functional mappings with virtual relations is that a query optimizer may manipulate virtual relations and actual relations similarly. Specifically, joins between virtual relations and actual relations may be re-ordered or deferred by an optimizer so that the costs of executing the functions specified in the virtual relations are reduced. For example, if a select operation greatly reduces the size of an actual relation by excluding many rows, a join between the actual relation and a virtual relation should be deferred until after the select operation to reduce the number of times the function is executed.

Formally, every virtual relation, R_V, defines two disjoint sets of fields, A and B, such that the schema of $R_V = AB$. A mutual functional dependency exists between A and B such that $A = f(B)$ and $B = g(A)$, that is, each value of A in R_V functionally determines the corresponding values of B and each value of B in R_V functionally determines the corresponding values of A. Note that, in general, A and B may each constitute more than one field. The functions f and g are executable functions that map from values of A to sets of values of B and vice versa. The important constraint exists that virtual relations can be joined to actual relations only on all the values of A or all the values of B. This constraint ensures that all of the arguments to the functions that implement the virtual relation are bound when the functions are executed (recall that the values of the fields on which the virtual relation is joined are substituted into the arguments of the functions).

In practice, many useful virtual relations represent functions that have no inverses. An example is the virtual relation "DOB-Age," which maps a precise date-of-birth to a current age in years:

```
DOB-Age =       field        functional definition
                curr-age:    compute_age(dob)
                dob:         <no function>.
```

Because any age in years implies a *range* of possible birth dates, there is no function that can map from an age to a single DOB. However, the useful operation of mapping from a DOB to a single age is possible and is performed by the function "compute_age()." The implication of this asymmetry is that the virtual relation "DOB-Age" may be joined to actual relations only on the field "dob." The virtual relation may not be joined on the field "curr-age." A query optimizer that reorders joins involving virtual relations must be aware of this constraint. For example, given the tables defined in Figures 4.3 and 4.9, the following are valid and invalid relational algebra expressions:

valid: **Proj(** **[** curr-age **],** (see Figure 4.9)
    Join(**[** DOB = dob **],**
        Patient,
        DOB-Age**))**

invalid: **Proj(** **[** dob **],** (see Figure 4.3)
    Join(**[** Age = curr-age **],**
        Doctors,
        DOB-Age**)).**

4.1.3.7 Examples

The complete syntax of the Extended Relational Algebra appears in Appendix A. Examples of relational algebra expressions encoded in the syntax of Table 4.1 appear in the following list (the semantics of these expressions are based on the relational schema shown in Figure 4.9):

Patient

Name	Phone#	DOB
John	(415) 723-7633	12-2-57
Mary	(510) 325-8790	3-25-62

Patient-Doc

Patient	Doctor
John	Gordo
John	Pam
Mary	Colin
Mary	Pam

Doctor

MD-Name	Phone#	DOB
Colin	(415) 494-8767	1-7-39
Gordo	(510) 320-6966	11-17-58
Pam	(510) 390-8754	4-23-49

5. The names and ages of all patients:

```
Proj( [ Name, curr-age ],
      Join( [ DOB = dob ],
            Patient,
            DOB-Age) )
```

(This expression assumes the definition of the virtual relation "DOB-Age" as specified above.)

6. The names and phone numbers of patients who are associated with at least one physician:

```
Proj( [Name, Phone#],
      Join( [ Name = Patient ],
            Patient,
            Exists( Patient-Doc ) ) )
```

7. The number of physicians associated with each patient:

```
Agg( [Doctor, Name, COUNT, Doc-Count],
     Join( [ Name = Patient ],
           Patient,
           Patient-Doc ) )
```

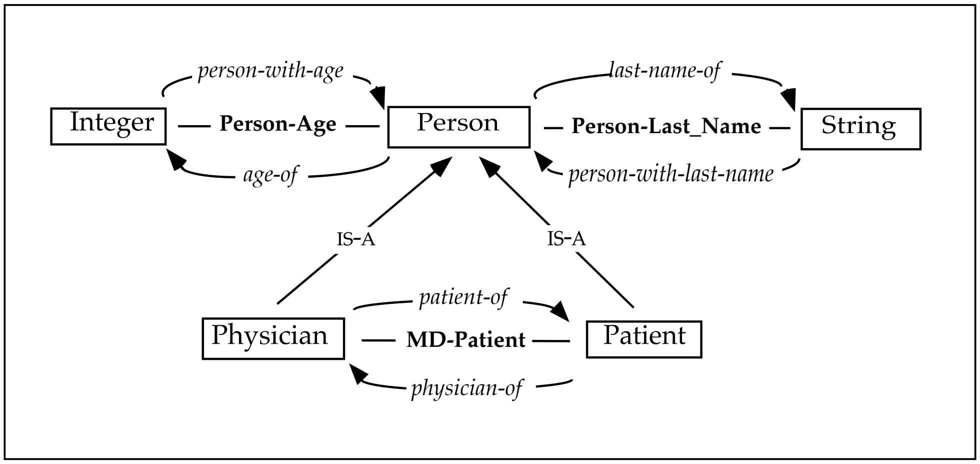
Expressions in the ERA language are generated by the semantic functions of the TransFER attribute grammar. The ERA expressions represent the relational translations of ReFER language constructs in queries parsed by the attribute grammar. The *site-specific* semantic functions generate the ERA translations of FER entity types and relationship types. I discuss these semantic functions in the next section. The *site-independent* semantic functions combine these ERA translations according to the syntax of a particular ReFER query to generate a single ERA expression that constitutes the relational translation of the query. I discuss these semantic functions in Section 4.3.

4.2 Mapping FER Schemas to Relational Databases

As I discussed in Section 4.1.2, certain of the semantic functions in the TransFER attribute grammar are site specific. These semantic functions define translations from ReFER expressions to ERA expressions that vary depending on the clinical database implementation at a particular site. Due to local modeling decisions, relational databases that store similar clinical information may vary from site to site in numerous respects, including the naming of tables and fields, the degree of normalization, the representation of type hierarchies, and the composition of controlled vocabularies. Therefore, the relational algebra translations of certain ReFER expressions also must vary from site to site to represent the semantics of the ReFER expressions accurately. For example, consider the ReFER query Q1, which is specified with respect to the FER schema in Figure 4.10:

Q1: [last-name-of(p) | (p : Person) AND age-of(p) > 55]

Given the semantics of the FER schema and the ReFER query language, the query Q1 requests the last names of all persons in the database over the age of 55.



2. The entity types of the type hierarchy Person-Patient-Physician are represented in one table at Site A but are distributed across two tables at Site B.

3. Persons are uniquely identified by the combination of their first and last names at Site A, whereas persons are uniquely identified by their social security numbers at Site B.

4. Each person's age is explicitly stored at Site A, whereas each person's age must be computed from the stored date of birth at Site B.

5. The associations between patients and their physicians are represented in a separate table solely for this purpose at Site A, whereas the associations are represented in the same table as patients are represented at Site B (which must be denormalized because the patient–physician association is many-to-many).

Despite these differences, note that the information content specified by the FER schema in Figure 4.10 is completely represented in both of the relational schemas. The relational schemas at sites A and B are, therefore, *semantically consistent* with the FER schema (according to the informal definition of semantic consistency that I proposed in Section 4.1).

Site A

PERSON

fname	lname	age	type
Phil	Smythe	38	pt
Carol	Flynn	33	pt
Colin	Krup	56	md
Gordo	Matic	37	md
Pam	Simmons	46	md

PHYSICIAN_ROSTER

pt-fname	pt-lname	md-fname	md-lname
Phil	Smythe	Gordo	Matic
Phil	Smythe	Pam	Simmons
Carol	Flynn	Colin	Krup
Carol	Flynn	Pam	Simmons

Site B

PATIENT

pt-ssn	first	last	dob	md-ssn
149584223	Phil	Smythe	12-2-57	249633805
149584223	Phil	Smythe	12-2-57	136420428
216426077	Carol	Flynn	3-25-62	186757471
216426077	Carol	Flynn	3-25-62	136420428

PHYSICIAN

md-ssn	first	last	dob
186757471	Colin	Krup	1-7-39
249633805	Gordo	Matic	11-17-58
136420428	Pam	Simmons	4-23-49

Note that the set of entity types and relationship types that appears in a TransFER attribute grammar depends on the FER schema for which the attribute grammar is defined. For example, the attribute grammar for the FER schema in Figure 4.10 would include the entity type symbols “Person,” “Patient,” and “Physician” and the relationship type symbols “Person-Last_Name,” “Person-Age,” and “MD-Physician” (the built-in entity types Integer and String are included by default).

The semantic functions for the entity types and relationship types of an FER schema assign values to the *attributes* of the entity-type and relationship-type symbols in the TransFER attribute grammar. In other words, each entity-type and relationship-type symbol has a set of attributes, the values of which are assigned by the semantic functions. Specifically, entity-type symbols have the attributes “ERA” and “Id,” and relationship-type symbols have the attributes “ERA,” “Id,” and “Fid.” The values of these attributes instruct the compiler how to assemble a relational algebra expression at the local database that is equivalent to the corresponding entity type or relationship type in the FER schema. In this section, I introduce the formal method by which the attributes specify these instructions; this method constitutes the *mapping language* of TransFER. In Appendix A, I present the complete specifications of the mapping language and I characterize the power of this language to map FER schemas to a variety of heterogeneous relational database schemas. When a ReFER query is parsed, the attribute grammar assembles the relational algebra expressions that correspond to the entity and relationship types within the query according to the site-specific mappings and combines these expressions to generate a complete relational algebra translation of the query (as described in Section 4.1.2).

4.2.1 Entity-type mappings

Recall that in the FER data model, entity types represent sets of unique objects in the domain of discourse. In order to map this concept to a relational database, the semantic function for each entity-type symbol, E, assigns an extended relational algebra expression, R, to the attribute “ERA” such that *each row in R represents a unique instance* of the entity type E. For example, the following are the assignments of ERA for the entity types of the FER schema in Figure 4.10 and the relational database implementations at sites A and B:

Site A

Person ::= ERA = " PERSON"
Patient ::= ERA = "Sel([type = "pt"], PERSON)"
Physician ::= ERA = "Sel([type = "md"], PERSON)"

Site B

```
Person ::=  ERA = "Union( Proj([ssn],
                              Rename([pt-ssn -> ssn],PATIENT)),
                         Proj( [ssn],
                              Rename([md-ssn -> ssn],PHYSICIAN)))"
```
Patient ::= ERA = "Proj([pt-ssn], PATIENT)"
Physician ::= ERA = "PHYSICIAN".

Additionally, the semantic function for each entity-type symbol, E, assigns a list of field names, F_I, to the attribute "Id" such that F_I denotes the *key* of the relational algebra expression for the entity type E. By the definition of "keyness," each element of F_I must be a field in the corresponding ERA expression, R, and the concatenated values of F_I for each row of R must be unique. For example, the following are the assignments of the attribute "Id" for the entity types of the FER schema in Figure 4.10 and the relational database implementations at sites A and B:

Site A

Person ::= Id = [lname, fname]
Patient ::= Id = [lname, fname]
Physician ::= Id = [lname, fname]

Site B

Person ::= Id = [ssn]
Patient ::= Id = [pt-ssn]
Physician ::= Id = [md-ssn].

Several points regarding the mappings for Person, Patient, and Physician at sites A and B are noteworthy:

1. The SELECT operator is required in the ERA values of the entity types Patient and Physician at site A because the instances of Patient and Physician are represented together in the table `PERSON`.

2. The UNION operator is required in the ERA value of the entity type Person at site B because the instances of Person are distributed across the tables `PATIENT` and `PHYSICIAN`.

3. The RENAME operator is required in the ERA value of the entity type Person at site B because the field names denoting the keys of the tables `PATIENT` and `PHYSICIAN` are different in each table (that is, "`pt-ssn`" and "`md-ssn`"); because a single field name is required as the value of Id for Person, the keys must be "unified" before the tables are unioned.

4. The PROJECT operator is required in the ERA value of the entity type Patient at site B because the table `PATIENT` is denormalized and therefore may contain multiple rows for each Patient entity; the PROJECT operator eliminates duplicates and ensures that the Id value, `[pt-ssn]`, is unique.

Together, the values of the attributes "ERA" and "Id" values for each entity-type symbol completely specify how that entity type is represented in a particular relational database.

4.2.2 Relationship-type mappings

In the FER data model, relationship types represent sets of unique objects that associate pairs of entities. In order to map this concept to a relational database, the semantic function for each relationship-type symbol, A, assigns an extended relational algebra expression, R, to the attribute "ERA" such that each row in R represents a unique instance of the association denoted by the relationship type A. Because each instance of a relationship type in the FER data model represents an association between two entities, the semantic function also assigns a paired list of field names, $<F_1, F_2>$, to the attribute "Fid;" F_1 and F_2 denote the fields of the relational algebra expression R that contain the identifiers of the entities associated by each row of R. Each field list, F_1 and F_2, is annotated with the appropriate entity type of the entity that it represents. For example, the following are the assignments of ERA and Fid for the relationship types of the FER schema in Figure 4.10 and the relational database implementations at sites A and B (it is useful to review the FER schema in Figure 4.10 and the relational database schemas in Figure 4.11 when inspecting these specifications):

Site A

```
Person-Last_Name ::=  ERA = " PERSON"
                      Fid = { Person([lname,fname]),
                              String([lname]) }
Person-Age ::=    ERA = " PERSON"
                  Fid = { Person([lname,fname]),
                          Integer([age]) }
MD-Patient ::=  ERA = "PHYSICIAN-ROSTER"
                 Fid = { Physician([md-lname,md-fname]),
                         Patient([pt-lname,pt-fname]) }
```

Site B

```
Person-Last_Name::=
            ERA = "Union( Proj([ssn,last],
                               Rename([pt-ssn -> ssn],PATIENT)),
                          Proj( [ssn,last],
                               Rename([md-ssn -> ssn],PHYSICIAN) ) )"
             Fid = { Person([ssn]),
                     String([lname]) }
Person-Age::=
        ERA = "Join([dob = date-of-birth],
                     Union( Proj([ssn,dob],
                               Rename([pt-ssn -> ssn],PATIENT)),
                           Proj( [ssn,dob],
                               Rename([md-ssn -> ssn],PHYSICIAN) ) ),
                     DOB-AGE)"
        Fid = { Person([ssn]),
                Integer([age]) }
MD-Patient ::=
        ERA = "PATIENT"
        Fid = { Physician([md-ssn]),
                Patient([pt-ssn]) }.
```

Note how the value of Fid for the relationship type MD-Patient at site A consists of one field list annotated by “Physician” and one field list annotated by “Patient” because the relationship type MD-Patient associates instances of the entity type Physician with instances of the entity type Patient . In each row of the relational algebra expression for MD-Patient, the respective values of the fields [`md-lname, md-fname`] and [`pt-lname, pt-fname`] represent the identities of the physician and the patient associated by that instance of the relationship type. Several other points regarding the mappings for Person-Last_Name, Person-Age, and MD-Patient at sites A and B are noteworthy:

1. At site A, one of the lists of field names in the Fid value is a subset of the other list. This is appropriate because the field [`lname`] that represents each person's last name in the table `PERSON` is part of the key of that table, [`lname,fname`], which represents each person's identity. In other words, the last name associated with each person at site A effectively can be derived from the unique identifier of the person.

2. At site B, the UNION operator is required in the ERA values of the relationship types Person-Last_Name and Person-Age because those relationship types associate *all* instances of Person (whether they are patients or physicians) with their names and their ages.

3. Because site B does not explicitly store the ages of patients and physicians, the ERA value of the relationship type Person-Age requires a join to the virtual table `DOB-AGE` in order to derive the age (Section 4.1.3.6 discusses virtual tables). `DOB-AGE` is a virtual table that implements the function “compute_age(),” which computes the difference in years between an arbitrary date and the current date. The virtual table “`DOB-AGE`” and the function “compute_age()” must be defined at the time the mappings are specified. For example, the following definition of `DOB-AGE` is consistent with the mapping for Person-Age at site B:

```
DOB-Age = field             functional definition
          date-of-birth:    <no function>
          age:              compute_age(date-of-birth).
```

Because relationships are themselves objects in the FER data model, each mapping for a relationship type must denote the unique identifier of the relationship instances. Therefore, the semantic function for each relationship-type symbol, A, assigns a list of field names, F_I, to the attribute "Id" such that F_I denotes the *key* of the relational algebra expression for the relationship type A. The same constraints apply to the Id values of relationship types as apply to the Id values of entity types. The Id values of relationship types are important when the relationship types themselves participate in relationship types, as the FER data model allows. The following are the assignments of Id for the relationship types of the FER schema in Figure 4.10 and the relational database implementations at sites A and B:

Site A
Person-Last_Name ::= Id = [`lname,fname`]
Person-Age ::= Id = [`lname,fname,age`]
MD-Patient ::= Id = [`md-lname,md-fname,`
`pt-lname,pt-fname`]

Site B
Person-Last_Name ::= Id = [`ssn,last`]
Person-Age ::= Id = [`ssn,age`]
MD-Patient ::= Id = [`pt-ssn,md-ssn`].

Together, the ERA, Fid, and Id values for each relationship-type symbol completely specify how that relationship type is represented in a particular relational database.

The mappings for entity-type and relationship-type symbols, which are defined at each site in the manner I have just described, assign values to the terminal symbols of a TransFER attribute grammar, thereby "customizing" the attribute grammar to a particular site. The assigned values, which consist of relational algebra expressions (the ERA values) and information about those expressions (the Id and Fid values), subsequently may be combined and manipulated by the site-independent semantic functions of the attribute grammar to produce relational translations of arbitrary ReFER queries. In the following section, I introduce the site-independent semantic functions and describe how they translate valid ReFER expressions to semantically equivalent SQL queries.

4.3 Translating ReFER queries to SQL queries

Recall from Section 3.3.7 that any ReFER query of the form

Query ::= '[' Termlist ' | ' Decllist 'AND' Predicate ']',

in which Termlist, Decllist, and Predicate conform to the specifications of the ReFER language, has the formal meaning

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

where B(Decllist, Predicate) = { b : b ∈ 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. I will refer to the term list, the declaration list, and the predicate that make up a ReFER query as the Termlist, Decllist, and Predicate *components* of the query, respectively. For example, the ReFER query

"[last-name-of(pt), last-name-of(md) | (pt : Patient), (md : Physician) AND
MD-Patient(md,pt) AND
age-of(pt) > 55"

returns the last names of each patient–physician pair in the database such that the physician is treating the patient and the patient is older than 55. The Termlist component is "last-name-of(pt), last-name-of(md)," the Decllist component is "(pt : Patient), (md : Physician)," and the Predicate component is "MD-Patient(md,pt) AND age-of(pt) > 55."

To translate the ReFER query accurately to an SQL query, TransFER must reproduce the semantics embodied in the formal definition of the ReFER query in a corresponding relational algebra expression. TransFER subsequently must translate the relational algebra expression to SQL. I discuss the first step in Sections 4.3.1–3 and the second step in Section 4.3.4.

4.3.1 Translating to the relational algebra: an overview

A ReFER query is translated to a relational algebra expression by the site-independent semantic functions of the TransFER attribute grammar, which combine the site-specific mappings for entity types and relationship types to reproduce the meaning of any ReFER query. The site-independent semantic functions perform this translation by first specifying relational algebra *subexpressions* for the Termlist, the Decllist, and the Predicate components of the ReFER query; the semantic functions subsequently combine the subexpressions by applying the JOIN operator. The result is a relational algebra expression that reproduces the formal meaning of the ReFER query. I will describe how the semantic functions specify the appropriate subexpressions for the Termlist, Decllist, and Predicate components of a ReFER query in Sections 4.3.2 and 4.3.3. In this section, I describe informally the relational algebra subexpressions generated for the Termlist, Decllist, and Predicate components, and I describe informally how the subexpressions are combined to reproduce the semantics of the original ReFER query. I hope to provide the reader with a high-level understanding of the translation process before I delve into the details of the translation mechanism.

4.3.1.1 Translations of ReFER query components

The meanings of the relational algebra subexpressions that are generated by the semantic functions for the Termlist, the Decllist, and the Predicate components of a ReFER query may be described informally as follows:

1. *Decllist*: For the Decllist component, the semantic functions generate the *Cartesian product* of the relational algebra mappings for each entity type that appears in the declaration list. For example, the Decllist component

   "(pt : Patient), (md : Physician)"

   of the sample query contains the entity types Patient and Physician; therefore, the semantic functions generate the Cartesian product of the relational algebra mappings for the entity type Patient and the entity type Physician. Recall that the mapping for each entity type specifies a relational algebra expression (that is, a relational table) such that each row of the table represents one instance of the mapped entity type (see Section 4.2.1). I shall call these tables "mapping tables." Each row of the mapping table for an entity type, therefore, represents one *binding* of the variable that is declared to range over that entity type. The Cartesian product of the mapping tables for each entity type in the declaration list, therefore, constitutes the Cartesian product of the bindings defined by each declaration in the declaration list. This construction reproduces precisely the semantics of Decllist in the ReFER query language (see Section 3.3.3).

2. *Predicate*: For the Predicate component, the semantic functions generate a relational algebra expression that contains one row for each binding of the variables in the predicate for which the predicate holds. For example, the Predicate component

   "MD-Patient(md,pt) AND age-of(pt) > 55"

   of the sample query results in a relational algebra expression that contains one row for each binding of the variables "md" and "pt" such that the physician bound to "md" is treating the patient bound to "pt" and the age of the patient bound to "pt" is greater than 55. Note that by *not* containing any rows that represent bindings of "md" and "pt" for which the predicate fails, the relational algebra expression effectively "filters out" all such bindings. This expression, therefore, reproduces the semantics of Predicate in the ReFER query language, as defined in Section 3.3.6. I describe the way in which the semantic functions of the attribute grammar filter out inappropriate bindings in Section 4.3.3.2.

3. *Termlist*: For the Termlist component, the semantic functions generate a relational algebra expression that contains the values of the term list computed for *all* bindings of the variables that appear in the term list. In other words, the semantic functions perform no filtering of variable bindings, although such filtering may be specified in the Predicate clause of the query. For example, the Termlist component

   "last-name-of(pt), last-name-of(md)"

of the sample query results in a relational algebra expression that contains one row for each pair of bindings of "pt" and "md"; the row contains the unique identifiers of the patient and the physician entities to which the variables are bound and the last names associated with these patient and physician entities. At a database site where social security numbers are used to identify persons, the Termlist of the sample query might result in the following relational table:

pt-ssn	pt-last-name	md-ssn	md-last-name
1234	Smythe	5678	Collins
5432	Brooks	9876	Matic
1234	Smythe	4444	Collins

Note that the table corresponding to the term list contains one row for every binding of "pt" and "md" although the query in which the term list appears includes the predicate "age-of(pt) > 55," which specifies that only certain bindings of "pt" should be used to compute the results of the query. The filtering specified by the predicate is applied to the term list later.

4.3.1.2 Combining the translations of ReFER query components

Given the meanings of the relational algebra subexpressions generated for the Termlist, Decllist, and Predicate components of a ReFER query, the semantic functions of the attribute grammar reproduce the meaning of the entire ReFER query by combining the subexpressions in the following manner:

1. The subexpression for the Decllist is joined to the subexpression for the Predicate. Because the Decllist subexpression contains one row for each binding of the declared variables whereas the Predicate subexpression contains rows for only those bindings that satisfy the predicate, the effect of the join is to produce the *intersection* of the set of all bindings and the set of bindings that satisfy the predicate. The resulting expression contains one row for each binding that (1) is defined by the declaration list and (2) satisfies the predicate; this set of bindings is, in fact, equivalent to the set of bindings for which the Termlist should be

evaluated according to the semantics of the ReFER query language (as defined in Section 3.3.7):

$$\{ b : b \in B(\text{Decllist}) \text{ and } \tau[\text{Predicate}, b] = \text{TRUE}\}.$$

For example, in the sample query, the relational subexpression for the Decllist "(pt : Patient), (md : Physician)" is joined to the relational subexpression for the Predicate "MD-Patient(pt,md) AND age-of(pt) > 55." The subexpression for the Decllist contains the Cartesian product of all Patient and Physician entities, whereas the subexpression for the Predicate contains only the pairs of Physician and Patient entities such that the entities are associated by the MD-Patient relationship type and the Patient entity is over the age of 55. The subexpressions are joined on the respective fields in each expression that contain the identifiers of the Physician and Patient entities (the semantic functions of the attribute grammar keep track of these sets of field names for each expression, as described in Section 4.3.2.1). Recall that the join of two tables simply drops all rows in either table that do not have corresponding rows in the other table. In this manner, the join "filters out" all of the bindings defined by the declaration list that do not satisfy the predicate. A join of the subexpressions for the Decllist and the Predicate of the sample query, therefore, produces the pairs of Patient and Physician entities such that the Physician entity is treating the Patient entity and the Patient entity has an age greater than 55. I call the relational algebra expression produced by joining the Decllist and the Predicate subexpressions the "Range." The Range expression contains precisely the set of bindings for which the term list should be evaluated according to the semantics of the ReFER query language.

2. The expression for the Termlist is joined to the Range expression. Because the Termlist expression contains the values of the term list computed for all bindings, whereas the Range expression contains only those bindings that satisfy the predicate, joining the expressions produces the values of the term list computed for only those bindings that satisfy the predicate. For example, in the sample query, the relational subexpression for the Termlist clause

"last-name-of(pt), last-name-of(md)"

is joined to the Range subexpression, which contains only those pairs of bindings for "pt" and "md" that satisfy the Predicate. The subexpressions are joined on the respective fields in each expression that contain the identifiers of the Patient and the Physician entities (the semantic functions of the attribute grammar keep track of these field names for each expression). Again, the effect of the relational join is to "filter out" all of the rows in the Termlist expression containing bindings of "pt" and "md" that do not appear in the Range expression. The resulting expression contains the values of the term list computed only for the set of variable bindings that appear in the Range expression. This is almost the correct result of the query, but one more operation is required.

3. The values of the Termlist expression are multiprojected. Recall that, for the sample query, each row of the Termlist expression contains the bindings of "pt" and "md" as well as the pair of last names associated with those bindings. The meaning of the query, however, is represented by only the pair of last names, so the multiproject operator must be applied to return only those fields. A multiprojection is specified rather than a projection because the formal meaning of ReFER queries is specified as the *multiunion* of generated values (see Section 3.3.7).

   For example, the result of the sample query is computed by multiprojecting only the fields "pt-last-name" and "md-last-name":

pt-last-name	md-last-name
Smythe	Collins
Brooks	Matic
Smythe	Collins

   The multiprojection is the final step in the translation process and the resulting relational algebra expression accurately represents the semantics of the ReFER query.

By first generating relational algebra subexpressions with the appropriate meanings for the Termlist, Decllist, and Predicate components of a ReFER query and by subsequently joining these subexpressions on the appropriate fields, the semantic functions of the attribute grammar succeed in reproducing the semantics of the original ReFER query. In the following section, I describe in detail how the semantic functions of the attribute grammar generate the correct relational algebra subexpressions for the Termlist, Decllist, and Predicate components and how the semantic functions manage the names of the relevant fields in each subexpression.

4.3.2 Site-independent semantic functions

The site-independent semantic functions of a TransFER attribute grammar combine the relational algebra expressions defined by the site-specific semantic functions to generate relational translations of arbitrary ReFER queries. Like the site-specific semantic functions, each site-independent semantic function generates a relational algebra expression (that is, table) and relevant information about that expression. Unlike the site-specific semantic functions, the site-independent semantic functions are not customized for specific relational database schemas; the same set of semantic functions appears in the TransFER attribute grammar at every site. The site-independent semantic functions are also different from their site-specific counterparts in that they may specify expressions that include *references to the attributes of grammar symbols assigned by other semantic functions*. For example, the following semantic function for the grammar symbol “Predicate” includes references to the attributes of the grammar symbols “Term” and “Op” (the context-free grammar rule for the symbol “Predicate” appears first, followed by the semantic function for that symbol's attribute “Predicate.ERA.”):

Grammar rule: Predicate ::= Term_1 Op Term_2

Semantic function: Predicate.ERA = Join([Term_1.Id Op.value Term_2.Id],
Term_1.ERA,
Term_2.ERA)

Note that the arguments to the relational algebra JOIN operator are not relational tables, fields, and comparators (as the syntax of the relational algebra specifies), but *references* to the attributes of grammar symbols on the right-hand side of the grammar rule. The values of these attributes represent relational tables, fields, and comparators; they must be computed recursively by the semantic functions for the grammar symbols “Term” and “Op.” I will show how these references are used to generate valid relational algebra expressions in Section 4.3.3.

4.3.2.1 The attribute “Vid”

In Section 4.2, I described how site-specific semantic functions assign values to the attributes “ERA,” “Id,” and “Fid” for grammar symbols that denote entity types and relationship types. Site-independent semantic functions assign values to the same attributes for all of the other symbols in the ReFER attribute grammar (such as “Term,” “Predicate,” and “Decl”). Additionally, site-independent semantic functions assign values to an attribute named “Vid.” The value of the Vid (“variable Id”) attribute denotes the fields of a relational algebra expression that represent the *bindings* of the variables that appear in the corresponding ReFER expression. The value of Vid is a *set of pairs*, in which each element has the form “(var, field-list),” where “var” denotes a variable and “field-list” denotes the relational fields that represent the bindings of that variable. For example, in Section 4.3.1.1, I presented the following table to represent the relational translation of the Termlist clause “last-name-of(pt), last-name-of(md)” for a database site in which social security numbers are used to identify persons:

pt-ssn	pt-last-name	md-ssn	md-last-name
1234	Smythe	5678	Collins
5432	Brooks	9876	Matic
1234	Smythe	4444	Simmons.

The table contains the last names of pairs of Patients and Physician entities. The value of the Vid attribute for this relational algebra expression is

{ (pt, [`pt-ssn`]), (md, [`md-ssn`]) },

because the ReFER term list “last-name-of(pt), last-name-of(md)” contains the variables “pt” and “md,” and the bindings of these variables in the relational table are represented by the values of the fields “`pt-ssn`” and “`md-ssn`,” respectively. By denoting which fields of a relational algebra expression represent the bindings of the variables that appear in the corresponding ReFER expression, the TransFER attribute grammar can filter out unwanted bindings by specifying SELECT and JOIN operations that involve those fields. This capability is critical for reproducing the semantics of ReFER queries in relational algebra expressions.

4.3.2.2 Site-independent semantic functions: an example

I now introduce the structure and operation of site-independent semantic functions through an example. The following semantic functions generate the relational translation of ReFER predicates that compare two terms, such as "pt = md" (again, the context-free grammar rule for the symbol appears first, followed by the semantic functions that define each attribute of the symbol):

Grammar rule: Predicate ::= Term_1 Op Term_2

Semantic functions: Predicate.ERA = Join([Term_1.Id Op.value Term_2.Id],
Term_1.ERA,
Term_2.ERA)
Predicate.Id = ∅
Predicate.Vid = Term_1.Vid ∪ Term_2.Vid
Predicate.Fid = ∅

The grammar rule parses the ReFER predicate "pt = md" because "pt" and "md" are variables (which are terms) and "=" is a valid comparison operator. The semantic function for the attribute "ERA" specifies a JOIN operation that references the Id and ERA attributes of the two terms that are compared in the predicate. The semantic functions that define the Id and ERA attributes of these two terms may themselves reference the attributes of other terms. The recursion terminates when the grammar reaches a symbol that represents an entity type or a relationship type, because the semantic functions for entity-type and relationship-type symbols specify operations over the *base* tables and fields of a relational database (see Section 4.2). The semantic function for the attribute "Vid" computes the union of the Vid values for the two terms that are compared by the predicate, thereby producing a set of field lists that represents the bindings of all variables that appear in the ReFER predicate. For example, if the Vid value of Term_1 were { (pt, [`pt-ssn`]) } and the Vid value of Term_2 were { (md, [`md-ssn`]) }, the union of these values would produce the Vid value { (pt, [`pt-ssn`]), (md, [`md-ssn`]) }. Lastly, the semantic functions for the attributes "Id" and "Fid" indicate that the values of these attributes are NULL because predicates (unlike terms) have no unique identifiers or foreign identifiers.

As an example of how the semantic functions operate, consider the relational translation of the ReFER expression "pt = md".

The grammar rule parses this expression as a Predicate clause, with "pt" as Term_1, "md" as Term_2, and "=" as Op. The semantic functions for the nonterminal symbols "Term" and "Op" assign appropriate values to each attribute of "pt," "md," and "=" (the semantic functions for "Term" and "Op" appear in Appendix B). Let us assume that the semantic functions have assigned the following attribute values for the expressions "pt," "md," and "=" (these assignments are consistent with the entity-type mappings for the relational schema Site B in Figure 4.11, assuming that the variable "pt" is declared to range over the entity type Patient and the variable "md" is declared to range over the entity type Physician):

"pt": Term.ERA = Proj([`pt-ssn`], `PATIENT`)
Term.Id = [`pt-ssn`]
Term.Vid = { (pt, [`pt-ssn`]) }
Term.Fid = ∅

"md": Term.ERA = `PHYSICIAN`
Term.Id = [`md-ssn`]
Term.Vid = { (md, [`md-ssn`]) }
Term.Fid = ∅

"=": Op.value = "="

When the attribute grammar substitutes the referenced values of the attributes into the semantic function definitions, the semantic functions generate the following relational algebra translation for the predicate "pt = md":

"pt = md": Predicate.ERA = Join([`pt-ssn = md-ssn`],
Proj([`pt-ssn`], `PATIENT`),
`PHYSICIAN`)
Predicate.Id = ∅

Predicate.Vid = { (pt, [`pt-ssn`]), (md, [`md-ssn`]) }
Term.Fid = ∅

Informally, the relational algebra translation of the predicate "pt = md" joins the table representing the Patient entity type with the table representing the Physician entity type on those fields that uniquely identify Patient and Physician entities in the respective tables; the result is a table that contains only rows in which the field that represents the binding of the variable "pt" equals the field that represents the binding of the variable "md"; that is, the table contains only rows representing pairs of Patient and Physician entities such that the Patient entity is the same as the Physician entity. In other words, the relational algebra translation of "pt = md" generates all the bindings of "pt" and "md" such that the predicate "pt = md" is true. This set of bindings is used later in the query-translation process to filter out the bindings for which the predicate does not hold (as described in Section 4.3.1). Note that the relational database at Site B in Figure 4.11 contains no persons who are both physicians and patients, so that no bindings "pt" and "md" satisfy the predicate "md = pt"; the relational algebra expression generated by the semantic functions for the predicate "pt = md," therefore, will return no rows at Site B.

4.3.2.3 Expanding predicates in relational algebra expressions

We have seen that site-independent semantic functions for the attribute "ERA" may define conditions in JOIN expressions that reference the Id attributes of other grammar symbols. For example, the JOIN condition in the semantic function

$$\begin{aligned}\text{Predicate.ERA} = \quad & \text{Join}(\ [\text{Term}_1.\text{Id Op.value Term}_2.\text{Id}],\\ & \quad \text{Term}_1.\text{ERA},\\ & \quad \text{Term}_2.\text{ERA}\ )\end{aligned}$$

references the Id attributes of Term_1 and Term_2. In general, JOIN conditions in site-independent semantic functions may compare the Id, Vid, or Fid attributes of grammar symbols. Because the values of these attributes, in general, may be *lists* of field names (rather than a single field name, as was the case in Section 4.3.2.2), the values of the attributes (such as [`fname, lname`]) cannot simply replace *syntactically* the corresponding references to these values (such as Term_1.Id) because the results may not be valid relational algebra expressions. For example, if Term_1.Id = [`fname,lname`] (a list of two fields) and Term_2.Id = [`first,last`] (also a list of two fields), the syntactic substitution of Term_1.Id and Term_2.Id with their corresponding field lists in the condition [Term_1.Id = Term_2.Id] produces the condition

[`fname,lname` = `first,last`].

This expression is not a valid JOIN condition in the syntax of relational algebra. To generate valid relational algebra JOIN conditions, the semantic functions must *expand* comparisons of attribute values that consist of multiple field names into *conjunctions* of comparisons for each pair of corresponding field names:

[`fname,lname` = `first,last`] ⟹ [`fname` = `first` and `lname` = `last`].

The semantic functions perform a similar expansion for cases where the values of Fid attributes are compared. Because the values of Vid attributes may be *sets* of field lists, in which each field list is associated with a variable, the expansion of Vid values is a little different. Specifically, the semantics of the predicate

[Term_1.Vid = Term_2.Vid]

specify that the field lists associated with each variable that *is common to both* Vid values are compared, and that the predicate expressions for each such comparison are conjoined into a single predicate. For example, if

Term_1.Vid ={ (pt, [`fname,lname`]), (md, [`first,last`]) } and
Term_2.Vid ={ (pt, [`Fst_name,Lst_name`]) },

then the expression [Term_1.Vid = Term_2.Vid] is expanded into

[`fname` = `Fst_name` and `lname` = `Lst_name`],

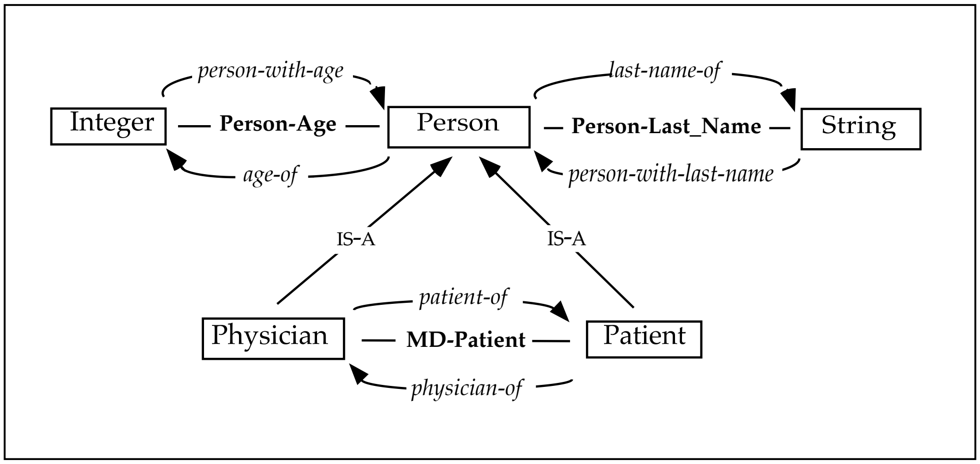
because the Vid values for Term_1 and Term_2 have the variable “pt” in common, and the field lists associated with the variable “pt” are [`fname,lname`] and [`Fst_name,Lst_name`], respectively. The rationale for these expansion rules will become evident with the sample query translation that appears in the next section.

4.3.3 A sample translation

I will show how the site-independent semantic functions of the TransFER attribute grammar perform query translations in general by tracing the translation of a sample ReFER query at two different database sites. The sample ReFER query requests the last names of all patients who are over the age of 35:

[last-name-of(pt) | (pt : Patient) AND age-of(pt) > 35].

The sample query is formulated with respect to the FER schema in Figure 4.12 (recall that the entity type Patient inherits the relationship types Person-Age and Person-Last_Name).



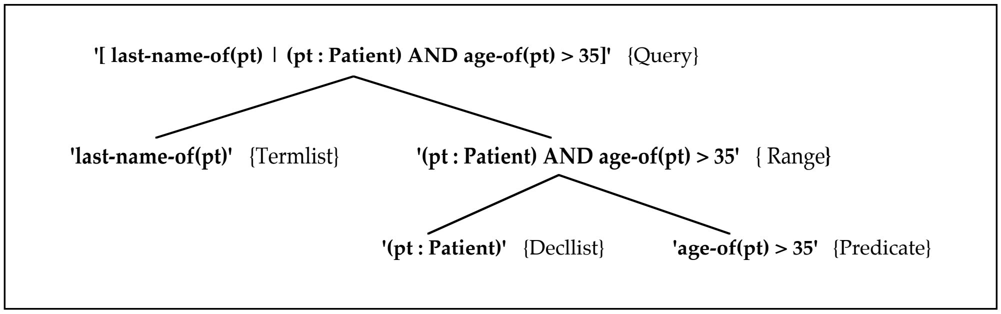


Figure 4.13: A portion of the parse tree for a sample ReFER query The context-free grammar of the ReFER query language decomposes the query into a Termlist, Decllist, and Predicate component (labeled in braces). The combination of the Decllist and Predicate components constitutes the Range component.

In the following sections, I will show how the semantic functions of the attribute grammar generate the appropriate relational algebra expressions for each of the three components and how the semantic functions combine the expressions to reproduce the semantics of the ReFER query. I will trace the operation of the TransFER attribute grammar with respect to each of the relational databases that appear in Figure 4.14. Note that the relational schemas in Figure 4.14 are the same schemas for which I formulated site-specific mappings in Section 4.2. The site-independent semantic functions that I describe in the following sections will use these site-specific mappings to generate the correct relational translations at each site for the Termlist, Decllist, and Predicate components of the query.

Site A

PERSON

fname	lname	age	type
Phil	Smythe	38	pt
Carol	Flynn	33	pt
Colin	Krup	56	md
Gordo	Matic	37	md
Pam	Simmons	46	md

PHYSICIAN_ROSTER

pt-fname	pt-lname	md-fname	md-lname
Phil	Smythe	Gordo	Matic
Phil	Smythe	Pam	Simmons
Carol	Flynn	Colin	Krup
Carol	Flynn	Pam	Simmons

Site B

PATIENT

pt-ssn	first	last	dob	md-ssn
149584223	Phil	Smythe	12-2-57	249633805
149584223	Phil	Smythe	12-2-57	136420428
216426077	Carol	Flynn	3-25-62	186757471
216426077	Carol	Flynn	3-25-62	136420428

PHYSICIAN

md-ssn	first	last	dob
186757471	Colin	Krup	1-7-39
249633805	Gordo	Matic	11-17-58
136420428	Pam	Simmons	4-23-49

The relational algebra expression (that is, ERA value) generated by the semantic functions for each declaration is equivalent to the relational algebra expression specified in the mapping for the entity type that appears in the declaration. For example, the relational algebra translation of the declaration "(pt : Patient)" is equivalent to the relational mapping of the entity type Patient. The semantic functions generate a Vid value for each declaration that associates the declared variable with the key fields of the relational algebra translation for the entity type in the declaration; this association specifies that the binding of the declared variable in each row of the relational algebra expression is expressed by the values of the key fields. As a side effect, the semantic functions also enter the association between the variable and its declared entity type in a symbol table so that other semantic functions that manipulate the variable can determine the entity type to which it corresponds.

When a declaration list consists of only one declaration, there is no need to compute a Cartesian product; the relational algebra expression for the declaration list is identical to the expression for the declaration. This equivalence is represented by the following semantic functions:

Grammar rule: Decllist ::= Decl

Semantic functions: Decllist.ERA = Decl.ERA
Decllist.Id = Decl.Id
Decllist.Vid = Decl.Vid
Decllist.Fid = Decl.Fid

Given these definitions and the mappings for the entity type Patient specified in Section 4.2, the semantic functions generate the following relational algebra translations for the declaration list "(pt : Patient)" at Site A and Site B:

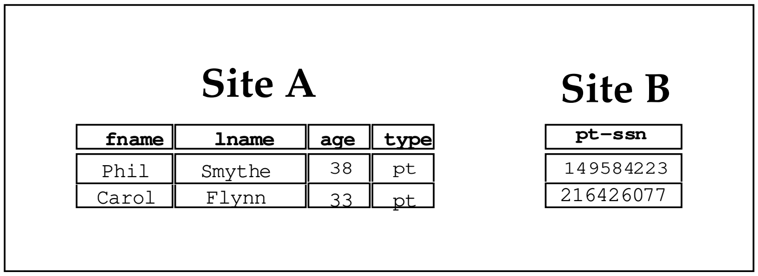
Site A

"(pt : Patient)" : Decllist.ERA = Sel([`type` = "`pt`"], `PERSON`)
Decllist.Id = ∅
Decllist.Vid = { (pt, [`lname`, `fname`]) }
Decllist.Fid = ∅

Site B

“(pt : Patient)” : Decllist.ERA = Proj([`pt-ssn`], `PATIENT`)
Decllist.Id = ∅
Decllist.Vid = { (pt, [`pt-ssn`]) }
Decllist.Fid = ∅

Figure 4.15 shows the tables generated by the semantic functions for the declaration list “(pt : Patient)” at site A and site B. Note that the table at each site contains one row for each patient that appears in the database. These tables represent the relational algebra translations for the Decllist component of the query.



Site A

fname	lname	age	type
Phil	Smythe	38	pt
Carol	Flynn	33	pt

Site B

pt-ssn
149584223
216426077

Figure 4.15: The relational algebra translations of the declaration list “(pt : Patient)” for the two relational databases shown in Figure 4.14.

4.3.3.2 The Predicate component

Recall from Section 4.3.1.1 that the relational translation of a predicate contains all bindings of the variables in the predicate for which the predicate is true. To compute this set, the semantic functions of the attribute grammar decompose predicates into their constituent syntactic parts, generate relational algebra expressions for each part, and subsequently combine the expressions such that only those bindings that satisfy the predicate appear in the result. The parse tree for the ReFER predicate “age-of(pt) > 55,” which is shown in Figure 4.16, shows that the predicate is decomposed into a variable, a function, a comparison operator, and a constant.

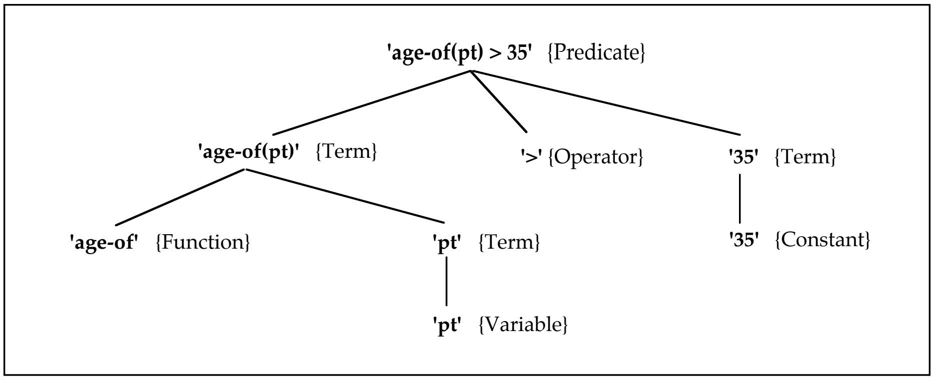


Figure 4.16: The parse tree for the Predicate component of the ReFER query. The context-free grammar decomposes the Predicate component into its constituent Function, Variable, Operator, and Constant components (labeled in braces).

The relational algebra expression for the variable “pt” is specified by the following semantic functions (recall that variables are, themselves, terms):

Grammar rule: Term ::= Var

Semantic functions: Entity_Type = Lookup_Symbol_Table(Var.value)
Term.ERA = Entity_Type.ERA
Term.Id = Entity_Type.Id
Term.Vid = { (Variable.value, Entity_Type.Id) }
Term.Fid = Entity_Type.Fid

For each variable, the semantic functions generate a relational algebra expression that is equivalent to the mapping specified for the entity type over which the variable ranges. The first semantic function retrieves the identity of this entity type from the symbol table. The semantic functions generate a Vid value that associates the variable with the key fields of the relational algebra mapping for the entity type. Given these definitions and the mappings for the entity type Patient specified in Section 4.2, the semantic functions generate the following relational algebra translations of the term “pt” at site A and site B:

Site A

“pt” ::= Term.ERA = Sel([`type = “pt”`], `PERSON`)
Term.Id = ∅
Term.Vid = { (pt, [`lname, fname`]) }
Term.Fid = ∅

Site B

“pt” ::= Term.ERA = Proj([`pt-ssn`], `PATIENT`)
Term.Id = ∅
Term.Vid = { (pt, [`pt-ssn`]) }
Term.Fid = ∅

The relational algebra translation of a term consisting of a relationship function applied to a variable (such as “age-of(pt)”) is specified by the following semantic functions:

Grammar rule: Term_1 ::= Function '(' Term_2 ')'

Semantic functions: Relationship = Function.relationship
Source_Type = Function.source
Target_Type = Function.target
Term_1.ERA = Join([Term_2.Id =
Relationship.Fid(Source_Type)],
Term_2.ERA,
Relationship.ERA)
Term_1.Id = Relationship.Fid(Target_Type)
Term_1.Vid = Term_2.Vid
Term_1.Fid = ∅

Recall that each relationship function of an FER schema is associated with an FER relationship type and maps from one FER entity type (the “source” entity type) to another FER entity type (the “target” entity type). For example, the relationship function “age-of” is associated with the relationship type “Person-Age,” and maps from the source entity type “Person” to the target entity type “Integer” (see the FER schema in Figure 4.12). The first three semantic functions determine the relationship type, source entity type, and target entity type associated with a relationship function, as specified by the FER schema. The values of these parameters are substituted into the semantic functions that determine the ERA, Id, and Vid attribute values for the symbol “Term_1.” The semantic functions evaluate references of the form “Relationship.Fid(<entity type>)” by returning the list of fields corresponding to <entity type> in the value of the Fid attribute for the relationship type. For example, the relational algebra mapping for the relationship type “Person-Age” at Site A is the following:

```
"Person-Age":      Relationship.ERA = PERSON
                   Relationship.Id = ∅
                   Relationship.Vid = ∅
                   Relationship.Fid = { Person([lname, fname],
                                        Integer([age]) }
```

Given this mapping, the value of Relationship.Fid(Person) is the field list [`lname, fname`] and the value of Relationship.Fid(Integer) is the field list [`age`].

Based on (1) the mappings specified in Section 4.2 for the relationship type Person-Age and (2) the relational algebra expressions previously computed for the term "pt," the semantic functions generate the following relational algebra translations of the term "age-of(pt)" at site A and site B:

<u>Site A</u>

```
"age-of(pt)" :    Term.ERA = Join( [lname = lname′ and fname = fname′],
                                 Sel([type = "pt"], PERSON),
                                        PERSON′)
                  Term.Id = [age′]
                  Term.Vid = { (pt, [ lname, fname]) }
                  Term.Fid = ∅
```

<u>Site B</u>

```
"age-of(pt)" :    Term.ERA = Join( [ pt-ssn = pt-ssn],
                                 Proj( [pt-ssn], PATIENT),
                                 Join([dob = date-of-birth],
                                       Union(Proj([ssn, dob],
                                                   Rename([pt-ssn′ => ssn],
                                                          PATIENT′)),
                                              Proj([ssn, dob],
                                                   Rename([md-ssn => ssn],
                                                          PHYSICIAN)),
                                       DOB-AGE))
                  Term.Id = [age]
                  Term.Vid = { (pt, [ pt-ssn  ]) }
                  Term.Fid = ∅
```

Note that when the same table appears twice in a relational algebra expression, the semantic functions implicitly rename one instance of the table (as well as any field names corresponding to that table); the renaming assures that references to separate instances of tables and fields are distinct (in the case of site A, renaming entails adding the prime symbol (′) to the table name `PERSON` and to the field names `lname`, `fname`, and `age`). Figure 4.17 shows the tables generated by the semantic functions for the term "age-of(pt)" at site A and site B.

Site A

fname	lname	age	type	fname′	lname′	age′	type′
Phil	Smythe	38	pt	Phil	Smythe	38	pt
Carol	Flynn	33	pt	Carol	Flynn	33	pt

Site B

pt-ssn	ssn	dob	date-of-birth	age
149584223	149584223	12-2-57	12-2-57	38
216426077	216426077	3-25-62	3-25-62	33

Figure 4.17: The relational algebra translations of the term "age-of(pt)"for the two relational databases shown in Figure 4.14. The semantic functions generate these tables by first substituting all references to attributes with their appropriate values and subsequently evaluating the resulting relational algebra expressions.

The relational algebra expression for any constant that appears in a ReFER query is a *constant relation*, that is, a relation that consists of one field and one row [Maier 1983 p. 26]; the value of the field is the value of the constant. The name of the field is defined arbitrarily to be "constant-field." The following semantic functions generate a constant relation to represent the relational algebra translation of any ReFER term that is a constant:

Grammar rule: Term ::= Constant

Semantic functions: Term.ERA = Constant_relation(Constant.value)
Term.Id = [`constant-field`]
Term.Vid = ∅
Term.Fid = ∅

Because constant terms contain no variables, the value of the Vid attribute is NULL. Given this definition, the semantic functions generate the following relational algebra translation of the constant "35" at both site A and site B (the expressions are the same at both sites because no site-specific mappings are referenced):

"35": Term.ERA = Constant_relation("35")
Term.Id = [`constant-field`]
Term.Vid = ∅
Term.Fid = ∅

Constant relations may appear in relational algebra expressions anywhere that conventional tables may appear. This is an important property because constants may appear in ReFER queries anywhere that terms may appear. The relational algebra translations of constants, therefore, must be consistent with the translations for any other terms, such as variables or functions. For example, the constant "35" appears as one of the terms that is compared in the predicate "age-of(pt) > 35."

Recall that the semantic functions for translating any predicate that compares two terms, which I first presented in Section 4.3.2.2, specify a join of the relational algebra translations for each term (note that the relational algebra translation for the constant "35" must be a relational table for the JOIN operation to be a valid):

Grammar rule: Predicate ::= Term_1 Op Term_2

Semantic functions: Predicate.ERA = Join([Term_1.Id Op.value Term_2.Id],
Term_1.ERA,
Term_2.ERA)
Predicate.Id = ∅
Predicate.Vid = Term_1.Vid ∪ Term_2.Vid
Predicate.Fid = ∅

Given these semantic-function definitions and the relational algebra expressions generated for the terms "age-of(pt)" and "35," the semantic functions generate the following relational algebra translations for the predicate "age-of(pt) > 35" at site A and site B:

Site A

```
"age-of(pt) > 35" :
        Term.ERA = Join( [ age′  > constant-field ],
                        Join( [lname = lname′ and fname = fname′ ],
                            Sel([type = "pt"], PERSON),
                            PERSON′ ),
                        Constant_relation("35"))
        Term.Id = ∅
        Term.Vid = { (pt, [lname, fname]) }
        Term.Fid = ∅
```

Site B

```
"age-of(pt) > 35"  :
        Term.ERA = Join( [ age  > constant-field ],
                        Join( [ pt-ssn = pt-ssn],
                            Proj( [pt-ssn], PATIENT),
                                Join([dob = date-of-birth],
                                    Union(Proj([ssn, dob],
                                              Rename([pt-ssn′ => ssn],
                                                     PATIENT′ )),
                                          Proj([ssn, dob],
                                               Rename([md-ssn => ssn],
                                                      PHYSICIAN)),
                                    DOB-AGE)),
                        Constant_relation("35"))
        Term.Id = [age]
        Term.Vid = { (pt, [pt-ssn]) }
        Term.Fid = ∅
```

Figure 4.18 shows the tables generated by the semantic functions for the predicate "age-of(pt) > 35" at site A and site B. Compare these tables to the tables in Figure 4.17 and note how the JOIN conditions specified in the semantic functions "filtered out" the bindings of the variable "pt" that did not satisfy the logic of the predicate (in Figures 4.17 and 4.18, the bindings of the variable "pt" are denoted by the values of the fields [`lname, fname`] at Site A and the value of the field [`pt-ssn`] at Site B). The resulting tables represent the relational algebra translations for the Predicate component of the ReFER query.

Site A

fname	lname	age	type	fname′	lname′	age′	type′	constant-field
Phil	Smythe	38	pt	Phil	Smythe	38	pt	35

Site B

pt-ssn	ssn	dob	date-of-birth	age	constant-field
149584223	149584223	12-2-57	12-2-57	38	35

Figure 4.18: The relational algebra translations for the predicate “age-of(pt) > 35” for the two relational databases shown in Figure 4.14.

4.3.3.3 The Termlist component

Recall from Section 4.3.1.1 that the semantic functions generate a relational algebra expression for the Termlist component that contains the values of the term list computed for all bindings of the variables that appear in the term list. The parse tree for the ReFER term list “last-name-of(pt)” is shown in Figure 4.19.

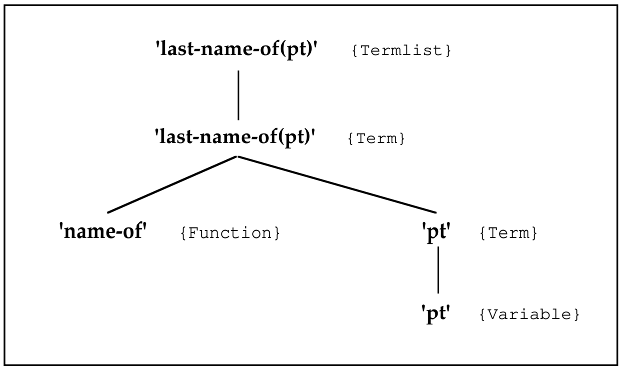


Figure 4.19: The parse tree for the Termlist component of a sample ReFER query. The context-free grammar decomposes the Termlist component into its constituent Function and Variable components (labeled in braces) . The parse tree indicates that the relational algebra translation of a term list that consists of only one term is the relational algebra translation of that term.

For term lists consisting of only one term, the relational algebra translation is equivalent to the relational algebra translation of that term. This equivalence is represented by the following semantic functions:

Grammar rule: Termlist ::= Term

Semantic functions: Termlist.ERA = Term.ERA
Termlist.Id = Term.Id
Termlist.Vid = Term.Vid
Termlist.Fid = Term.Fid

Given this equivalence, the following semantic functions are applied again to generate the appropriate relational translation of the term list "last-name-of(pt)":

Grammar rule: Term$_1$ ::= Function '(' Term$_2$ ')'

Semantic functions: Relationship = Function.relationship
Source_Type = Function.source
Target_Type = Function.target
Term$_1$.ERA = Join([Term$_2$.Id =
Relationship.Fid(Source_Type)],
Term$_2$.ERA,
Relationship.ERA)
Term$_1$.Id = Relationship.Fid(Target_Type)
Term$_1$.Vid = Term$_2$.Vid
Term$_1$.Fid = ∅

In this case, the relationship type associated with the function "last-name-of" is Person-Last_Name, the source entity type is Person, and the target entity type is String. Given these semantic-function definitions and the mappings for the relationship type Person-Last_Name at site A and site B, the semantic functions generate the following relational algebra translations for the term list "last-name-of(pt)":

Site A

"last-name-of(pt)" :
Term.ERA = Join([`lname = lname'` and `fname = fname'`],
Sel([`type` = "`pt`"], `PERSON`),
`PERSON'`),
Term.Id = [`lname'`]
Term.Vid = { (pt, [`lname, fname`]) }
Term.Fid = ∅

Site B

```
"last-name-of(pt)" :
        Term.ERA = Join( [ pt-ssn = ssn],
                         Proj( [pt-ssn], PATIENT),
                         Union(Proj([ssn, last],
                                    Rename([pt-ssn' => ssn],
                                           PATIENT')),
                               Proj([ssn, last],
                                    Rename([md-ssn => ssn],
                                           PHYSICIAN)))),
        Term.Id = [last]
        Term.Vid = { (pt, [pt-ssn]) }
        Term.Fid = ∅
```

Figure 4.20 shows the tables generated by the semantic functions for the term list "last-name-of(pt)" at site A and site B. These tables represent the relational algebra translations of the Termlist component of the query.

Site A

fname	lname	age	type	fname'	lname'	age'	type'
Phil	Smythe	38	pt	Phil	Smythe	38	pt
Carol	Flynn	33	pt	Carol	Flynn	33	pt

Site B

pt-ssn	ssn	last
149584223	149584223	Smythe
216426077	216426077	Flynn

Figure 4.20: The relational algebra translations of the term list"last-name-of(pt)" for the two relational databases shown in Figure 4.14.

4.3.3.4 The Range expression

Recall from Section 4.3.1.2 that the semantic functions of the attribute grammar join the expression generated for the Decllist component to the expression generated for the Predicate component to compute the set of bindings for which the Termlist component of a query should be evaluated. I defined the result of this JOIN operation as the “Range” expression. The following semantic functions specify the operations required to generate the Range expression:

```
Grammar rule:         Range ::= Decllist 'AND' Predicate

Semantic functions:   Range.ERA = Join( [Decllist.Vid = Predicate.Vid],
                                        Decllist.ERA,
                                        Predicate.ERA )
                      Range.Id = ∅
                      Range.Vid = Decllist.Vid ∪ Predicate.Vid
                      Range.Fid = ∅
```

The semantic functions join the relational algebra translations of the Decllist and the Predicate components on the values of their respective Vid attributes; this is because the field lists denoted by the Vid values contain the variable bindings that are represented in each relational table. By only including in the result those variable bindings that appear in the relational algebra translation for the Predicate component, the semantic functions effectively “filter out” any bindings specified in the Decllist component that do not satisfy the logic of the predicate. These are precisely the semantics of the ReFER query language for the Range component. Given these semantic-function definitions and the relational algebra expressions generated for the Decllist and Predicate components, the semantic functions defined for the Range component generate the following relational algebra translations for “(pt : Patient) AND age-of(pt) > 35” at site A and site B:

Site A

```
“(pt : Patient) AND age-of(pt) > 35”:
        Term.ERA = Join( [ lname″ = lname and fname″ = fname],
                         Sel([type″ = “pt”], PERSON″)
                         Join( [ age′  > constant-field ],
                               Join( [lname = lname′ and fname = fname′ ],
                                     Sel([type = “pt”], PERSON),
                                     PERSON′ ),
                               Constant_relation(“35”)))
        Term.Id = ∅
        Term.Vid = { (pt, [lname, fname]) }
        Term.Fid = ∅
```

Site B

```
"(pt : Patient) AND age-of(pt) > 35":
        Term.ERA = Join( [pt-ssn" = pt-ssn],
                         Proj( [pt-ssn"], PATIENT")
                         Join( [ age' > constant-field ],
                               Join( [ pt-ssn = pt-ssn' ],
                                     Proj( [pt-ssn], PATIENT),
                                     Join([dob = date-of-birth],
                                          Union(Proj([ssn, dob],
                                                      Rename([pt-ssn' => ssn],
                                                             PATIENT')),
                                                Proj([ssn, dob],
                                                      Rename([md-ssn => ssn],
                                                             PHYSICIAN)),
                                          DOB-AGE)),
                               Constant_relation("35")))
        Term.Id = ∅
        Term.Vid = { (pt, [pt-ssn]) }
        Term.Fid = ∅
```

Note the renaming of certain tables and fields with the double-prime symbol (") in these relational algebra expressions; the renaming is performed again to distinguish separate instances of identically named tables and fields. Figure 4.21 shows the tables generated by the semantic functions for the Range expression "(pt : Patient) AND age-of(pt) > 35" at site A and site B. Compare these tables to the tables in Figure 4.15 and note how the JOIN conditions specified in the semantic functions "filtered out" the bindings of the variable "pt" that did not satisfy the logic of the predicate (in Figures 4.15 and 4.21, the bindings of the variable "pt" are denoted by the values of the fields [`lname, fname`] at Site A and the value of the field [`pt-ssn`] at Site B). The resulting tables represent the relational algebra translations of the Range component of the query.

Site A

fname″	lname″	fname	lname	fname′	lname′	constant-field
Phil	Smythe	Phil	Smythe	Phil	Smythe	35

Site B

pt-ssn″	pt-ssn	ssn	constant-field
149584223	149584223	149584223	35

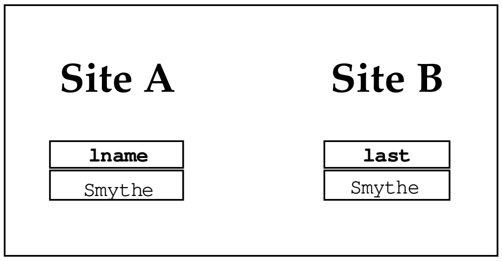
Figure 4.21: The relational algebra translations of the range expression “(pt : Patient) AND age-of(pt) > 35” for the two relational databases shown in Figure 4.14. Note that I have “condensed” the representation of certain fields in the tables to save space

4.3.3.5 The complete query

The semantic functions of the attribute grammar generate the complete relational algebra translation for a ReFER query by joining the expression generated for the Range component to the expression generated for the Termlist component, and subsequently projecting the values of the Termlist component. The following semantic functions perform this operation:

```
Grammar rule:         Query ::= '[' Termlist ' | ' Range ']'

Semantic functions:   Query.ERA = Proj( [Termlist.Id],
                                        Join([ Termlist.Vid = Range.Vid ],
                                              Termlist.ERA,
                                              Range.ERA)
                      Query.Id = Termlist.Id
                      Query.Vid = Termlist.Vid
                      Query.Fid = Termlist.Fid
```

Again, the semantic functions join the Termlist and Range expressions on their respective Vid values to “filter out” the bindings of the variable “pt” in the relational translation for the Termlist component that do not satisfy the logic of the predicate. Because the complete relational algebra expressions generated by these semantic functions are extensive, I leave their formulation as an exercise. However, Figure 4.22 shows the tables that result from evaluating the relational algebra translations produced at site A and site B. These tables represent the site-specific relational translations of the sample ReFER query.



Site A

lname
Smythe

Site B

last
Smythe

Figure 4.22: The results of evaluating the relational algebra translations of the query "last-name-of(pt) | (pt : Patient) AND age-of(pt) > 35" for the two relational databases shown in Figure 4.14.

Most relational DBMSs cannot evaluate relational algebra expressions directly, however, because commercial DBMSs only offer query interfaces based on SQL (a relational calculus language). At each site, therefore, TransFER must convert the relational algebra translations produced by the attribute grammar to equivalent SQL queries that can be executed by the local DBMS.

4.3.4 Translating relational algebra to SQL

The translation of ERA expressions to SQL queries is a straightforward process, owing to the semantic equivalence of a relational algebra augmented with aggregation and the relational calculus [Klug 1982]. Several algorithms for performing this translation exist [Klug 1982, Ullman 1988]. The only complication is that certain subexpressions of a relational algebra query must be formulated as *separate SQL queries*, rather than as SQL subqueries. Specifically, relational algebra subexpressions that specify the UNION operation or the AGGREGATE operation must be formulated as distinct SQL queries. The reason is that the current SQL standard does not allow SQL union or aggregation operations to appear within the relational expressions in the "FROM" list of an SQL query. For example, the following SQL query is not valid:

```
SELECT  last
FROM  (SELECT pt-ssn, last FROM PATIENT)
        UNION
      (SELECT md-ssn, last FROM PHYSICIAN).
```

Therefore, the results of SQL union and aggregation operations must be saved in temporary intermediate tables; the names of these tables appear in the FROM lists of subsequent SQL queries. The results of relational algebra subexpressions that specify *external functions* also must be computed separately and stored in temporary tables. The reason is that external functions, by definition, cannot be executed by an SQL query engine. The SQL query engine can retrieve only the arguments to external functions; the functions must subsequently operate on these arguments in the host environment and return their results to temporary tables in the database. These tables then appear in subsequent SQL queries. Therefore, a single relational algebra expression that results from the attribute-grammar translation of a ReFER query may be translated into a sequence of multiple SQL queries that communicate via intermediate results saved in temporary tables. The final result of a ReFER query translation is a *database program* consisting of (potentially) multiple SQL queries interleaved with run-time commands to create temporary tables and to call external functions. Figure 4.24 shows the site-specific output of a TransFER query translation.

```
Begin database program {

INSERT INTO Temp_Table (Field1, Field2)
  ( SELECT pt-ssn, dob
    FROM PATIENT )
  UNION
  ( SELECT md-ssn, dob
    FROM PHYSICIAN)


SELECT pt.lname
FROM Temp_Table tt, PATIENT pt
WHERE (datediff(year,tt.Field2,getdate()) > 35 AND
      (pt.pt-ssn = tt.Field1)

} End database program
```

operations may, in fact, be *redundant* in the sense that if the operations were omitted, the meanings of the relational algebra expressions would be unchanged. Redundant relational algebra operations are introduced into query translations by the way in which the TransFER attribute grammar combines the site-specific mappings for FER entity types and relationship types when translating ReFER queries. Although redundant operations do not affect the correctness of the relational algebra expressions generated by the attribute grammar, they do, in practice, increase the time and computing resources that are required to evaluate the expressions. For this reason, it is desirable to detect and to remove redundant relational operations from expressions produced by the attribute grammar. I now describe the theory and the mechanism for performing certain of these optimizations.

The TransFER optimizer detects and eliminates redundant JOIN operations by applying a theory of "PROJECT equivalence" that is based on the relational algebra. In this section, I describe the theory, I prove its correctness, and I illustrate its application to the problem of optimizing relational algebra expressions produced by the TransFER attribute grammar.

4.4.1 The theory of PROJECT equivalence

The TransFER compiler applies optimizations that are based on a specific kind of equivalence between relational algebra expressions called "PROJECT equivalence." Informally, PROJECT equivalence between two relational algebra expressions (that is, relations) means that the two relations are identical except for the existence of certain *duplicated attributes*. A duplicated attribute is an attribute of a relation, the value of which is equal to the value of another attribute for every row of the relation. For example, two "PROJECT-equivalent" relations appear in Figure 4.23. PROJECT equivalence between two relations implies that the same PROJECT operation performed over both relations produces identical results.

The expression "$R \hat{=} S$" means that the relations R and S are PROJECT equivalent. I define PROJECT equivalence formally as follows:

> **Definition**: If $R \hat{=} S$, then there exists some transformation Δ (defined below) such that the application of Δ to R generates S, that is, $\Delta(R) = S$.

> The transformation Δ consists of some finite sequence of transformation functions δ (defined below) such that $\Delta = [\delta_1,\delta_2,..,\delta_n]$ where $n \geq 0$. Each transformation function δ must consist of one of the following operations:
>
> 1. *Duplicate*. For some attribute α of the relation R, a new attribute α' is added to the relation R such that α' has the same domain as α and $\alpha'=\alpha$ in every row of R. The Duplicate operation essentially reproduces one column of the relation R and assigns a unique attribute name to the new column. Because the new column contains the same values as an existing column, the Duplicate operation adds no information.
>
> 2. *Merge*. For some pair of attributes α, α' of the relation R where $\alpha = \alpha'$ in every row of R, the attribute α' is eliminated from the relation R. The merge function essentially eliminates a redundant column in the relation R, generating a relation with only one "copy" of the column. Because the eliminated column contained the same values as an existing column, the Merge operation removes no information.
>
> The sequential application of the transformation functions $[\delta_1,..,\delta_n]$ to the relation R performs the transformation $\Delta(R)$, such that if $R \triangleq S$ and $\Delta = [\delta_1,\delta_2,..,\delta_n]$, then $\Delta(R) = \delta_n(..\delta_2(\delta_1(R))..) = S$. Note that Duplicate and Merge are inverse operations. This property implies that if a transformation Δ exists such that $\Delta(R) = S$, then some transformation Δ' exists such that $\Delta'(S) = R$. Also note that two relations are PROJECT equivalent if and only if they have the same cardinality, because neither the Duplicate nor Merge operations can change the cardinality of a relation.

Figure 4.23 shows two relations that are PROJECT equivalent. The relation R can be transformed to the relation S by first duplicating the attribute `fname` in R and subsequently merging the attributes `age` and `age'`. The inverse sequence of Duplicate and Merge operations perform the transformation from S to R.

Relation R

fname	lname	age	age′	type
Phil	Smythe	38	38	pt
Carol	Flynn	33	33	pt
Colin	Krup	56	56	md

$\triangleq$

Relation S

fname	fname′	lname	age	type
Phil	Phil	Smythe	38	pt
Carol	Carol	Flynn	33	pt
Colin	Colin	Krup	56	md

Figure 4.23: Two relations that are PROJECT equivalent. A sequence of one Merge and one Duplicate operation can transform either relation into the other.

PROJECT equivalence is relevant to query optimization because the same projection of two PROJECT-equivalent relational algebra expressions produces the same result. This implies that if the elimination of a JOIN from a relational algebra expression generates a PROJECT-equivalent expression, then any subsequent projection of that expression will produce the same results as if the JOIN had not been eliminated. This property allows the optimizer to remove certain JOINs without changing the meaning of the query. Formally, I define this property of PROJECT equivalence as Theorem 4.1. Unless otherwise stated, I will use "conventional" relational algebra notation in the remainder of this chapter. The uppercase letters R, S, and T will denote relations; uppercase letters near the beginning and end of the alphabet will denote sets of relational attributes (for example, A, B, X, Y); the symbol α will denote a single relational attribute.

Theorem 4.1 if $R \triangleq S$,

$$\text{then } \pi_X R = \pi_{\rho_{R\to S}(X)} S$$

The function $\rho_{R\to S}(X)$ is a *renaming* operation that renames certain attributes in the attribute list X. The renaming takes place so that the names of the projected attributes are consistent with the transformation between R and S that is defined by $R \triangleq S$. Specifically, the renaming operation $\rho_{R\to S}$ defines a *sequence* of renaming functions $[\rho_1, \rho_2, .., \rho_n]$ such that each ρ_i corresponds to one of the transformation functions $\delta_i \in \Delta$ where $\Delta(R) = S$. Each ρ_i consists of an ordered pair of attribute names $<\alpha \to \alpha'>$ such that ρ_i renames every instance of the attribute name α in an attribute list to the attribute name α'. The pair of attribute names for each ρ_i that corresponds to a δ_i is determined as follows:

1. If δ_i is a Merge operation that merges the attributes α and α' such that only the attribute α remains, then the pair of attribute names for ρ_i is $<\alpha' \rightarrow \alpha>$.
2. If δ_i is a Duplicate operation that duplicates the values of the attribute α, then the pair of attribute names for ρ_i is $<\alpha \rightarrow \alpha>$ (that is, no renaming is specified).

Informally, the renaming operation ensures that any attribute α' in the relation R that is eliminated by a Merge operation when R is transformed to S is appropriately replaced by the attribute α in S if α' appears in the list of attributes to be projected. Because the Duplicate operation does not eliminate any attributes, no renaming is required. For example, the following renaming operation is defined by the transformation from the relation R to the relation S in Figure 4.23:

$$\rho_{R \rightarrow S} = [<\texttt{fname} \rightarrow \texttt{fname}>, <\texttt{age'} \rightarrow \texttt{age}>].$$

When applied to the attribute list [fname, lname, age′], the renaming operation produces the attribute list [fname, lname, age]. Inspection of the relations shown in Figure 4.23 demonstrates that this operation is required for PROJECT equivalence to hold in the following example:

$$\pi_{[\text{fname,lname,age}']}R = \pi_{[\text{fname,lname,age}]}S.$$

Theorem 4.1 is proved by induction on the sequential application of the transformation functions $\delta_1,..,\delta_n$ that constitute the transformation Δ between R and S. I claim (without proof) that Theorem 4.1 holds equally whether I use the PROJECT or the MULTIPROJECT operation in the following proof (I therefore use the symbol π to denote either operation):

> *Proof*: The inductive hypothesis is that Theorem 4.1 holds before any transformations are performed, that is, when R = S. In this case, $\Delta = \varnothing$, which implies that $\rho_{R \rightarrow S} = \varnothing$ by the definition of the renaming operation. If $\rho_{R \rightarrow S} = \varnothing$, then $\rho_{R \rightarrow S}(X) = X$. This equality proves the inductive hypothesis because we know that $\pi_X R = \pi_X S$ when R = S. The inductive step is that Theorem 4.1 holds when a single transformation

function, δ, is applied to the relation R to produce the relation S. There are two cases:

Case 1. δ is a Duplicate operation that adds the attribute α′ to R, which contains the same values as the existing attribute α for every row. Because the Duplicate operation creates a `NULL` renaming operation (that is, $\rho_{R \to S} = [<A \to A>]$), we know that $\rho_{R \to S}(X) = X$ for any attribute list X. Therefore, the theorem holds if the equality $\pi_X R = \pi_X S$ holds given the Duplicate operation that transforms R to S. Because the relation S is simply the relation R with one additional attribute (α′), I can write $R = \pi_Y S$ where Y is the original set of attributes in the relation R. Substituting, the theorem holds if the equality $\pi_X(\pi_Y S) = \pi_X S$ holds. Because Y is the original set of attributes of R, we know that $X \subseteq Y$, because otherwise the expression $\pi_X R$ would be invalid. This containment implies, by an equivalence of the relational algebra [Ullman 1989], (p. 664) that $\pi_X(\pi_Y S) = \pi_X S$.

Case 2. δ is a Merge operation that eliminates the duplicate attribute α' from R, where α′ contains the same values for every row as some remaining attribute α. In this case, $\rho_{R \to S} = \{<\alpha' \to \alpha>\}$, so that each instance of α′ in the attribute list X is replaced by α. Two subcases exist:

> **2a**. X contains no instances of α′. In this case, $\rho_{R \to S}(X) = X$ and the theorem holds if the equality $\pi_X R = \pi_X S$ holds for the Merge operation that transforms R to S. Because the relation S is simply the relation R with one attribute removed (α′), I can write $S = \pi_Z R$, where Z is the original set of attributes in R minus the removed attribute α′. Substituting, the theorem holds if the equality $\pi_X R = \pi_X(\pi_Z R)$ holds. Because we have assumed in this case that X does not contain the removed attribute, α′, we know that $X \subseteq Z$. Therefore, by the same equivalence, $\pi_X R = \pi_X(\pi_Z R)$.
>
> **2b**. X contains one or more instances of α′. In this case, each such instance is renamed to α. By the definition of the

Merge operation, we know that the value of α' in each row of R is equal to the value of α in each row of S. We also know that the values of the attributes in the attribute list $X-\alpha'$ (that is, the attributes that are common to both relations) are equal in each row of R and S because the Merge operation did not affect these attributes. Therefore, projecting the combination of the values of α' and $X-\alpha'$ in the relation R produces the same result as projecting the value of α and $X-\alpha'$ from the relation S. In other words, $\pi_X R = \pi_{\rho_{R\to S}(X)} S$.□

4.4.2 An application of PROJECT equivalence

In Section 4.3, I showed how the semantic functions of the TransFER attribute grammar may introduce redundant operations when translating ReFER queries to relational algebra expressions. For example, the semantic functions generate the following relational translation for the term "last-name-of(pt)" at site B:

```
Term.ERA = Join( [ pt-ssn = ssn],                    (expression 1)
                Proj( [pt-ssn], PATIENT),
                Union(Proj([ssn, last],
                           Rename([pt-ssn' => ssn],
                                  PATIENT')),
                      Proj([ssn, last],
                           Rename([md-ssn => ssn],
                                  PHYSICIAN)))).
```

Inspection of the relational schema for the database at site B (shown in Figure 3.15) reveals that the last name of each patient can be computed by the much simpler relational algebra expression:

```
Term.ERA = Proj( [ pt-ssn, last], PATIENT).          (expression 2)
```

The semantic functions of the attribute grammar produce the more complex relational algebra translation for "last-name-of(pt)" because the site-specific and site-independent semantic functions of the attribute grammar must be specified in the most general way possible. For example, the site-specific mapping for the relationship type Person-Last_Name must specify a union of the `PATIENT` and the `PHYSICIAN` tables, because the relationship type may be used to derive the last names of patients, physicians, or both (recall that Person-Last_Name is defined for the supertype Person in the FER schema). When the relationship type is used to derive the last name of a patient, as in "last-name-of(pt)," no data from the `PHYSICIAN` table is needed and the union is unnecessary. At the time the mapping is specified, however, there is no way to anticipate how it will be applied in any particular query, so it must be specified to accommodate the most general case. Likewise, the semantic function that translates the application of a relationship function (such as "last-name-of") to a term (such as "pt") must specify a JOIN between the relational algebra translation of the relationship function and the relational algebra translation of the term. The JOIN is necessary because the semantic function is *site-independent* and, in the most general case, a JOIN is required to derive the associations between entities. When the relationship function "last-name-of" is used to derive the last name of a patient at site B, however, the JOIN is unnecessary because the last name is represented in the same table as the patient. The role of the TransFER optimizer is to recognize these unnecessary operations in specific relational algebra expressions and to transform the expressions into equivalent expressions without the operations. The optimizer exploits the properties of PROJECT equivalence to perform certain of these transformations.

For example, the following relational algebra equivalence allows the optimizer to transform the original expression produced by the attribute grammar for the term "last-name-of(pt)" (expression 1) to the simpler expression that I produced manually (expression 2) without changing the meaning of the expression (again, I use the "conventional" relational algebra notation):

Theorem 4.2 $(\pi_X R \cup \pi_Y S) \underset{A = B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R$

if the list of attributes A = the list of attributes B, and the values of the attributes in A are unique in the relation $(\pi_X R \cup \pi_Y S)$.

The optimizer eliminates one JOIN in transforming the original expression, which has the form of the left-hand side of the equivalence in Theorem 4.2, to the simpler expression, which has the form of the right-hand side. The optimizer can apply this transformation because the last relational algebra operation specified by the semantic functions of the attribute grammar is a multiprojection (see Section 4.3.1.2). If the attributes that are projected by this operation are appropriately renamed, then the properties of PROJECT equivalence ensure that the results of the query will be the same after the transformation is performed. However, for Theorem 4.2 to hold, the PROJECT operations denoted on the left-hand and right-hand sides of the PROJECT equivalence must be "classic" PROJECT operations, that is, PROJECT operations that eliminate duplicate rows. The reason for this constraint is apparent in the proof of Theorem 4.2 that appears in the next section.

4.4.3 Proving correctness of the optimizations

The proof of Theorem 4.2 proceeds in two steps. The first step proves the relational equivalence:

Lemma 4.1 $(\pi_X R \cup \pi_Y S) \underset{A=B}{\bowtie} \pi_Z R = \pi_X R \underset{A=B}{\bowtie} \pi_Z R$

if the list of attributes A = the list of attributes B,
and the values of the attributes in A are unique in
the relation $(\pi_X R \cup \pi_Y S)$

Note that the symbol "=" appears in Lemma 4.1, rather than "$\triangleq$." The second step proves a more general PROJECT equivalence, which actually is the basis for a large number of optimizations:

Theorem 4.3 $\pi_X R \underset{A=B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R$

if the list of attributes A = the list of attributes B,
and the values of the attributes in A are unique in
either the relation $\pi_X R$ or $\pi_Z R$

Note that Theorem 4.3 may be applied to the proof of Theorem 4.2 because the condition that the values of the attributes in A are unique in the relational expression ($\pi_X R \cup \pi_Y S$) *implies* that the values of the attributes in A are unique in the relational expression $\pi_X R$. The combination of Lemma 4.1 and Theorem 4.3 proves Theorem 4.2 because of the following properties of PROJECT equivalence:

1. Relational equivalence implies PROJECT equivalence, that is, if $R = S$, then $R \triangleq S$. The definition of PROJECT equivalence specifies that one possible transformation, Δ, from R to S such that $R \triangleq S$ is the NULL transformation, that is, $\Delta = \varnothing$. Thus, if $R = S$, then $\Delta(R) = S$ where $\Delta = \varnothing$ and $R \triangleq S$. This property implies the following:

   $$\text{if} \qquad (\pi_X R \cup \pi_Y S) \underset{A=B}{\bowtie} \pi_Z R = \pi_X R \underset{A=B}{\bowtie} \pi_Z R$$

   $$\text{then} \qquad (\pi_X R \cup \pi_Y S) \underset{A=B}{\bowtie} \pi_Z R \triangleq \pi_X R \underset{A=B}{\bowtie} \pi_Z R\,.$$

2. PROJECT equivalence is transitive, that is, if $R \triangleq S$ and $S \triangleq T$, then $R \triangleq T$. Transitivity holds because one can simply *compose* the sequence of transformation functions that transform R to S and S to T to transform R to T, that is, if $\Delta_1(R) = S$ where $\Delta_1 = \{\delta_1,..,\delta_n\}$ and $\Delta_2(S) = T$ where $\Delta_2 = \{\delta_{n+1},..,\delta_m\}$, then $\Delta_3(R) = T$ where $\Delta_3 = \Delta_1 + \Delta_2 = \{\delta_1,..,\delta_n,\delta_{n+1},..,\delta_m\}$. The transitivity of PROJECT equivalence implies the following:

   $$\text{if} \qquad (\pi_X R \cup \pi_Y S) \underset{A=B}{\bowtie} \pi_Z R \triangleq \pi_X R \underset{A=B}{\bowtie} \pi_Z R \qquad \text{and}$$

   $$\pi_X R \underset{A=B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R$$

   $$\text{then} \qquad (\pi_X R \cup \pi_Y S) \underset{A=B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R.$$

These two properties confirm that the proof of Theorem 4.2 rests on the proofs of Lemma 4.1 and Theorem 4.3.

Proof of Lemma 4.1: By the distributive property of the JOIN operator in relational algebra,

$$(\pi_X R \cup \pi_Y S) \underset{A=B}{\bowtie} \pi_Z R = (\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R).$$

The proof of Lemma 4.1, therefore, depends on establishing the following equivalence:

$$(\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = (\pi_X R \underset{A=B}{\bowtie} \pi_Z R)$$

I prove this equivalence by considering three possibilities for the value of the attribute A: The value appears only in the relation $\pi_X R$, the value appears only in the relation $\pi_Y S$, and the value appears in both the relations $\pi_X R$ and $\pi_Y S$.

Case 1. The value of the attribute A appears only in $\pi_X R$ and not in $\pi_Y S$. For all such values,

$$(\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = \varnothing$$

because the JOIN condition cannot be satisfied by any values of A in $\pi_Y S$. This implies

$$\begin{aligned}(\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) &= (\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup \varnothing \\ &= (\pi_X R \underset{A=B}{\bowtie} \pi_Z R)\end{aligned}$$

Case 2. The value of the attribute A appears only in $\pi_Y S$ and not in $\pi_X R$. For all such values,

$$(\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = \varnothing.$$

This is true because both the set of attributes X and the set of attributes Z must contain the set of attributes A; if they do not, the original expression

$$(\pi_X R \cup \pi_Y S) \underset{A = B}{\bowtie} \pi_Z R$$

is not a valid relational algebra expression (recall that the set of attributes A = the set of attributes B). Given that both X and Z contain A, if the values of the attributes A do not appear in $\pi_X R$, then they cannot appear in $\pi_Z R$ (because both expressions include the projection of the attributes A). Therefore, for all values of the attributes A that do not appear in $\pi_X R$,

$$(\pi_Y S \underset{A = B}{\bowtie} \pi_Z R) = \varnothing$$

because the JOIN condition cannot be satisfied by any values of A in $\pi_Z R$. This implies

$$\begin{aligned} (\pi_X R \underset{A = B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A = B}{\bowtie} \pi_Z R) &= (\pi_X R \underset{A = B}{\bowtie} \pi_Z R) \cup \varnothing \\ &= (\pi_X R \underset{A = B}{\bowtie} \pi_Z R). \end{aligned}$$

Case 3. The value of the attribute A appears in *both* $\pi_X R$ and in $\pi_Y S$. Because we know that the values of A are unique in ($\pi_X R \cup \pi_Y S$), this implies that the values of each attribute in the attribute set X-A and in the attribute set Y-A *are the same* for each value in the columns of A that appears in both $\pi_X R$ and $\pi_Y S$, respectively; the values of X-A and of Y-A must be the same so that the relational UNION operator will “merge” the rows from $\pi_X R$ and $\pi_Y S$ that have the same values of A into a single row when the expression ($\pi_X R \cup \pi_Y S$) is evaluated. Therefore, for each pair of rows in $\pi_X R$ and $\pi_Y S$ that have the same values of the attributes A, the values of the other attributes are also the same. In other words, the rows are identical, that is $\pi_X R = \pi_Y S$. For all such rows,

$$(\pi_X R \underset{A = B}{\bowtie} \pi_Z R) = (\pi_Y S \underset{A = B}{\bowtie} \pi_Z R) = T.$$

Substituting,

$$(\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = T \cup T = T$$

$$= (\pi_X R \underset{A=B}{\bowtie} \pi_Z R).$$

Because I have shown that the equivalence

$$(\pi_X R \underset{A=B}{\bowtie} \pi_Z R) \cup (\pi_Y S \underset{A=B}{\bowtie} \pi_Z R) = (\pi_X R \underset{A=B}{\bowtie} \pi_Z R)$$

holds for all of the possible values of the attributes in A, I have proven Lemma 4.1.□

Given Lemma 4.1, Theorem 4.2 is proved by proving Theorem 4.3:

$$\pi_X R \underset{A=B}{\bowtie} \pi_Z R \triangleq \pi_{(X \cup Z)} R$$

> if the list of attributes A = the list of attributes B,
> and the values of the attributes in A are unique in
> *either* the relational expression $\pi_X R$ or $\pi_Z R$.

Proof: Assume that the values of the attributes in A are unique in $\pi_X R$ (a symmetrical proof exists for the case in which the values are unique in $\pi_Z R$). There are three possible *containment relationships* between the sets of attributes X and Z: $X = Z$ (implying that $X \cup Z = X$), $X \supset Z$ (implying that $X \cup Z = X$), and $X \subset Z$ (implying that $X \cup Z = Z$). I prove Theorem 4.3 by proving that the following assertions hold for each containment relationship:

1. The expression on the left-hand side of the PROJECT equivalence *has the same cardinality* as the expression on the right-hand side.

2. There exists a *sequence of Merge operations* that transforms the expression on the left-hand side of the PROJECT equivalence to the expression on the right-hand side.

By the definition of PROJECT equivalence, these two assertions prove Theorem 4.3. In the proof below, I adopt the following notational convention: When the same relation appears twice in an expression, I rename one "copy" of the relation and each attribute of that relation with the prime (′) symbol. For example, this renaming changes the expression

$$\pi_X R \underset{A = A}{\bowtie} \pi_Y R$$

(where A is a set of attributes in the relation R) to the expression

$$\pi_X R \underset{A = A'}{\bowtie} \pi_Y R'.$$

If A = the set of attributes $\{\alpha_1,..,\alpha_n\}$ in the relation R, then A′ = $\{\alpha_1' ,..,\alpha_n' \}$ in the relation R′. This convention helps to distinguish copies of attributes with the same name that result from joining a relation to itself.

Case 1. X = Z. In this case, I must show

$$\pi_X R \underset{A = A'}{\bowtie} \pi_{X'} R' = T \triangleq \pi_X R.$$

Because A is unique in $\pi_X R$, A′ is unique in $\pi_{X'} R'$. This implies a one-to-one relationship between the rows of $\pi_X R$ and $\pi_{X'} R'$ for each value of the attributes in A; therefore, $|T| = |\pi_X R|$, by the definition of the JOIN operation. For each attribute $\alpha' \in X'$, either $\alpha' \in A'$ or $\alpha' \in X'-A'$ (note that A′ must be a subset of X′ for the JOIN operation to be valid). For each $\alpha' \in A'$, there is an $\alpha \in A$ such that (1) both α′ and α are in the relation T (because $A' \subset X'$ and $A \subset X$) and (2) in each row of T, the value of α′ = the value of α (because that equality is implied by the JOIN condition). Therefore, for each $\alpha' \in A'$, the attributes α′ and α in T may be merged into the single attribute α, by the definition of the Merge operation. For each $\alpha' \in X' - A'$, there must be a functional dependency $A' \to \alpha'$ in R′, such that there is only one possible value of α′ for each value of A′; this dependency must exist for the values of A′ to be unique in $\pi_{X'} R'$ because otherwise there could be two values of α′ for each value of A′ and A′ would not be unique in $\pi_{X'} R'$. By the same argument, there is an identical functional dependency $A \to \alpha$ in R because the values of A are unique in $\pi_X R$. Because the value of A′ = the value of A in each row of T (by the JOIN condition), the functional dependencies imply that the value of α' = the value of α in each row of T. Therefore, for each $\alpha' \in X' - A'$, the attributes α' and α in T may be merged into the single attribute α. Because every $\alpha' \in X'$ may be merged with an α in the relation T (that is, in the result of the JOIN), there exists a transformation from T to $\pi_X R$ that is consistent with the definition of PROJECT equivalence; therefore, T and $\pi_X R$ are PROJECT equivalent.

Case 2. $X \supset Z$. In this case, I must show

$$\pi_X R \underset{A = A'}{\bowtie} \pi_Z R' = T \triangleq \pi_X R.$$

Because the values of A are unique in $\pi_X R$, the values of A′ must be unique in $\pi_Z R'$ because $X \supset Z$ (a more restrictive projection certainly cannot create any duplicate values). Therefore, there is a one-to-one relationship between the rows of $\pi_X R$ and $\pi_Z R'$ for each value of the attributes in A, implying again that $|T| = |\pi_X R|$. Note that for each attribute $\alpha' \in Z$, there must be a corresponding $\alpha \in X$ because $X \supset Z$. For each $\alpha' \in A'$, there is an $\alpha \in A$ such that (1) both α' and α are in the relation T and (2) in each row of T, the value of α' = the value of α (because that equality is implied by the JOIN condition). Therefore, for each $\alpha' \in A'$, the attributes α' and α in T may be merged into the single attribute α. For each $\alpha' \in Z - A'$, there must be a functional dependency $A' \rightarrow \alpha'$ in R′ (by the same argument as in Case 1) because A′ is unique in $\pi_Z R'$. Given $A' \rightarrow \alpha'$ in R′, there is an identical functional dependency $A \rightarrow \alpha$ in R (recall that R and R′ are the same relations). Because the value of A′ = the value of A in each row of T (by the JOIN condition), the functional dependencies imply that the value of α' = the value of α in each row of T. Therefore, for each $\alpha' \in Z - A'$, the attributes α' and α in T may be merged into the single attribute α. Because every $\alpha' \in Z$ may be merged with an α in the relation T (that is, in the result of the JOIN), there exists a transformation from T to $\pi_X R$ that is consistent with the definition of PROJECT equivalence, again proving that T and $\pi_X R$ are PROJECT equivalent.

Case 3. $X \subset Z$. In this case, I must show

$$\pi_X R \underset{A = A'}{\bowtie} \pi_Z R' = T \triangleq \pi_Z R'.$$

Because A is unique in $\pi_X R$, there is exactly one row in $\pi_X R$ that corresponds to each row in $\pi_Z R'$ such that the value of A = the value of A' (that is, such that the JOIN condition holds). This implies a many-to-one relationship between the rows of $\pi_Z R'$ and $\pi_X R$ for each value of A′ (note that the values of A′ are not necessarily unique in $\pi_Z R'$). Therefore, $|T| = |\pi_Z R'|$, by the definition of the JOIN operation. Note that for each attribute $\alpha \in X$, there must be a corresponding $\alpha' \in Z$ because $X \subset Z$. For each $\alpha \in A$, there is an $\alpha' \in A'$ such that (1) both α and α' are in the relation T and (2) in each row of T, the value of α = the value of α' (because that equality is implied by the JOIN condition). Therefore, for each $\alpha \in A$, the attributes α and α' in T may be merged into the single attribute α'. For each $\alpha \in X\text{-}A$, there must be a functional dependency $A \rightarrow \alpha$ in R, because A is unique in $\pi_X R$ (see Case 1). Given $A \rightarrow \alpha$ in R, there is an identical functional dependency $A' \rightarrow \alpha'$ in R′ (recall that R and R′ are the same relations). Because the value of A = the value of A′ in each row of T (by the JOIN condition), the functional dependencies imply that the value of α = the value of α' in each row of T. Therefore, for each $\alpha \in X\text{-}A$, the attributes α and α' in T may be merged into the single attribute α'. Because every $\alpha \in X$ may be merged with an α' in the relation T (that is, in the result of the JOIN), there exists a transformation from T to $\pi_Z R'$ that is consistent with the definition of PROJECT equivalence; therefore, T and $\pi_Z R$ are PROJECT equivalent. □

The proof of Theorem 4.3 completes the proof of Theorem 4.1. However, the TransFER optimizer applies Theorem 4.3 to justify a large set of simplifying transformations. For example, Theorem 4.3 implies each of the following PROJECT equivalences:

Corollary 4.3a $R \underset{A = A}{\bowtie} R \triangleq R$

Corollary 4.3b $\pi_X R \underset{A = A}{\bowtie} R \triangleq R$

Corollary 4.3c $\pi_X R \underset{A = A}{\bowtie} \pi_Y R \triangleq \pi_X R \quad \text{if } X \supseteq Y$

Lastly, one easily can generalize Theorem 4.3 to Theorem 4.4:

Theorem 4.4 $(\pi_X R \underset{C = D}{\bowtie} S) \underset{A = B}{\bowtie} (\pi_Z R \underset{E = F}{\bowtie} T) \triangleq ((\pi_{(X \cup Z)} R \underset{C = D}{\bowtie} S) \underset{E = F}{\bowtie} T)$

if the list of attributes A = the list of attributes B,
the attributes A appear only in the relation $\pi_X R$, the
attributes B appear only in the relation $\pi_Z R$, and
the values of the attributes A are unique in *either*
the relation $\pi_X R$ or $\pi_Z R$

The proof of Theorem 4.4 follows directly from the commutativity and associativity properties of the JOIN operator [Ullman 1988, p. 62] and is sufficiently straightforward that I have omitted it.

4.4.4 The optimization theorems in practice

Theorem 4.3 and its corollaries are useful for eliminating many JOIN operations that appear in the relational translations of ReFER queries. The wide applicability of Theorem 4.3 is based on the way in which the site-specific semantic functions of the attribute grammar specify entity-type and relationship-type mappings and the way in which the site-independent semantic functions of the attribute grammar translate relationship functions and relationship predicates.

Recall that the site-specific mapping for each entity type specifies (1) a relational algebra expression in which each row represents a distinct instance of the entity type and (2) a set of key fields that represent the identifier of each instance of the entity type (see Section 4.2.1). The semantic functions that specify these mappings assign the relational algebra expression to the attribute ERA and the set of key fields to the attribute Id. Because each row in the relational algebra expression represents a distinct entity and the values of the key fields represent the identifier of each entity, the values of the key fields are unique in the relational algebra expression. For example, the following semantic functions specify the mapping for the entity type Person at Site A (see Figure 4.11 for the complete schema at Site A):

Person : Entity_Type.ERA = "`PERSON`"
Entity_Type.Id = [`lname, fname`]

This mapping implies that the concatenated values of the fields [`lname, fname`] are unique in the table `PERSON` at site B. Therefore, by Theorem 4.3, if the mapping for the entity type Person is joined to another instance of the table `PERSON` on the fields [`lname, fname`], that JOIN may be eliminated without changing the meaning of the relational algebra query. Note that such a JOIN will occur in the translation of "last-name-of(p)" (where "p" is declared to range over the entity type Person) at site A because the site-specific mapping for the relationship type Person-Last_Name also assigns the table `PERSON` to the ERA attribute:

Person-Last_Name: Relationship_Type.ERA = "`PERSON`"
Relationship_Type.Fid = { Person([`lname, fname`]),
String([`lname`]) }

The semantic functions for the mappings of both Person and Person-Last_Name assign the table `PERSON` to the ERA attribute because at Site A, the representation of each person's last name is stored as a *field* in the same table that stores the set of Person entities (see Figure 4.11). Given the definition of the site-independent semantic function that translates terms that apply relationship functions (such as "last-name-of"), the relational translation of the term "last-name-of(p)" at Site A is

Term.ERA = Join([`lname = lname'` and `fname = fname'`],
`PERSON`,
`PERSON'`),
Term.Id = [`lname'`]
Term.Vid = { (pt, [`lname, fname`]) }
Term.Fid = ∅.

Note that the value of the ERA attribute takes the form

$$R \underset{A = A'}{\bowtie} R'$$

where R and R′ both represent the table `PERSON` and A and A′ both represent the attribute list [`lname, fname`]. By Corollary 4.3a, the relational translation of the term “last-name-of(p)” at Site A may be simplified to

Term.ERA = `PERSON`
Term.Id = [`lname`]
Term.Vid = { (pt, [`lname, fname`]) }
Term.Fid = ∅

provided that all subsequent references to `lname′` and `fname′` are replaced by references to `lname` and `fname`. The corollary of Theorem 4.3 may be applied because the values of the attribute list [`lname, fname`] are unique in the table `PERSON`, as Theorem 4.3 requires.

4.5 Summary

The TransFER query-translation methodology provides a principled approach to translating abstract clinical queries to equivalent and site-specific relational database programs. Based on the theories of relational algebra and attribute-grammar translation, the translation methodology supports (1) the formal specification of database mappings to bridge representational heterogeneity, (2) the systematic composition of site-specific mappings to translate arbitrary queries, and (3) the principled optimization of relational algebra translations to improve computational efficiency. However, the ultimate utility of the TransFER methodology can be judged only in the context of the problem that TransFER addresses: providing a uniform query interface to disparately implemented clinical databases. In the next chapter, I present the criteria that I used to evaluate TransFER's utility and I describe the design and the results of a formal evaluation that I conducted to test TransFER's utility.

Chapter 5

The Implementation and Evaluation of TransFER

All computer applications are tools intended to perform a function or to help users perform a function. As such, a new computer technology should be judged by its ability to (1) perform a useful function that could not be performed previously, (2) perform a function more effectively than it could be performed previously, or (3) perform a function more efficiently than it could be performed previously. By some criterion, the new technology should constitute an *improvement* over the status quo. TransFER is a computer tool intended to facilitate the sharing of decision-support applications that require clinical data. Although currently available methods of sharing decision-support applications leave much to be desired, as discussed in Chapter 2, the TransFER methodology must demonstrate improved performance before one can advocate its use. This chapter describes the general criteria by which the methodology of TransFER was judged and the specific means by which an implementation of TransFER was evaluated.

5.1 A Controlled Experiment

I conducted an experiment to evaluate the utility and generality of the TransFER query-translation methodology. The hypotheses of the experiment were (1) that the TransFER methodology could map a reference schema of clinical data to three independently designed relational database schemas, all of which represent the same data set, and (2) that the TransFER methodology automatically could translate queries formulated against the single reference schema to queries that could be executed against each of the relational schemas. Specifically, the experiment entailed the following steps:

1. Three groups, each consisting of one clinician and one database expert, designed a relational database schema based on the same clinical data set.

2. I designed a global conceptual schema in the FER conceptual modeling language to represent the same clinical data set.

3. A small group of physicians informally formulated a set of queries to retrieve clinically relevant information represented in the data set. I encoded these queries in the ReFER query language in a manner consistent with the conceptual FER schema.

4. I mapped the conceptual FER schema to each relational schema using the extended relational algebra mapping language.

5. The TransFER compiler translated the ReFER queries to three sets of SQL queries such that each set was consistent with one of the relational database schemas.

6. I manually translated the ReFER queries to three sets of "gold-standard" SQL queries, which I compared to the TransFER-generated queries with respect to correctness and efficiency.

The experiment allowed me to evaluate the TransFER methodology in the context of a realistic clinical data set and with respect to heterogeneously implemented databases. The experiment addressed several specific questions:

1. Is the ReFER query language sufficiently powerful to capture the semantics of the specified clinical queries?

2. Is the relational algebra mapping language sufficiently powerful to map the FER schema of the test data set to each relational schema? If it is not, what are the sources of mapping "failures?"

3. Can one specify the mappings between the FER schema and the relational database schemas declaratively, or must one resort to procedural specifications (that is, external functions)?

4. Does the TransFER compiler accurately translate the ReFER queries to semantically equivalent SQL queries?
5. Are the SQL queries that the TransFER compiler generates comparable in efficiency to manually encoded SQL queries?

The high-level goal of the evaluation was to determine whether the TransFER methodology is superior to existing methods for sharing applications that query clinical databases. To answer this question, I characterized the existing methods, I defined specific performance criteria that could differentiate TransFER from the existing methods, I built a prototype implementation of TransFER, and I assessed the prototype with respect to these performance criteria. The remainder of this chapter describes each of these steps.

Section 5.2 discusses deficiencies of current methods for sharing decision-support applications and defines criteria by which one should evaluate a competing technology. Section 5.3 describes the prototype implementation of TransFER that I used to evaluate the methodology. Section 5.4 describes fully the design of the evaluation experiment that I performed and Section 5.5 presents the results of that experiment.

5.2 Evaluation Criteria for TransFER Technology

Medical decision-support applications that rely upon clinical data currently can be shared in one of several general ways. As discussed in Chapters 1 and 2, none of these methods is satisfactory. In order to outline the criteria by which TransFER may be judged superior to existing methods, it is useful to review the existing methods briefly.

> **Method 1**: Manual query execution. In this strategy, used by MYCIN [Buchanan and Shortliffe 1984]and QMR [Miller and Masarie 1990], the physician user interprets all of the application's information requests and responds to them manually by entering patient-specific data that the user knows or that is documented in the medical record.

> **Method 2**: Manual query translation. In this strategy, used by the Arden medical knowledge-representation language [Hripcsak, et al. 1990], each clinical data query that appears in an application is manually translated by a database programmer so that the new query is consistent with the local clinical database implementation.
>
> **Method 3**: Automated procedural query translation. This strategy, used by the Physician Workstation (PWS) environment [Annevelink, et al. 1992], automatically translates queries issued against a reference data model to a set of operations that retrieves data from an existing database. This method is similar to the TransFER methodology, except that the query translations in the PWS environment are based on *procedural* mappings, whereas TransFER query translations are based on declarative mappings (recall that declarative mappings allow queries to be optimized automatically).

All three methods have deficiencies with respect to at least one of three general properties: *automation*, *efficiency*, or *latitude*. The definition of each property and the trade-offs among the properties are important to consider in comparing a new technology for sharing clinical queries to existing technologies.

5.2.1 Automation

When a mismatch exists between the query model of a decision-support application and the query model of a local database implementation, several existing methods of reconciliation require that the queries be *manually* transformed. Manual query execution and manual query translation both involve labor-intensive translations: Manual query execution requires that users interpret and manually respond to all application queries; manual query translation requires that programmers manually translate all queries that appear in an application. The goal of automation is to relieve users and programmers of tedious and error-prone labor. The manual translation task is especially onerous when it must be performed frequently. Manual query execution and manual query translation require human intervention each time an existing query is issued and each time a new query is added, respectively. Thus, both methods entail a relatively high *variable cost* for reconciling query-model mismatches. A query-translation method would constitute an improvement in automation if it either reduced the variable cost by facilitating the transformation of queries or if it changed the variable cost to a fixed cost by automating the translation of queries based on encoded mappings.

An important assumption of the automation property is that the query translations are *correct*. Correct translations may be defined informally as those translations that preserve the semantic equivalence of source and target queries. In general, semantic equivalence between alternative representations of queries is difficult to verify, but I propose an empirical method for testing equivalence in Section 5.5.3. Although automation does not necessarily preserve semantic equivalence, manual methods do not guarantee correct translations either. Because “source” queries within shared applications often are encoded informally and idiosyncratically, the meanings of the queries as intended by their authors may be unclear or ambiguous. In these cases, personnel who are translating the queries manually cannot be certain that they have preserved the semantics of the original queries because they may not know definitively the semantics of the original queries [Pryor and Hripcsak 1994].

5.2.2 Efficiency

When disparate query models are reconciled via the translation of queries, either manual or automated, it is desirable that the resulting queries be as efficient as possible. In some cases, the time cost of evaluating queries is critical to the objectives of the application. For example, ICU monitors that adjust ventilator settings in real time cannot spend tens of seconds performing complex queries against disk-resident data files. In general, the costs of executing a query depend on where the data resides, how the data is represented, how the query is decomposed into subparts, which agents execute which subparts, and in which sequence the subparts are executed [Ullman 1989 Chapter 11].

The existing methods for sharing clinical queries all vary with respect to the efficiency with which the resulting queries can be executed. Manual query execution, which requires that physician users personally “execute” queries by entering data—possibly after looking it up in the patient record—is clearly the least efficient. Manual query translation, which relies upon programmers to manually translate queries that are subsequently executed by a database query engine, takes advantage of DBMS query facilities, but relies upon the ability of programmers to formulate efficient queries. Although professional programmers are the best personnel for this task, certain clinical queries must be translated into multiple subqueries and the optimal ordering of the subqueries may depend upon knowledge that the programmers do not have or that is subject to change, such as the sizes of data files and indexes. Automated procedural query translation requires that programmers specify the mappings between the query model known to decision-support applications and the query models of underlying databases; the query engine subsequently uses the mappings to translate queries from one model to another automatically. However, the query engine cannot perform certain optimizations because the mappings between query models are specified as *procedural* functions that cannot be decomposed, reordered, and recombined to achieve maximum efficiency [Annevelink, et al. 1992]. To maximize the potential for optimization, a *declarative* representation of query-model mappings is necessary. Because a query optimizer can inspect and manipulate declarative mappings based on an encoded understanding of the operations within the mappings, the optimizer can apply information about the semantics of the data, the state of the database, and the capabilities of the DBMS to determine an optimal sequence and allocation of the operations.

5.2.3 Latitude

A large variety of legacy clinical databases exists. The objective of any method for sharing clinical queries is to support the mapping of queries to as many of these databases as possible. I refer to the proportion of legacy database implementations that a method can support as the method's *latitude*. The latitudes of the existing methods for sharing decision-support applications are quite large, because translation techniques that rely on manual intervention or procedural programming are quite powerful. However, the existing methods achieve wide latitude by sacrificing automation or efficiency (as I discussed in Sections 5.2.1 and 5.2.2).

It is unlikely that an automated query-translation technique based on declarative mappings (such as TransFER) can achieve wider latitude than the existing query-translation methods. Because the ERA mapping language is confined to a small set of relational algebra operators, the transformations supported by these operators are limited to relational database implementations. However, limiting the mapping language to a small set of formal relational operators confers a theoretical foundation to the query-translation process, which enables the process to validate and to optimize automatically the query translations that it produces. This theoretical foundation results in greater automation and efficiency *for relational database implementations* than is achieved by existing methods for sharing clinical queries. The database implementations that TransFER does not support (that is, nonrelational implementations) must default to one of the existing query-translation methods (such as manual query translation), but, in doing so, remain no worse off than they are currently. Relational database implementations, meanwhile, reap whatever automation and efficiency benefits are conferred by TransFER's formal methodology. Therefore, if I can show that TransFER is superior to existing query-translation methods for just relational databases, I will show that TransFER provides an improvement over the status quo.

5.2.4 Performance criteria for TransFER

Based on the demonstrated automation, efficiency, and latitude of existing methods for sharing clinical queries, one can enumerate a set of specific performance criteria by which the TransFER methodology may be compared directly to these existing methods. The criteria embody the general properties of automation, efficiency, and latitude in a manner that may be objectively evaluated in the context of a controlled experiment. The five criteria and their informal definitions are as follows:

1. **Latitude:** the ability of the TransFER mapping language to fully map the FER schema of clinical data to most legacy relational databases, and, therefore, to support automated query translation. Poor latitude with respect to relational database implementations would indicate that TransFER cannot yet replace existing query-translation methods at *any* provider sites and, therefore, constitutes little improvement over the status quo.

2. **Declarativeness:** the ability of the TransFER mapping language to map the FER schema to most legacy relational databases without requiring

extensive use of external functions. Because external functions are procedurally specified and encapsulated, they cannot be manipulated by the query optimizer, whereas combinations of relational algebra operators can be inspected and reordered by the optimizer. Also, because the evaluation of external functions must occur in the host environment, it cannot benefit from indexes and other query-optimization features of commercial relational DBMSs. The extensive use of external functions (that is, minimal declarativeness) would indicate that TransFER is no more effective for translating queries than existing methods based on procedural mappings.

3. **Correctness:** the ability of the TransFER compiler to generate target queries that are semantically consistent with the source queries. Systematic errors in query translation may be due to unsound algorithms used by the compiler. The generation of incorrect queries would indicate that the TransFER method is unreliable and cannot replace existing methods.

4. **Efficiency:** the ability of the TransFER query optimizer to generate target queries that are efficient enough for client applications. For example, the execution of certain queries must be sufficiently fast to be performed as background processes in an integrated clinical alerting system, such as the HELP system [Kuperman, et al. 1991]. Poor efficiency would indicate that manual translation of queries is still required for database access to be fast enough for decision-support applications.

5. **Expressiveness:** the ability of the FER data model and the ReFER query language to represent the semantics of clinical queries that are typically required by decision-support applications. Inadequate expressiveness necessitates that decision-support applications perform certain data-manipulation operations themselves that otherwise could be performed more efficiently by the DBMS if the ReFER query language could express the operations. Inadequate expressiveness precludes the optimal allocation of operations between an application and a DBMS because the operations are not known to the TransFER optimizer and to the TransFER query engine.

For the TransFER methodology to be an improvement over the status quo, it must satisfy these performance criteria, which effectively constitute a lower bound for improved functionality. I assert that, although satisfying the performance criteria in the context of a controlled experiment does not prove that TransFER is generally applicable, it does suggest that TransFER is superior to existing methods and that more exhaustive evaluations are warranted.

To test TransFER with respect to the performance criteria, I conducted a controlled experiment that pitted a prototype implementation of TransFER against several heterogeneous database implementations.

5.3 Implementation of the TransFER Methodology

To evaluate the TransFER methodology, I implemented a prototype that is consistent with the specifications of TransFER as described in Chapters 3 and 4. In this section, I describe the architecture of the prototype and the development environment in which it was implemented.

5.3.1 Architecture

I implemented the TransFER methodology as a set of modules that reads a ReFER query and returns data from a relational database. The implementation is functionally divided into three *compiler modules* , which translate the ReFER query to an optimized relational database program, and one *runtime module*, which executes the database program and retrieves information stored in the relational database. Figure 5.1 illustrates the architecture of the prototype.

The compiler modules consist of (1) a TransFER attribute-grammar processor, which converts the input ReFER query into a site-specific relational algebra query, (2) a query optimizer, which eliminates redundant operations from the relational algebra query, and (3) an SQL translator, which converts the relational algebra query to a database program consisting of one or more SQL queries. The site-specific attribute-grammar processor is, itself, composed of three components: the set of site-independent semantic functions (which I discussed in Section 4.3 and enumerated in Appendix B), the site-specific mapping expressions (which database programmers defined for the local relational database), and the global FER schema (which application developers used to encode the ReFER queries). Before the mapping expressions are integrated into the attribute grammar, a mapping validator confirms that they are syntactically and semantically correct. This step is necessary because the mapping expressions are *user-defined* components of the attribute-grammar module, and therefore subject to error. For example, the mapping validator checks that all tables and fields referenced in the mapping expressions appear in the relational schema and the set of mapping expressions includes mappings for all entity and relationship types in the FER schema (although not shown in Figure 5.1, the mapping compiler uses both the FER schema and the relational schema to perform these validations). The FER schema is integrated into the attribute grammar to validate that all entity types, relationship types, and relationship functions that appear in ReFER queries are, indeed, parts of the global FER schema.

The query optimizer traverses the relational algebra expression tree generated by the attribute-grammar processor and applies any simplifying transformations that are allowed (based on the theory of PROJECT-equivalence described in Section 4.4). These transformations typically replace join and union operations with simpler subexpressions. The optimizer currently applies the transformation rules in a pre-defined order and traverses the expression tree only once.

The SQL translator subsequently traverses the simplified expression tree and identifies any relational algebra operations whose results must be stored in a temporary table; these operations include all external function calls and any relational algebra operators whose results cannot appear in the “where” clause of SQL queries (such as the UNION and AGGREGATION operators). The SQL translator separates these operations into distinct relational algebra expressions and external function invocations, translates the relational algebra expressions to SQL, and specifies the names and structures of temporary tables to store intermediate results. The output of the SQL translator is a database program consisting of a correctly sequenced list of SQL queries, external function calls, and database loading operations that implements the semantics of the original ReFER query. In this sense, the database program constitutes the “compiled” ReFER query.

The runtime module executes the database program. If the database program consists of multiple operations, the runtime engine stores the intermediate results of these operations as temporary tables in the relational database; subsequent SQL queries and external-function calls operate on these tables. The last operation in the database program generates the result of the query, which is printed to the screen.

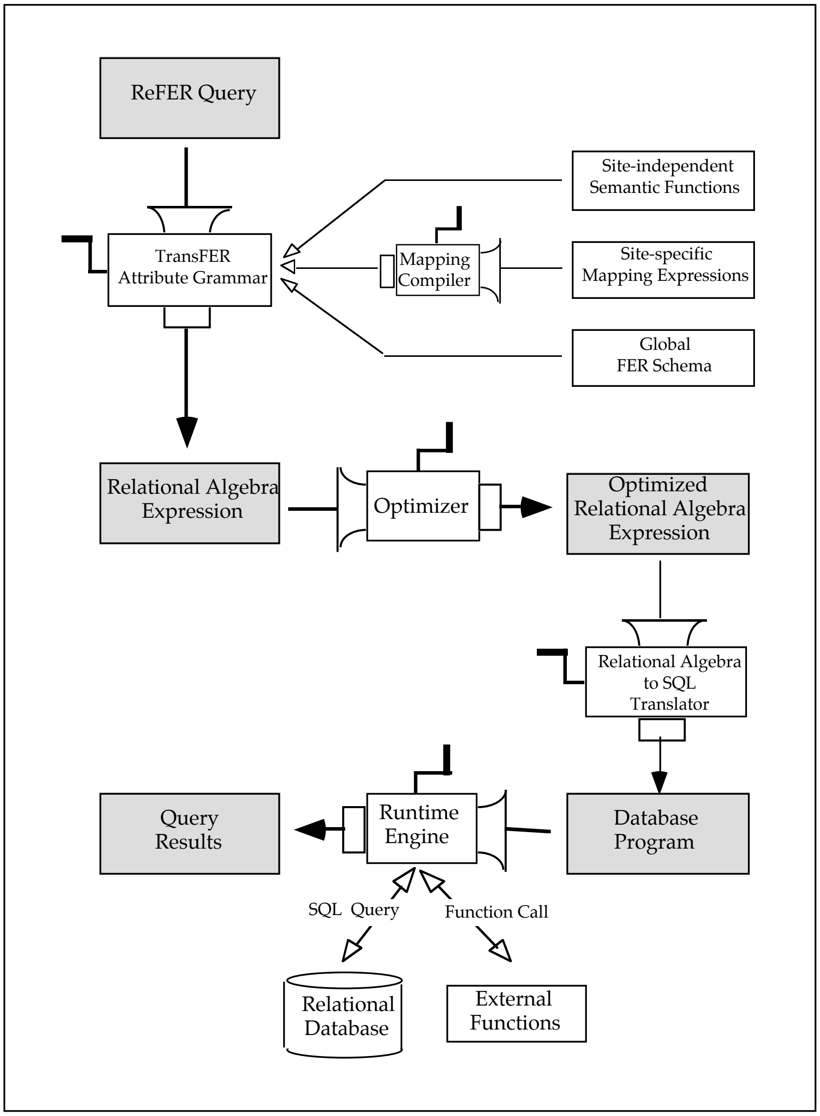


Figure 5.1: Architecture of the TransFER prototype. The translation and execution of a ReFER query may be traced by following the figure from top to bottom. The "hand crank" icons represent code modules and the boxes represent the inputs and outputs of code modules. Shaded boxes represent sequential versions of a ReFER query as it is translated and executed.

5.3.2 Development environment

I implemented the TransFER prototype in the Prolog programming language, using the Quintus Prolog[1] development environment (Version 3.2). Quintus Prolog is an implementation of Edinburgh Prolog that includes a compiler, a debugger, and many built-in utility and input/output functions. The implementation runs on a Sun SPARCstation under the SunOS operating system (Release 4.1). The prototype accesses relational databases implemented in the Sybase[2] relational DBMS (Server Release 10.0), also running on a Sun SPARCstation under SunOS 4.1. A client–server architecture is employed, such that the Prolog query-translation and runtime modules run on one processor and the Sybase DBMS on another. The Prolog application issues SQL queries to the Sybase database via the ProDBI[3] database interface (Version 3.2). ProDBI is a set of Prolog routines that connects to a Sybase server, issues SQL queries to a Sybase database, and converts retrieved query results to Prolog data structures. In addition to Sybase, ProDBI also provides access to Informix, Ingres, and Oracle relational databases.

I encountered several limitations of the development environment during my implementation of TransFER. The first was that the Sybase query engine cannot process SQL queries with more than 16 joins. Certain of the SQL queries generated by the TransFER compiler in the course of the evaluation did, in fact, contain more than 16 joins, and Sybase simply could not execute these queries . Future versions of the compiler should detect relational expressions that produce SQL queries with more than 16 joins and decompose these expressions into smaller subexpressions that yield “valid” SQL queries. Another limitation was that the ProDBI Prolog-to-Sybase interface did not import date-valued data correctly. Specifically, database values having the Sybase data type “datetime” (a data type that specifies a date and a time of day, such as “10/07/92 07:30”) were returned to Prolog with their values truncated to just the date component (for example, “10/07/92”). This loss of time-of-day information caused certain temporal comparisons that appeared in ReFER queries to be incorrect. I

---

[1] Quintus, Inc., 47212 Mission Falls Court, Fremont, CA 94539

[2] Sybase, Inc., 6475 Christie Avenue, Emeryville, CA 94608

[3] ProDBI is also available from Quintus, Inc.

circumvented this problem by converting all datetime-typed values to canonically formatted text strings before performing any temporal comparisons (such as "92/10/07 0730"), but this solution introduced extra runtime operations. Optimal performance will require a database-interface library that imports datetime-typed values without truncation.

Although I implemented the prototype on a Sun SPARCstation running the SunOS operating system, the application could be ported with minimal or no modifications to any environment jointly supported by Quintus Prolog, ProDBI, and Sybase. Currently, these environments include the following hardware platforms (with operating systems in parentheses): DEC Alpha (OSF/1), HP 9000–series 700/800 (HP-UX), IBM RS/6000 (AIX), SGI (IRIX 5), and Sun SPARC (Solaris). Furthermore, because the SQL-translator module generates standard ANSI SQL queries, one could use the TransFER prototype to access Informix, Ingres, or Oracle databases running on any of the supported hardware and operating-system environments. Although I have not yet attempted to run the TransFER prototype in any other hardware, operating-system, or DBMS environment, a significant degree of portability theoretically exists.

5.4 Structure of the Experiment

In this section, I describe the protocol of the experiment that I conducted using the prototype implementation. The objective of the experiment was to evaluate the TransFER query-translation methodology with respect to the performance criteria defined in Section 5.2.4. The experiment was a controlled trial that posed a realistic, though not exhaustive, set of database mapping problems to the TransFER methodology. I described the general design of the experiment in Section 5.1. I now discuss each sequential phase of the experiment in detail.

5.4.1 Phase 1: Independent clinical database design

Three groups of experimental subjects were given the same clinical data set and asked to design a relational database schema to represent that data set. Each group consisted of one clinician and one database expert, recruited from the Section on Medical Informatics or the Computer Science Department at Stanford. All subjects were volunteer graduate students. To avoid bias in the design of the database schemas, I chose volunteers who had minimal knowledge of the

TransFER query-translation methodology, including the FER data model and the ReFER query language.

The clinical data set that I provided to the subject groups represented a *subset* —in terms of both size and scope—of the patient-specific data that typically appear in clinical information systems. The data set described the hospital admissions and inpatient stays of two actual patients in the ICU of a major hospital. The data set consisted of approximately 40 distinct data elements (for example, "Medications") and 300 total data values (for example "Albuterol MDI 4 puffs q4"). The types of information represented included demographics, medical history, vital signs, laboratory results, and medication records. Obviously, many data that typically appear in electronic medical records are omitted, including orders, radiology reports, progress notes, and administrative information. However, the data elements that were included constituted most of the *structured* data that appear in electronic medical records today and that represent the difficult modeling challenges faced by designers of relational clinical databases. I assert that it is the modeling of these data that leads to representational heterogeneity among clinical databases. The complete data set and the instructions to the volunteer groups appear in Appendix C.

I provided the data set to the subject groups in a "representationally neutral" format so as to minimally influence the design of the relational schemas. To this end, I formatted the primary data much as they would appear in a paper medical record: The history and physical were presented in narrative form, the laboratory results were presented as episodic sets of test values, and the vital signs and ICU monitor data were presented as charted flow sheets. Also, all data values were represented by natural-language descriptors, such as "Blood Glucose," rather than "GLU," to avoid biasing the choice of coding schemes. In providing the data set in a representationally neutral format, I hoped to simulate the database design process that takes place at clinical institutions when paper-based medical records are modeled in clinical information systems. The intent of simulating the database design process was to achieve a representative degree of heterogeneity among the resulting schemas.

The result of the database design phase was a set of independently specified relational schemas. Each schema consisted of a set of valid SQL data-definition statements that could be processed by a relational DBMS to create the database structure. For the purposes of this experiment, all three groups used the Sybase DBMS. After implementing its database, each group also populated the database with the clinical data set, employing whatever coding scheme it chose to represent the data elements. The groups were given the flexibility to use existing codes drawn from standard terminologies, such as ICD9 and SNOMED, or to define their own ad hoc codes, provided that the meanings of all codes were documented.

The specific constraints imposed on the volunteer groups in designing the relational databases are summarized by the following rules:

1. The relational schemas had to represent all of the data provided in the clinical data set.
2. The groups could *enrich* the data set by adding types of objects that were not already represented, as well as instances of those types. However, the groups could not *expand* the data set by adding instances of any object types that were already represented.
3. No group could consult with any other group.

5.4.2 Phase 2: Design of the FER schema

In this phase, which took place concurrently with Phase 1, I modeled the same clinical data set given to the subject groups using the FER data model. The result was an FER schema that included all of the entity types, relationship types, and IS-A connections represented (explicitly or implicitly) in the clinical data set. Phase 2 took place before I had seen any of the relational schemas designed by the subject groups or any of the clinical queries formulated by the domain experts (see Section 5.4.3). This sequence prevented my design of the FER schema from being biased by knowledge of either the specific relational schemas to which the FER schema would be mapped or the data-access paths that the specific queries would require. Because I was evaluating the ability of the FER model and the ReFER query language to represent a wide range of database designs and clinical queries, the validity of the experiment depended on avoiding bias in these regards. The FER schema that I defined for the clinical data set appears in Appendix E.

5.4.3 Phase 3: Formulation of clinically relevant queries

Two physicians independently formulated a set of queries requesting clinically relevant information known to be available in the data set. I subsequently encoded the queries in the ReFER query language such that they were consistent with the conceptual FER schema I had specified.

The clinicians were instructed to generate queries that were representative of the varied uses of clinical data, including direct patient care, clinical practice guidelines, clinical alerts, and outcomes assessment. The queries were based on the data actually available in the clinical data set, rather than on the more complete data set that typically appears in medical records. The data were provided to the physicians in the same representationally neutral format as they were provided to the database-design groups, so that the formulation of the queries would not be biased toward any particular relational database design. This method of query formulation is consistent with the way in which queries are formulated in actual decision-support applications that will be shared: Query authors have no knowledge of the various relational implementations to which their queries will be translated.

The result of the query formulation phase was a set of 23 clinically relevant queries encoded in *natural language*. An example of such a query is "How many hours after Patient #2 was placed on Cefotetan did her temperature fall below 101?" Because there currently exist neither a standard reference model for clinical data nor a standard query language for clinical queries, natural language provided the only site-independent format in which to formulate queries that could be shared among clinical database implementations.

I subsequently translated each of the natural-language queries generated by the physicians to equivalent ReFER queries formulated with respect to the FER schema that I had defined for the clinical data set. Note that I translated the queries to ReFER before I had specified any mappings between the FER schema and the relational schemas so as not to bias my formulation of the queries based on the characteristics of the mappings. The complete set of natural-language queries that the physicians formulated and the ReFER translations of those queries that I specified are listed in Appendix F.

5.4.4 Phase 4: Mapping the relational schemas

Using the extended relational algebra (ERA) mapping language, I mapped the FER schema that I had defined to each of the relational database schemas that the volunteer groups had implemented. Recall that the FER schema and each of the relational schemas were based on the same clinical data set. For each relational schema, the mapping consisted of an ERA expression for each entity type and relationship type in the FER schema. External functions were specified where necessary and were implemented in a general-purpose programming language as part of the TransFER run-time system.

In a realistic application of the TransFER technology, the ERA mappings between the FER schema and a relational database would have been performed at each database site by personnel who are familiar with the semantics of the local database and the FER schema. However, it was not feasible in the context of my experiment for each database design group to perform the mappings, because a suitable user interface in which to specify mappings did not yet exist and because the volunteer subjects did not have sufficient time to learn the details of the TransFER model. For purposes of the experiment, therefore, I assumed that personnel at each site could perform the mappings as accurately as I could perform them myself. I can test the validity of this assumption in subsequent evaluations of TransFER.

5.4.5 Phase 5: Automatic translation of the ReFER queries

For each relational implementation of the clinical data set, I translated the set of ReFER queries to a set of corresponding SQL database programs by using the TransFER compiler. The compiler used the ERA mappings specified for each database implementation. The inputs to the TransFER compiler were the “source” ReFER queries, the FER schema, and the implementation-specific ERA mappings. The output of the compiler was a set of database programs consisting of SQL queries and calls to external functions (when required). The TransFER runtime system executed each set of the query programs against its respective database implementation and I collected and analyzed the results. I present the data produced by the evaluation experiment in the following section.

5.5 Results

Each step of the experiment yielded useful data for (1) understanding the nature of database schema heterogeneity in the context of clinical information systems and (2) assessing the effectiveness of the TransFER technology with respect to the performance criteria.

5.5.1 Database schema heterogeneity

The intent of TransFER is to render transparent to decision-support applications the heterogeneity of legacy clinical databases. The evaluation experiment is only valid, therefore, if the subject groups generated database schemas with realistic and representative degrees of heterogeneity. Such heterogeneity should include naming differences, structural differences, semantic differences, and data content differences [Deen, et al. 1987]. The three generated schemas manifested heterogeneities in each of these categories. The complete relational schemas for the three databases appear in Appendix D.

1. **Naming differences** are characterized by distinct syntactic labels for the same semantic constructs across schemas. Naming differences among the generated clinical databases included numerous variations in table names (for example “meds” versus “Medications”) and field names (for example “dose” versus “MedAmount”). Naming differences also were manifested in the coding systems used to populate the databases. For example, one database-design team used SNOMED-III codes {{ref}} to represent all medical concepts, whereas the other two teams used nonstandard numerical codes and text strings.

2. **Structural differences** entail different decompositions of semantic objects into the tables and fields of a relational schema. A structural difference among the database schemas was variation in the normalization of medication data. One schema represented in a single table the medications that patients had received and the properties of those medications (resulting in a denormalized schema); another schema represented data regarding the dosage units and routes of administration of medications in a separate reference table from data regarding the medications that patients received (resulting in a

normalized schema). Clearly, the degree of normalization influences the query operations that are required to access and associate information. Structural heterogeneity also appeared in the representation of drug allergies: One schema recorded all of the allergies for a patient as a single text string in one field of the patient demographics table, whereas another schema represented patients' allergies in a dedicated table that included one row for each allergy (see Figure 5.2).

Schema 1

PATIENT

pt-ssn	first	last	dob	Allergies
149584223	Phil	Smythe	12-2-57	Penicillin, NSAIDs
216426077	Carol	Flynn	3-25-62	Morphine, ASA

Schema 2

PATIENT_ALLERGIES

pt-ssn	Allergy
149584223	Penicillin
149584223	NSAIDs
216426077	Morphine
216426077	ASA

> and medication stop times from one of the schemas. For both data elements, the omitted values were assumed to be NULL, although this value was not necessarily consistent with the values represented in the other databases. When a data element that appears in the FER schema is omitted from a database, the database mapper either must make an assumption about the value of the missing datum or must assign a special value to the missing datum indicating that it is unknown.

Given that significant heterogeneity did exist among the clinical database schemas that were generated, the performance of the TransFER technology in the context of the experiment is a meaningful, albeit preliminary, indicator of the utility of TransFER in the "real world." The following sections discuss TransFER's performance with respect to each of the criteria that I defined in Section 5.2.

5.5.2 Latitude and declarativeness

Table 5.1 addresses the performance of TransFER with respect to latitude and declarativeness.

Schema	Mapping Yield (%)	External Function Use (%)
Schema 1	145/183 (79%)	12/145 (8%)
Schema 2	183/183 (100%)	13/183 (7%)
Schema 3	183/183 (100%)	13/183 (7%)

Table 5.1: Success rates for mapping the FER schema to three relational schemas. Mapping Yield indicates the proportion of entity types and relationship types in the FER schema that were mapped successfully to the relational schema. External Function Use indicates the proportion of entity-type and relationship-type mappings that included calls to external functions.

The "Mapping Yield" column in Table 5.1 indicates the proportion of entity types and relationship types in the FER reference schema that I successfully mapped to each of the three relational schemas using the relational algebra mapping language.

Although I was able to map Schemas 2 and 3 completely to the FER reference schema, Schema 1 proved problematic because its designers chose not to model clinical diagnoses and medical procedures in a structured relational format. Rather, the diagnoses and procedures associated with each patient were represented in a single text field as a narrative description of the patient's chief complaint and past medical history. The contents of this field resembled the following text string: “asthma since age 8, left nasal polyp removal 1985, chronic sinusitis since 1980.” The extraction of distinct disease and procedure entities from such narratives was beyond the capabilities of the relational algebra or of any external functions available to me. In this sense, Schema 1 was beyond the “latitude” of TransFER. However, of the comparative query-translation techniques, only the manual inspection and interpretation of Schema 1 could have achieved a better result given current technology. In the future, natural-language processing techniques for clinical narratives [Sager, et al. 1994] may become sufficiently powerful and reliable to extract structured information from free-text descriptions. For the present, however, disease and procedure data modeled as free-text narratives are beyond the latitude of any automated retrieval techniques.

The “External Function Use” column of Table 5.1 denotes the proportion of mappings that required at least one external function call to represent accurately the semantics of FER entity types or relationship types. As the data indicate, external functions were required sparingly. Operations requiring external functions included (1) the decomposition of a single text field containing multiple allergies into a set of text fields each containing a single allergy and (2) the data-type conversion of date-valued fields to a format that readily supported temporal comparisons. Given the degree of heterogeneity among the generated relational schemas, the infrequent need for external functions indicates the power of the relational algebra as a mapping language and bodes well for the optimization potential of queries generated by the TransFER methodology.

5.5.3 Expressiveness and correctness

Table 5.2 addresses the performance of TransFER with respect to expressiveness and correctness. The ability of the ReFER query language to express the clinical queries formulated in Phase 3 of the experiment reflects the adequacy of the TransFER model in supporting the data-access needs of decision-support applications.

Parameter	Value
Physician queries encoded in ReFER (%)	70% (16/23)
Translated queries returning correct values (%)	92% (36/39)

Table 5.2: The expressiveness and correctness of TransFER. The percentage of queries successfully encoded in ReFER reflects the expressiveness of the ReFER query language. The percentage of translated queries returning correct values reflects the correctness of the TransFER query-translation methodology.

The data in the first row of Table 5.2 show that seven of the 23 queries specified by physicians as relevant to decision support (30%) could not be encoded in the ReFER query language. Five of the seven queries could not be encoded because ReFER currently lacks constructs to perform certain temporal operations, such as retrieving the three most recent values of a parameter or computing the difference in hours between two time points. For example, the following specific queries (as formulated by the physician experts) could not be supported by the ReFER query language:

1. Retrieve the last 3 arterial blood gas (ABG) measurements for Patient #1 as of 1:00 p.m. on August 27, 1993.

2. Retrieve all of the ABG measurements for Patient #2 within 6 hours before and 1 hour after mechanical ventilation.

The 30% failure rate for encoding clinically relevant queries implies that the addition of more powerful temporal operators to the query language would improve TransFER's ability to directly support certain queries. However, even in the absence of such operators, decision-support applications can use the TransFER model as an interface to clinical databases provided that the applications implement certain temporal operations themselves. For example, to derive the last three ABG measurements for a patient, an application could retrieve *all* of the ABG measurements and subsequently filter out all but the three most recent.

Each of the ReFER queries subsequently was processed by the translating compiler three times, once using each set of schema-specific ERA mappings. The result of each translation was a database program consisting of SQL queries interspersed with external function calls. When executed against the appropriate relational database, each program should have returned the data requested in the original ReFER query. Any discrepancy between the data originally requested and the data ultimately retrieved constituted a correctness failure. Of the 48 potential query translations, 39 produced "valid" database programs. The remaining nine translations produced programs that were not valid because at least one of the generated SQL queries referenced more than 16 relational tables, an arbitrary limit of the database software used for the evaluation (Sybase).

When the 39 valid translations were executed, 36 (92%) produced correct results. The gold standard for correctness was determined by manually inspecting the original data set in its paper-based format. The modest size of the data set made such a manual inspection feasible. Of the three cases in which incorrect data were returned, a different reason was responsible for each failure:

1. One query failed due to an erroneous mapping expression; when I corrected the mapping error and recompiled the query, the correct data was returned. Mapping errors may be subtle and remain undetected in the absence of rigorous testing. This fact points to the need for extensive test suites of standardized data and queries to increase the likelihood of detecting errors.

2. Another query failed because one of the subject groups populated its relational schema in a manner inconsistent with the semantics of the original clinical data set; after we modified the relational data, the same query produced correct results.

3. The third failure occurred because one relational schema contained no data regarding patient allergies, although this data was included in the clinical data set. When I mapped the FER relationship type representing each patient's allergies to this schema, I assumed that patients had no allergies because no data existed from which to derive an appropriate set of allergies. This assumption was incorrect and caused incorrect data to be retrieved by a query. Missing clinical data elements pose a significant problem for the sharing of clinical decision-support applications, because incorrect assumptions regarding the values of such elements may affect the validity of inferences in ways that are not apparent to users. The alternative to making assumptions regarding the values of missing data at the time that a relational schema

> is mapped is to assign a special value to those data that notifies the application or the user that the requested information is not available. This alternative allows an application to incorporate the knowledge that certain data is missing into its logic and to respond in an application-specific and appropriate manner (for example, by querying the user).

5.5.4 Efficiency

To assess the efficiency of the relational queries produced by the TransFER query-translation methodology, I compared the 39 TransFER-generated database programs to a corresponding set of 39 database programs that I *manually encoded* for the same ReFER queries. The manually encoded database programs constitute a *gold standard* of efficiency in the sense that I formulated these queries with the intent of specifying the minimal set of SQL operations required to retrieve the requested data. Recall from Section 4.4 that the TransFER query-translation methodology may introduce redundant relational operations into the translations of ReFER queries, which result in relational expressions that are significantly more complex than necessary; the query optimizer subsequently detects and eliminates certain of these operations in an attempt to arrive at a minimal relational algebra expression that has the same meaning as the original expression. The purpose of directly comparing the TransFER-generated database programs to the gold-standard database programs is to assess the effectiveness of the query optimizer in removing unnecessary operations and in automatically generating SQL queries that are as efficient as manually generated queries.

I characterize the complexity of each database program generated by TransFER in terms of two parameters: (1) the number of distinct SQL queries in each database program ("SQL queries/ReFER query") and (2) the *Join Index* of each database program ("Join Index/ReFER query"). The Join Index (JI) of a database program is defined as the total number of joins across all of the SQL queries in the program *plus* two times the number of SQL queries in the program. The JI is an approximate measure of the complexity of database programs that consist of multiple SQL queries. The assumption underlying this metric is that the overhead associated with each distinct SQL query in a database program is roughly equivalent to two relational joins. I use relational joins as a measure of complexity because they typically constitute the bulk of the run-time processing of SQL queries. This definition of the JI implies, for example, that a database program consisting of five SQL queries that each contain one join (JI = 15) is more costly to execute than a database program consisting of one SQL query containing five joins (JI = 7).

Parameter	Manual translation	TransFER translation (no optimization)	TransFER translation (partial optimization)
Average # SQL queries/ReFER query	1.1 (1–2)	5.5 (3–11)	3.5 (1–9)
Average JI/ReFER query	4.4 (2–9)	22.9 (9–54)	17.1 (2–44)

Table 5.3: The complexity of TransFER-generated queries Numbers in parentheses indicate the range of values of each parameter.

Table 5.3 shows the average number of SQL queries and the average join index for the entire set of 39 database programs generated by the TransFER compiler. The parameters were calculated once before any optimizations were implemented (column 3 of Table 5.3) and once after a subset of the proposed optimizations was implemented (column 4 of Table 5.3). The results in column 4 do not represent the truly optimal complexity of TransFER-generated queries because not all of the potential optimizations described in Chapter 4 were implemented at the time I obtained the results. Column 2 of Table 5.3 shows the average number of SQL queries and the average join index for the gold-standard database programs that I generated manually.

It is useful to consider the complexity of the original ReFER queries that formed the basis of the translations characterized in Table 5.3. The 39 ReFER queries that were translated to database programs contained, on average, one term, two declarations, and four predicates. The following ReFER query, therefore, is representative of the average complexity of the 39 translated queries:

```
[ value-of(na) | (pt : Patient), (na : Serum_Na) AND
                 Patient-Test(pt,na) AND
                 last-name-of(pt) = “Smythe” AND
                 time-of(na) > “1995/09/20 0000” AND
                 time-of(na) < “1995/09/21 0000” ]
```

The data in Table 5.3 indicate that the automatically generated database programs (before optimization) are more complex than the ideal (gold-standard) programs by a factor of five to one. Direct comparison of corresponding database programs confirms that superfluous JOIN and UNION operations in the TransFER-generated programs are, in fact, responsible for the disparity in complexity. However, the data also indicate that implementation of just a subset of the optimization rules described in Section 4.4 results in a 20 percent reduction in complexity. This result is promising because it indicates that optimization techniques based on the principles of relational algebra can significantly improve the efficiency of TransFER-generated queries. The development and implementation of additional optimization rules will likely lead to further performance improvements, bringing the efficiency of TransFER-generated database programs closer to that of the gold standard. I will continue to use the join-index metric to gauge progress as the development of optimizations continues.

The definition of formal evaluation criteria, the implementation of a working prototype, and the execution of a carefully planned experiment allowed me to rigorously assess the merits of the TransFER methodology with respect to a non-trivial query-translation problem. In the next chapter, I discuss the value as well as the limitations of the experimental results that I obtained. I also look ahead to future research based on the TransFER model for query translation, and I summarize the contributions of the TransFER methodology to sharing decision-support applications.

Chapter 6

Summary and Conclusions

In this dissertation, I have described and evaluated a novel technology to facilitate the sharing of decision-support applications. Predicated on the belief that decision-support applications should access clinical databases directly and that clinical databases will remain heterogeneous in their organization and implementation, the TransFER technology facilitates sharing by *automatically translating database queries to a variety of equivalent formulations* so that decision-support applications can retrieve data directly from a variety of clinical databases. Although TransFER's query-translation methodology supports databases implemented in the *relational model* only, the methodology accommodates a large space of different relational database implementations. These implementations may vary in their relational representations of real-world objects, in their naming of tables and fields, in their coding of medical data, and in their dialects of relational query languages. The TransFER technology automatically resolves many of these heterogeneities so that developers can embed queries in decision-support applications without concerning themselves with the implementation details of existing clinical databases.

6.1 The Essence of TransFER

In Chapter 1, I described how heterogeneity among database implementations currently impedes the sharing of decision-support applications that retrieve data directly from clinical databases. The crux of the problem is that database queries embedded in decision-support applications must be customized at each site where the application is deployed, and this process is costly, time-consuming, and error-prone. In Chapter 2, I discussed previous research on standard data models and automated query-translation methods to reduce the costs and errors associated with customizing database queries. Although significant work has been done by distributed database developers, artificial intelligence researchers, and informatics-standards bodies, I concluded that no satisfactory integration of data-modeling and query-translation methods yet exists to support fully the sharing of decision-support applications.

In Chapters 3 and 4, I presented my own methodology for customizing database queries, called TransFER, which automatically translates queries based on the encoded specifications of various relational databases. The methodology consists of four formal components:

1. A semantic data model for representing the contents of clinical databases
2. A logic-based query language for encoding queries in a database-independent way
3. A database-mapping language based on a variation of the relational algebra
4. A query-translation algorithm based on the theory of attribute grammars

The data model and query language are sufficiently abstract that they allow developers of decision-support applications to encode queries in manner that *subsumes* the database schema and query language of any specific relational database implementation. The mapping language allows database experts at each site to encode declaratively the correspondence between the *global abstract schema* of clinical information (specified in the semantic data model) and the *local implemented schema* of clinical information (specified in the relational data model). The query-translation algorithm applies the encoded mappings at each site to automatically translate queries that appear in decision-support applications to equivalent relational queries that are specific to the database structure, clinical vocabulary, and query language in use at that site. The result of the translation is a database program consisting of SQL queries (which the local database engine can execute) and calls to arbitrary functions (which the local computing environment can process). The appropriate interleaving of SQL queries and function calls retrieves from the local database the clinical data requested by the decision-support application.

Although other methods exist for automatically translating database queries from a semantic data model to the relational data model, the TransFER methodology is novel in that it bases the query translation on *explicit* and *declarative* mappings between *individual constructs* of the semantic database schema and a relational database schema. The mapping of individual schema constructs in a declarative language simplifies the database-mapping process for human mappers by dividing the task into discrete modular steps and by supporting automated validation. The explicit specification of mappings facilitates the automated query-translation process by representing precisely which transformations are required to effect the translation, rather than requiring a compiler to search the space of possible transformations. The declarative representation of mappings in the extended relational algebra language supports the optimization of translated queries via equivalence-preserving transformations based on the formal properties of the relational algebra; I have defined and proven formally a set of such transformations. The explicit, declarative, and modular nature of the mappings allows queries that appear in decision-supported applications to be customized more easily, accurately, and efficiently than was possible with previous methods.

6.2 The Meaning of the Evaluation

In Chapter 5, I presented and discussed the results of an experiment I conducted to test the effectiveness of my methodology in practice by translating queries to three independently designed clinical databases. Three groups of volunteer subjects generated the relational databases based on a single clinical data set, and two physicians specified a set of representative queries with respect to the same clinical data set. The TransFER compiler subsequently translated the queries to three sets of database-specific programs based on mappings that I specified for each database implementation.

Given that the generated relational databases manifested significant heterogeneity, the experimental data was valid for evaluating the TransFER methodology. By assessing TransFER with respect to the five performance criteria that I defined in Section 5.2.4, I can now consider whether TransFER is superior to existing methods for sharing applications that access clinical databases. I address TransFER's performance with respect to each criterion in the following sections.

6.2.1 Latitude

The relational algebra mapping language handled heterogeneity in the *structured* representation of clinical data; this heterogeneity included table/field naming differences, coding differences, denormalization, and implicit versus explicit modeling. The mapping methodology could not accommodate certain unstructured representations, such as free-text fields that contained narrative clinical descriptions. However, no existing method for translating queries such that they can be executed automatically accommodates all narrative descriptions either. The latitude of TransFER is, therefore, equivalent to existing query-translation methods. When natural-language processing techniques become available to structure narrative clinical data, the extensibility of the TransFER mapping language via external-function calls will be able to exploit the technology to increase the latitute of the TransFER methodology.

6.2.2 Declarativeness

Less than 10 percent of the mappings across all of the schemas included external function calls. The Physician Workstation (PWS) environment, the only existing method that performs query translation automatically, requires procedural functions for every mapping. TransFER, therefore, provides significantly greater potential for query optimization than PWS because the declarative semantics of the relational mapping language support automated query transformation. I demonstrated this advantage by showing a 20 percent improvement in the efficiency of generated database programs when I implemented optimization rules based on the formal properties of the relational algebra.

6.2.3 Expressiveness

Although nearly a third of the queries generated by the physician experts could not be encoded completely in the ReFER query language, the missing query constructs were primarily aggregation and arithmetic-manipulation operators for temporal data (as opposed to data-retrieval operators). In practice, this deficiency is not a critical limitation because decision-support applications can perform aggregation or arithmetic manipulation of temporal data themselves after the TransFER interface has retrieved a superset of the required data. Furthermore, the standard SQL query language itself does not contain aggregation operators or arithmetic operators for temporal data; this implies that no query-translation method can generate SQL queries that capture the semantics of the information requests more fully than ReFER.

6.2.4 Correctness

The errors of correctness in the evaluation experiment resulted from missing data in the relational implementations or from faulty mapping specifications, rather than from any intrinsic flaws in the query-translation mechanism. If one assumes that the relational databases to which an FER schema is mapped contain all of the data represented in the FER schema and that the mappings for all entity types and relationship types in the FER schema are defined correctly, then the data from the experiment indicates that the TransFER methodology translates queries with 100 percent accuracy. This result confirms that the query-translation methodology, *per se*, is sound. Although, in practice, certain relational databases may not contain all of the information represented in the FER schema, these cases could be handled by extending the FER data model to include constructs that explicitly denote occurrences of missing data; this information would allow applications to respond to the absence of requested data in an appropriate manner (such as querying the user).

6.2.5 Efficiency

The data from the experiment clearly identify efficiency as the greatest current limitation of the TransFER methodology compared to existing manual query-translation methods. The four-fold difference in complexity between TransFER-generated database programs and manually generated database programs probably precludes the clinical use of TransFER at this time. However, the opportunity for optimization, both theoretical and realized, that TransFER's declarative mappings provide constitutes an improvement over any automated query-translation methods that currently exist. Further investigation and implementation of query-optimization techniques ultimately may make the efficiency of TransFER-generated query translations comparable to that of manually generated translations. Given that TransFER is already competitive with existing methods in terms of latitude, declarativeness, expressiveness, and correctness, significant advancements in efficiency may well establish TransFER as the preferred method for sharing decision-support applications that access clinical databases.

6.2.6 Generalizing the results

The experiment that I conducted provided an initial evaluation of the power and generality of the TransFER translation methodology. In designing the experiment, I took steps to create as realistic a set of problems as possible against which to test TransFER's capabilities. These steps included acquiring actual clinical data, transforming the clinical data to a representationally neutral form, taking care to not bias participants, and carefully ordering the sequence of data-modeling and query-formulation tasks. However, even accounting for these measures, the problems posed to the TransFER technology by this experiment were not *complete* with respect to the heterogeneity that exists among “real-world” clinical databases. To determine the meaning of the experiment and the generalizability of the results, one must characterize the nature and the extent of the differences between “real-world” database heterogeneity and the database heterogeneity encountered in the controlled experiment. Specifically, the experiment simplified the query-translation problem in at least five ways.

6.2.6.1 A single clinical data set

I distributed to each of the database-design groups the same clinical data set that I used to design the FER schema. Because I instructed each database-design group to include in its relational database implementation *all* of the data represented in the clinical data set, almost all data elements modeled in the FER schema appeared in each relational schema. This situation is unrealistic, because certain “real-world” legacy databases may be missing data elements that appear in the FER schema; missing data may make the mapping problem more difficult and may compromise the correctness of applications that use the TransFER model. However, the issue of FER data elements missing from relational database implementations is not central to the evaluation of the TransFER translation methodology, *per se*, because the translation methodology assumes that all target databases contain the same information as the FER schema. I can test the validity of this assumption in subsequent evaluation experiments and develop methods to inform applications that information is missing, if necessary.

6.2.6.2 A circumscribed clinical data set

The scope of the clinical data set that I distributed to the database-design groups was significantly smaller than the scope of data sets that are modeled in real-world clinical databases. A larger number of data elements increases the complexity of the relational database designs and results in greater representational heterogeneity among the designs. For example, the data set used in the experiment included only ICU patients, whereas a real-world data set might also include outpatients, ward inpatients, etc., resulting in a more complex relational schema. However, although the clinical data set that I used did not generate a complete set of representational heterogeneities, it was sufficiently large and complex to provide an opportunity to address several important issues of representational heterogeneity. As described in Section 5.5.1, representational heterogeneities of several kinds appeared among the three relational databases that were generated for the experiment. These heterogeneities provided a set of realistic and non-trivial mapping problems for the TransFER methodology.

6.2.6.3 A limited number of target databases

Realistically, TransFER will need to map the FER query model to clinical databases at *hundreds* of provider sites. The experiment that I conducted required TransFER to map the FER query model to only a handful of clinical databases. A greater number of target databases undoubtedly will create a greater degree of representational heterogeneity to be resolved by the TransFER methodology. However, the degree of heterogeneity probably is not proportional to the number of clinical database sites: There may be only a finite number of ways in which typical clinical data is modeled relationally, and certain sites may share the same relational implementation because they have purchased commercial clinical databases from the same vendor. Although a handful of clinical database implementations did not exhaustively test TransFER's ability to resolve representational heterogeneities, it did indicate whether the TransFER methodology is a *feasible* means of resolving the heterogeneities that typically arise in the representation of clinical data.

6.2.6.4 A limited number of clinical queries

The FER clinical data schema and the TransFER query-translation methodology are intended to support the data-access needs of numerous and diverse decision-support applications. Although the physicians who generated the set of test queries for the experiment attempted to simulate these data-access needs as completely as possible, the full range of queries required by medical decision-support applications could not be represented by the relatively modest set of queries that the physicians formulated. However, the query authors were encouraged to produce and had the background to produce a diversity of data-access requests related to patient care, clinical research, outcomes assessment, and clinical practice guidelines. Although the queries were not exhaustive, they were representative in terms of content, complexity, and diversity.

6.2.6.5 A single relational DBMS

All of the subject groups implemented the relational databases that they designed using the Sybase relational DBMS. Realistically, health care institutions implement clinical databases using a variety of relational DBMS platforms, such as Sybase, Oracle, Informix, IBM DB2, etc. Each of these platforms differs slightly in the dialect of SQL that it provides as a query interface. By requiring all of the database-design groups to use Sybase, I avoided the added complication of translating ReFER queries to multiple dialects of SQL. However this simplification does not detract appreciably from the validity of the evaluation because the objective of the TransFER methodology is to resolve *representational heterogeneity* rather than *access heterogeneity* among clinical databases [Sheth and Larson 1990]. Access heterogeneity entails primarily differences in network connections to database servers, application programming interfaces to DBMSs, and syntactic details of SQL. Several commercial vendors already have addressed the problem of access heterogeneity among relational DBMSs [Burleson 1995], and I could incorporate these technologies into future versions of TransFER to provide access to various database-system platforms.

Because of the factors described in Sections 6.2.6.1–6.2.6.5, the problems that I presented to the TransFER methodology in the evaluation experiment were not *complete* with respect to the representational heterogeneities that exist among real-world clinical databases. However, the problems were *representative* of those heterogeneities. Because the experiment provided a realistic, representative, and non-trivial test of TransFER's capabilities, the favorable results indicate that TransFER is a significant novel technology with potential utility for sharing decision-support applications.

The results of the experiment also indicate that additional research and evaluation is required before one can deploy TransFER as a general technology for translating database queries automatically across heterogeneous database sites.

6.3 Future Work

Although the TransFER methodology shows promise as an effective model for sharing decision-support applications that include database queries, more work is required before TransFER can be applied to solve the real-world problems of application portability. Outstanding research issues include:

- Development of a global FER schema of clinical information that is sufficiently comprehensive to support a variety of clinical decision-support applications yet sufficiently constrained to represent the contents of most clinical databases

- Development and implementation of additional optimization techniques to reduce further the complexity of relational queries generated by the TransFER compiler

- Development of application programmer interfaces (APIs) to TransFER from a number of programming languages, so that ReFER queries can be embedded in and integrated with the code of decision-support applications

- Development of a sound method to represent that a certain datum is not available in a clinical database and to notify decision-support applications of this fact at the time the datum is requested

- Design and execution of further evaluation experiments that involve actual clinical databases and decision-support applications

Work in these areas is necessary before the concept of TransFER described in this dissertation can be realized in practice. Several extensions to the TransFER model are also worth investigating:

- Development of additional query-translation algorithms that can translate ReFER queries to the query languages of *nonrelational* legacy database systems, such as object-oriented databases and hierarchical databases
- Enhancement of the TransFER query-translation algorithm to aggregate data from *multiple* heterogeneous databases, so that TransFER is more valuable for multisite research and decision-support applications
- Combining the FER data model with the type system of a programming language to create a *database programming language* [Atkinson and Buneman 1987] , which could be used to build decision-support applications that efficiently combine data access with knowledge-based inference.

6.4 Conclusions

Computerized decision-support tools may well become a critical resource in a complex healthcare system that faces pressure to maintain quality while reducing costs. The effectiveness and acceptance of these tools depends, at least partially, on their ability to access up-to-the-minute patient-specific data in an accurate, timely, and unobtrusive manner. As electronic clinical databases increase in prevalence and sophistication, they offer fabulous potential to provide data to decision-support tools quickly and automatically, allowing the tools to be integrated smoothly into the healthcare-delivery process. This desirable model of application–database interoperability, however, is significantly complicated by the extensive and multifaceted heterogeneity that exists and will continue to exist among clinical databases. Database heterogeneity makes it exceedingly difficult to deploy an effective and proven decision-support tool across multiple healthcare-provider sites because the tool must be painstakingly customized to the highly variable database specifications at each site. Until now, no technology has been available to automate the customization process in a principled and efficient manner.

The TransFER query-translation methodology constitutes an important step towards such a technology. The formal data model and query language allow the developers of decision-support applications to specify queries precisely enough to be unambiguous, yet abstractly enough to be portable across heterogeneous databases. The modular, powerful, and declarative database-mapping language allows database programmers to specify schema-to-schema transformations for a large variety of relational databases in a manner that is intuitive and that supports efficient query translation and automated query optimization. The attribute-grammar-based translation model allows the TransFER compiler to use a fixed set of mapping expressions to translate an arbitrary number of queries such that the resulting database programs are semantically equivalent and locally executable. Although the computational efficiency of automatically translated queries may be currently insufficient to meet the demands of many decision-support applications, the formal elements of the query-translation model support the development of many further optimization techniques. Additionally, the extensibility of the mapping language via external-function calls allows TransFER to incorporate future technologies for resolving database heterogeneity into the current query-translation model. Given the demonstrated soundness of the basic query-translation model, more powerful optimization techniques and more advanced database-mapping technologies may well complete the link provided by TransFER between decision-support applications and heterogeneous relational databases.

Appendix A

The ERA Mapping Language

This appendix specifies the extended relational algebra (ERA) mapping language that is used to encode the site-specific portion of a TransFER attribute grammar. Section A.1 contains the context-free grammar of the mapping language in standard Backus-Naur form (BNF) [Sammet 1969, Chap. 2, Sec. 6]. A sample mapping between a specific FER schema and relational database schema appears in Section A.2. Section A.3 discusses the general strengths and limitations of the mapping language with respect to resolving heterogeneity among relational database implementations.

A.1 Context-free grammar of the ERA mapping language

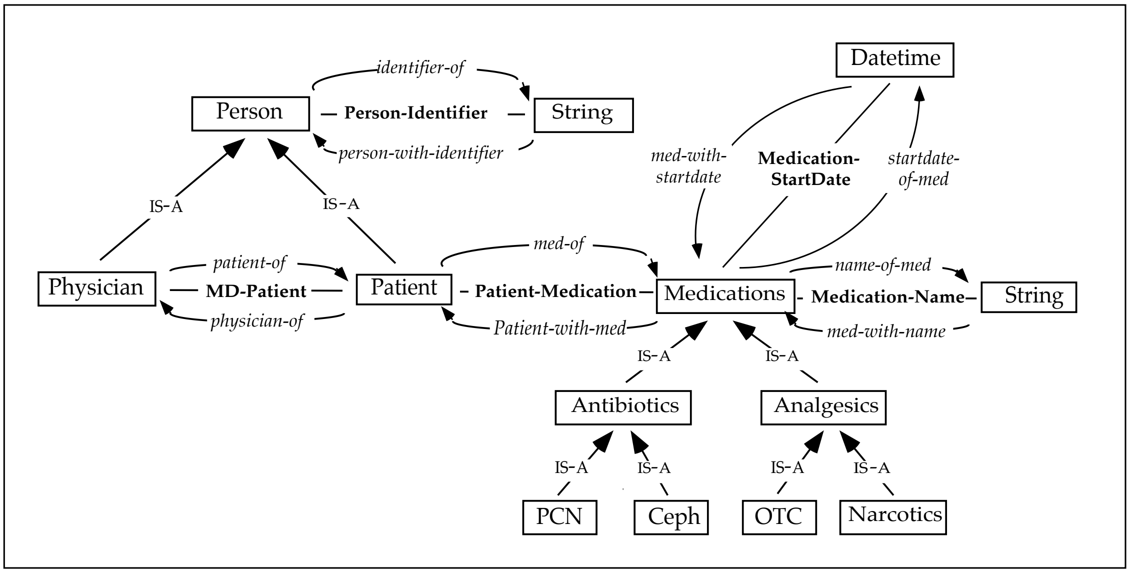
```
<mapping> ::= <defn-list>

<defn-list> ::= <defn>,<defn-list>  |
                <null>

<defn> ::= <entity-defn>  |
           <relship-defn> |
           <virt-defn>    |
           <built-in-entity-defn>

<entity-defn> ::= defEntity(name(<id-atom>),
                            era(<table-defn>),
                            id(<id-defn>)).

<relship-defn> ::= defRelship(name(<id-atom>),
                              era(<table-defn>),
                              id(<id-defn>),
                              fids([<fid-defns>])).
```

```
<virt-defn> ::= defVirtual(name(<id-atom>),   %define virtual relation
                            attrs([<virt-attr-defn>,
                                   <virt-attr-defn>])).

<built-in-entity-defn> ::= defBuiltInEntity(name(<id-atom>)).

<table-defn> ::= rel(<id-atom>)                                   |
                 vrel(<id-atom>)                                  |
                 rename([<rename-list>],<table-defn>)             |
                 extend([<extend-list>],<table-defn>)             |
                 proj([<id-atom-list>],<table-defn>)              |
                 sel([<sel-pred>],<table-defn>)                   |
                 join([<join-pred-list>],<table-defn>,<table-defn>) |
                 union(<table-defn>,<table-defn>)                 |
                 aggregate(aggparms(<parameters>),<table-defn>)

<rename-list> ::= <rename-item>                       |
                  <rename-item>,<rename-list>

<rename-item> ::= <id-atom> => <id-atom>

<extend-list> ::= <extend-item>                       |
                  <extend-item>,<extend-list>

<extend-item> ::= <id-atom> == <factor>

<fxn-designation> ::= sql | ext  %fxn is either SQL built-in or external

<factor-list>   ::= <factor>                      |
                    <factor>,<factor-list>

<id-atom-list> ::= <id-atom>                      |
                   <id-atom>,<id-atom-list>

<factor> ::= <id-atom>            |
             const(<const-atom>)  |


<const-atom> ::= <quoted-string-atom>  |  e.g. '"PT"'
                 <integer-atom>        |  e.g. '7'
                 <float-atom>          |  e.g. '6.4'
                 <date-string-atom>       e.g. '"12/02/63"'
```

```
<sel-pred> ::= and(<sel-pred-list>)   |
               or(<sel-pred-list>)    |
               not(<sel-pred>)        |
               null(<id-atom>)        |
               not_null(<id-atom>)    |
               <factor> <comp-operator> <factor>

<sel-pred-list> ::= <sel-pred>                 |
                    <sel-pred>,<sel-pred-list>

<comp-operator> ::= <> | = | > | < | >= | <=

<join-pred-list> ::= <join-pred>,<join-pred-list> |
                     <null>

<join-pred> ::= <factor> = <factor>

<id-defn> ::= [id-atom-list]

<parameters> ::= [<arg-list>],[<group-list>],
                 <agg-fxn-designation>,[<new-attr-list>]

<arg-list> ::= <id-atom-list>
<group-list> ::= <id-atom-list>
<new-attr-list> ::= <id-atom-list>

<agg-fxn-designation> ::= aggsql(<id-atom>) |  %sql built-in aggregator
                          aggext(<id-atom>)    %external aggregator

<fid-defns> ::= <fid-defn>,<fid-defns>  |
                <null>

<fid-defn> ::= fid(<id-atom>,<factor-list>)

<virt-attr-defn> ::= attrdef([<id-atom-list>],[<attr-defn>] )

<attr-defn> ::= const(<const-atom>)
                fxn(<fxn-designation>(<id-atom>),[<factor-list>]) |
                undefinedFxn([<id-atom>,  %name of virtual relationship
                             [<id-atom-list>],    %source entity type
                             [<id-atom-list>] ])  %target entity type

<id-atom> ::=  any prolog atom (see language reference)
```

A.2 A sample ERA mapping

This section contains a small but complete ERA mapping between an FER schema and a relational database schema. Although the sample mapping does not illustrate all of the features of the mapping language, it provides a representative example of how one encodes mappings between an FER schema and a relational database schema. The FER schema appears in Figure A.1 and the corresponding relational database schema appears in Section A.2.1. The ERA mapping between the FER and relational schemas appears in Section A.2.2.



between the codes used in the relational database to denote medications and the entity-type names used in the FER schema to denote the same medications.

```
create table Patient (
        mr_number               char(11)        not null,
        last_name               varchar(15)     null,
        first_name              varchar(15)     null,
        birthdate               varchar(15)     null)

create table Physician (
        dea_number              char(6)         not null,
        last_name               varchar(15)     null,
        first_name              varchar(15)     null,
        birthdate               varchar(15)     null,
        specialty               varchar(20)     null )

create table Admission (
        mr_number               char(11)        not null,
        dea_number              char(6)         not null,
        admit_datetime          varchar(15)     not null,
        location                varchar(10)     null,
        discharge_datetime      varchar(15)     null )

create table Therapy (
        mr_number               char(11)        not null,
        med_code                char(3)         not null,
        start_datetime          datetime        not null,
        end_datetime            datetime        null,
        quantity                varchar(6)      null)

/*****************************************************************/
/* The following tables are not part of the original database
   schema.  They were added to the schema later (by the schema
   mapper) to support the mapping of the database to the FER schema */
/*****************************************************************/

create table MEDICATION_CODES (
        med_code     varchar(50)       not null,   %site-specific code
        ent_type     varchar(50)       not null)   %Corresponding FER
                                                    entity type

Example contents:       med_code          ent_type
                        125               "Ceph"
                        104               "PCN"
                         70               "Narcotics"
```

```
/** ADDITIONAL TABLES FOR TYPE HIERARCHY AND VOCABULARY MAPPING  **/

create table FER_TYPE_HIERARCHY (
        ent_type          varchar(50)   not null,  %FER entity type
        desc_ent_type     varchar(50)   not null)  %descendant type
                                                    in type hierarchy

Example contents:       ent type          desc ent type
                        "Antibiotics"     "Ceph"
                        "Antibiotics"     "PCN"
                        "Analgesics"      "Narcotics"
                        "Analgesics"      "OTC"


create table ENTITY_TYPE_NAMES (
        ent_type             varchar(50)     not null,
        ent_type_name        varchar(50)     not null)

Example contents:       ent type          ent type name
                        "Ceph"            "Cephalosporin"
                        "PCN"             "Penicillin"
                        "OTC"             "over-the-counter analgesic"
```

A.2.2 The ERA mapping expressions

The complete mapping between the FER schema in Figure A.1 and the relational schema in Section A.2.1 appears in this section. The mapping specifications conform to the context-free grammar defined in Section A.1. Note that each entity type and each relationship type in the FER schema of Figure A.1 has a corresponding mapping specification. A mapping compiler can process these specifications into a form that is integrated with the site-independent semantic functions of the TransFER attribute grammar (described in Appendix B). Together, the site-specific mappings and the site-independent mappings constitute a complete TransFER attribute grammar.

```
/* Built-in entity types; they require no mapping specifications */

defBuiltInEntity(name('Entity')).    % the "root" entity type
defBuiltInEntity(name('String')).
defBuiltInEntity(name('Datetime')).
```

```
/* Demographics */

defEntity(name('Person'),
          era(union(proj(['Id','Type'],
                         rename(['mr_number' => 'Id'],
                                 extend(['Type' == const('"PT"')],
                                        rel('Patient')))),
                    proj(['Id','Type'],
                         rename(['dea_number' => 'Id'],
                                 extend(['Type' == const('"MD"')],
                                        rel('Physician')))))),
          id(['Id','Type'])).

defRelship(name('Person-Identifier'),
           era(join(['Id' = 'IdPart',
                     'Type' = 'TypePart'],
                  vrel('Identifier-IdParts'),
                  union(proj(['Id','Type'],
                          rename(['mr_number' => 'Id'],
                              extend(['Type' == const('"PT"')],
                                     rel('Patient')))),
                        proj(['Id','Type'],
                          rename(['dea_number' => 'Id'],
                              extend(['Type' == const('"MD"')],
                                     rel('Physician')))))),
          id(['Identifier']),
          fids( [fid('Person',['Id','Type']),
                 fid('String',['Identifier'])] )).



defEntity(name('Physician'),
          era(rename(['dea_number' => 'Id'],
                     extend(['Type' == const('"MD"')],
                            rel('Physician')))),
          id(['Id','Type'])).



defEntity(name('Patient'),
          era(rename(['mr_number' => 'Id'],
                     extend(['Type' == const('"PT"')],
                            rel('Patient')))),
          id(['Id','Type'])).
```

```
defRelship(name('MD-Patient'),
           era(extend(['MD_Type' == const('"MD"'),
                        'PT_Type' == const('"PT"'), ],
                       rel('Admission'))),
           id(['dea_number','MD_Type','mr_number',
               'PT_Type','admit_datetime']),
           fids( [fid('Physician',['dea_number','MD_Type'])
                  fid('Patient',['mr_number','PT_Type']),   ] ))..

/* Medications */
defEntity(name('Medications'),
          era(rel('Therapy')),
          id( ['mr_number','med_code','start_datetime'] )).

defEntity(name('Antibiotics'),
          era(sel( ['anc_ent_type' = const('"Antibiotics"')],
                   join([ 'desc_ent_type' = 'ent_type' ],
                      rename(['ent_type' => 'anc_ent_type'],
                             rel('FER_TYPE_HIERARCHY')),
                      join([ 'med_code' = 'med_code' ],
                           rel('MEDICATION_CODES'),
                           rel('Therapy'))))),
          id( ['mr_number','med_code','start_datetime'] )).

defEntity(name('Analgesics'),
          era(sel( ['anc_ent_type' = const('"Analgesics"')],
                   join([ 'desc_ent_type' = 'ent_type' ],
                      rename(['ent_type' => 'anc_ent_type'],
                             rel('FER_TYPE_HIERARCHY')),
                      join([ 'med_code' = 'med_code' ],
                           rel('MEDICATION_CODES'),
                           rel('Therapy'))))),
          id( ['mr_number','med_code','start_datetime'] )).

defEntity(name('PCN'),
          era(sel( ['anc_ent_type' = const('"PCN"')],
                   join([ 'desc_ent_type' = 'ent_type' ],
                      rename(['ent_type' => 'anc_ent_type'],
                             rel('FER_TYPE_HIERARCHY')),
                      join([ 'med_code' = 'med_code' ],
                           rel('MEDICATION_CODES'),
                           rel('Therapy'))))),
          id( ['mr_number','med_code','start_datetime'] )).
```

```
defEntity(name('Ceph'),
          era(sel( ['anc_ent_type' = const('"Ceph"')],
                   join([ 'desc_ent_type' = 'ent_type' ],
                      rename(['ent_type' => 'anc_ent_type'],
                             rel('FER_TYPE_HIERARCHY')),
                      join([ 'med_code' = 'med_code' ],
                           rel('MEDICATION_CODES'),
                           rel('Therapy'))))),
          id( ['mr_number','med_code','start_datetime'] )).

defEntity(name('OTC'),
          era(sel( ['anc_ent_type' = const('"OTC"')],
                   join([ 'desc_ent_type' = 'ent_type' ],
                      rename(['ent_type' => 'anc_ent_type'],
                             rel('FER_TYPE_HIERARCHY')),
                      join([ 'med_code' = 'med_code' ],
                           rel('MEDICATION_CODES'),
                           rel('Therapy'))))),
          id( ['mr_number','med_code','start_datetime'] )).

defEntity(name('Narcotics'),
          era(sel( ['anc_ent_type' = const('"Narcotics"')],
                   join([ 'desc_ent_type' = 'ent_type' ],
                      rename(['ent_type' => 'anc_ent_type'],
                             rel('FER_TYPE_HIERARCHY')),
                      join([ 'med_code' = 'med_code' ],
                           rel('MEDICATION_CODES'),
                           rel('Therapy'))))),
          id( ['mr_number','med_code','start_datetime'] )).


defRelship(name('Medication-Name'),
           era(join([ 'ent_type2' = 'ent_type' ],
                  rename(['ent_type' => 'ent_type2'],
                        rel('ENTITY_TYPE_NAMES')),
                  join(['desc_ent_type' = 'leaf_ent_type'],
                     rel('FER_TYPE_HIERARCHY'),
                         join([ 'med_code' = 'med_code' ],
                              rename(['ent_type' => 'leaf_ent_type'],
                                     rel('MEDICATION_CODES')),
                              rel('Therapy'))))),
           id( ['mr_number','med_code',
                'start_datetime','ent_type_name'] ),
```

```
          fids( [ fid('Medications', ['mr_number', 'med_code',
                                      'start_datetime']),
                  fid('String', ['ent_type_name']) ] )).

defRelship(name('Patient-Medication'),
           era(extend(['Type' == const('"PT"')],
                  rel('Therapy'))),
           id( ['mr_number','med_code','start_datetime'] ),
           fids( [ fid('Patient', ['mr_number','Type']),
                   fid('Medications',['mr_number','med_code',
                                      'start_datetime']) ] )).

defRelship(name('Med-Start_Time'),
           era(rel('Therapy')),
           id( ['mr_number','med_code','start_datetime'] ),
           fids( [ fid('Medications', ['mr_number','med_code',
                                       'start_datetime'] ),
                   fid('Datetime',['start_datetime']) ] )).

defRelship(name('Med-Stop_Time'),
           era(rel('Therapy')),
           id(['mr_number','med_code','start_datetime','end_datetime']),
           fids( [ fid('Medications', ['mr_number','med_code',
                                       'start_datetime'] ),
                   fid('Datetime',['end_datetime']) ] )).

/* VIRTUAL RELATIONS */

defVirtual(name('Identifier-IdParts'),
           attrs( [ attrdef(['Identifier'],
                        [fxn(ext(concat_id),['Id','Type'])]),
                    attrdef(['Id','Type'],
                        [undefinedFxn(['Identifier-IdParts'],
                                      ['Id','Type'],
                                      ['Identifier']) ]) ] )).
```

Note: The virtual relation "Identifier-IdParts" applies the (one-directional) function "concat_id," which concatenates the pair of string values "Id" and "Type" into the single string "Identifier." The function is required to generate a single string that uniquely identifies each person, as is represented in the FER schema.

A.3 The Power of the ERA Mapping Language

The objective of the TransFER methodology is to allow applications to access heterogeneous relational databases through a uniform conceptual data model. The ERA mapping language, therefore, is designed to transform a wide variety of relational database implementations to a single canonical representation—the FER global schema. The mapping language derives power to accommodate widely varying relational implementations through (1) its use of extended relational algebra operators that can exploit the full capabilities of SQL and (2) its ability to apply arbitrarily complex external functions that execute in the host environment.

Several database researchers have characterized the forms of heterogeneity that may exist among relational databases containing similar information [Agarwal 1991, Deen, et al. 1987, Kim and Jungyun 1991]. These forms of heterogeneity may be generally classified into structural differences, naming differences, semantic differences, and content differences. In this section, I analyze how the ERA mapping language handles each form of heterogeneity. Although the analysis is informal and based largely on examples, it serves to characterize the capabilities and the limitations of the ERA mapping language.

A.3.1 Structural Differences

Structural differences entail different decompositions of semantically similar objects into the tables and fields of a relational schema. The flexibility of the relational model allows database designers to model the same semantic object in a variety of relational representations, depending on local performance requirements, local data constraints, and local normalization preferences [Batini, et al. 1992]. At least three kinds of structural differences may exist among relational databases that represent similar information: alternative table decompositions, varying data and meta-data representations, and differing structured and free-text encodings.

A.3.1.1 Alternative table decompositions

This form of structural heterogeneity has two subtypes: alternative *horizontal* and *vertical* table decompositions. Alternative horizontal decompositions entail different distributions of the same fields among one or more tables. Fields may be distributed across multiple tables to normalize a database or fields may be aggregated into a single table to improve performance [Batini, et al. 1992]. For example, Figure A.2 shows how the fields of the table `PATIENT` in Database B may be distributed into the tables `PATIENT` and `PHYSICIAN` in Database A to achieve third normal form. The ERA mapping language easily can convert from Database A to Database B by applying the JOIN operator to the tables `PATIENT` and `PHYSICIAN` (joining on the field `MD_SSN)`, and the mapping language can convert from Database B to Database A by applying the PROJECT operator twice to the table `PATIENT`.

Database A

PATIENT

MRN	LName	DOB	MD_SSN
147564773	Harmonic	12-2-57	249633805
216426077	Flynn	3-25-62	249633805

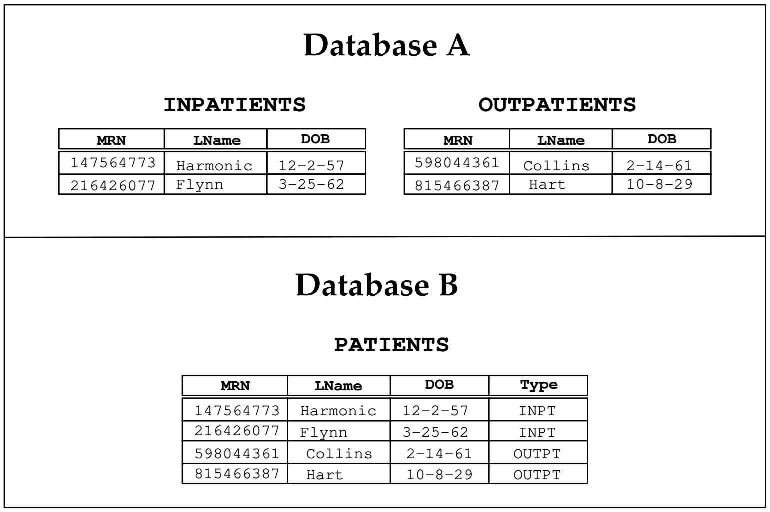
PHYSICIAN

MD_SSN	MD_LName	MD_DOB
249633805	Mason	11-17-58

Database B

PATIENT

MRN	LName	DOB	MD_SSN	MD_LName	MD_DOB
147564773	Harmonic	12-2-57	249633805	Mason	11-17-58
216426077	Flynn	3-25-62	249633805	Mason	11-17-58



Database A

INPATIENTS

MRN	LName	DOB
147564773	Harmonic	12-2-57
216426077	Flynn	3-25-62

OUTPATIENTS

MRN	LName	DOB
598044361	Collins	2-14-61
815466387	Hart	10-8-29

Database B

PATIENTS

MRN	LName	DOB	Type
147564773	Harmonic	12-2-57	INPT
216426077	Flynn	3-25-62	INPT
598044361	Collins	2-14-61	OUTPT
815466387	Hart	10-8-29	OUTPT

Database A

SODIUM_TESTS

MRN	Date	Value
147564773	10-08-95 0800	137.2
216426077	11-14-95 2200	144.8

POTASSIUM_TESTS

MRN	Date	Value
147564773	10-08-95 0800	4.3
216426077	11-14-95 2200	3.8

Database B

LAB_TESTS

MRN	Date	Na_Value	K_Value
147564773	10-08-95 0800	137.2	4.3
216426077	11-14-95 2200	144.8	3.8

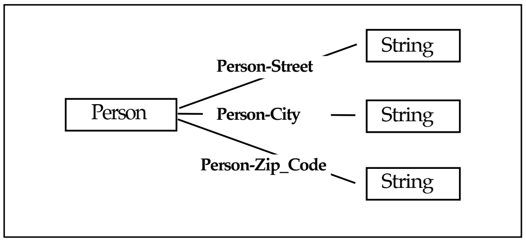
Database C

LAB_TESTS

MRN	Type	Date	Value
147564773	Sodium	10-08-95 0800	137.2
216426077	Sodium	11-14-95 2200	144.8
147564773	Potassium	10-08-95 0800	4.3
216426077	Potassium	11-14-95 2200	3.8

A.3.1.3 Structured versus free-text encodings

An entity or a relationship in an FER schema may be modeled in a corresponding relational database as either a *structured atomic value* or as a component of a *complex free-text value*. For example, the following FER schema specifies that a Person entity is associated with a street, a city, and a zip code:



"extract_street," "extract_city," or "extract_zip," which accepts a text string and returns the value of the desired address component. An ERA mapping expression invokes the external function by specifying a join between the database table `CONTACT_INFORMATION` and a virtual table that references the external function (see Section 4.1.3.6 for a discussion of virtual tables). For example, the mapping expression for the relationship type Person-Street is:

Person-Street::= ERA = "Join([`Complete_Address` = `input_address`],
`CONTACT_INFORMATION`,
`ADDRESS-TO-STREET`)"
Fid = { Person([`Soc_Sec_Number`]),
String([`street`]) }.

The definition of the virtual table `ADDRESS-TO-STREET` is:

`ADDRESS-TO-STREET =`	Field	Functional definition
	`street`:	extract_street(`input_address`)
	`input_address`:	<no function>.

By allowing arbitrary external functions in mapping expressions, the ERA mapping languages significantly extends the power of the relational algebra to resolve the representation of entities and relationships within complex text strings.

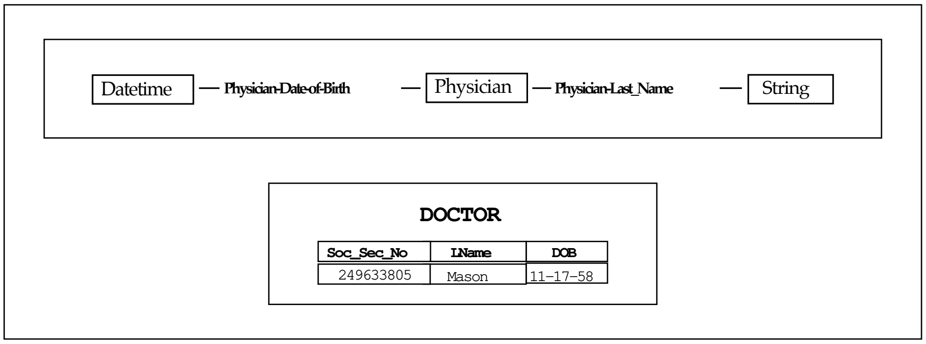
Although the address example can be handled by a simple parser, relational databases, in general, may represent clinical information in arbitrarily complex and unstructured forms that are very difficult to process automatically. For example, a relational database could store information regarding a patient's past medical problems and surgical procedures as a free-text admission note consisting of arbitrary natural-language sentences, phrases, and abbreviations. Although researchers have achieved some success in applying natural-language-processing (NLP) techniques to extract structured data from unstructured clinical documents [Sager, et al. 1994], no automated technique to process natural-language medical documents is widely available yet. It is important to note that, while this limitation constrains the set of relational implementations that the ERA mapping language can accommodate, no competing method for sharing database-intensive applications (including manual query-translation) can reliably extract clinical data that are represented as unstructured text strings.

A.3.2 Naming Differences

Naming differences are characterized by distinct lexical terms denoting the same semantic object in an FER schema and a relational database. Naming differences may be manifested as meta-data differences (that is, as variations in the *names* of relational tables and fields) or as data differences (that is, as variations in the *values* of relational fields).

A.3.2.1 Meta-data differences

Figure A.6 shows an FER schema and a relational schema that represent the same information content, specifically a set of physicians and their last names and dates of birth. The names of the tables and fields in the relational schema, however, are completely different than the names of the entity types and relationship types in the FER schema.



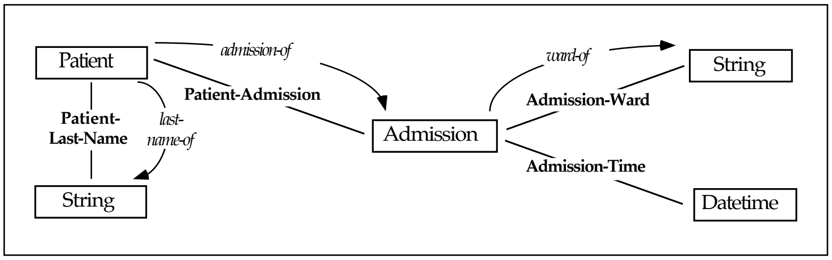
Soc_Sec_No	LName	DOB
249633805	Mason	11-17-58

Physician-Last_Name::= ERA = `DOCTOR`
Id = [`Soc_Sec_Number`, `LName`]
Fid = { Physician([`Soc_Sec_Number`]),
String([`LName`]) }

If the table or the fields in the relational schema of Figure A.6 were renamed, the new names simply would be substituted into the appropriate slots in the mapping expressions. In this case, the absence of structural, semantic, or content differences between the FER schema and the relational schema make the mapping straightforward.

A.3.2.2 Data differences

Data differences are characterized by disparities between the symbols used to denote *instances* in an FER schema and the symbols used to denote the same instances in the fields of a relational database. For example, the FER schema in Figure A.7 specifies that each patient's admission to the hospital is associated with a hospital ward via the relationship type Admission-Ward.



ADMISSIONS

MRN	LName	Ward	Admit_date
147564773	Harmonic	Med	12-2-57
216426077	Flynn	Med	3-25-62
598044361	Collins	Peds	2-14-61
815466387	Hart	ICU	10-8-29

The SQL engine will perform the join and resolve the naming conflict at the time the translated query is executed. Because there is a one-to-one correspondence between terms denoting identical concepts, the disparity between the FER schema and the relational schema is simply a naming difference. If the correspondence between terms is not one-to-one (one-to-many, for example), a *semantic* difference exists. Semantic differences are more difficult to resolve, as discussed in the following section.

A.3.3 Semantic Differences

Semantic differences occur when the meaning of an entity type or a relationship type in an FER schema is similar to but not precisely equivalent to that of a corresponding data structure in a relational database. At least three kinds of semantic differences may exist between FER schemas and relational schemas: differences in value sets, differences in level of abstraction, and differences in relationship cardinality.

A.3.3.1 Differences in value sets

Differences in value sets exist when there is not a one-to-one correspondence between the terms that may appear in the fields of a relational database and the terms that are designated as instances of an entity type in the FER schema. This disparity occurs because there is not a one-to-one correspondence between the *concepts* that the terms denote. For example, an FER schema may specify that values for the entity type Blood_Culture_Growth are represented qualitatively as "no growth," "moderate growth," and "significant growth," whereas a clinical database contains in the field Blood_Culture_Growth the values 0, 1+, 2+, 3+, or 4+. A semantic difference exists between the two value sets because the concept of moderate bacterial growth cannot be represented by any unique term in the value set {0, 1+, 2+, 3+, 4+}. In this case, a reasonable many-to-one mapping exists between the relational field values and the FER entity-type instances:

0 → "no growth"
1+ or 2+ → "moderate growth"
3+ or 4+ → "significant growth"

Because each element of the relational value set {0, 1+, 2+, 3+, 4+} maps to a unique element of the FER value set {"no growth," "moderate growth," "significant growth"}, one can map the relational schema to the FER schema by specifying a relational JOIN to a simple mapping table, as in the example of Section A.3.2.2. In this case, the necessary mapping table would be

GROWTH_MAPPING_TABLE

local_growth_code	standard_growth_code
0	no growth
1+	moderate growth
2+	moderate growth
3+	significant growth
4+	significant growth

In general, however, the correspondence between the value sets of a relational schema and an FER schema may be one to many or many to many, rather than many to one. In these cases, there are no unique mappings from elements of the relational value set to elements of the FER value set. For example, a one to many correspondence would exist if the relational value set were {"no growth," "moderate growth," "significant growth"} and the FER value set were {0, 1+, 2+, 3+, 4+}, that is, the inverse of the values above. This case would necessitate mapping from a single value to a set of values, which the mapping language does not explicitly support:

"no growth" → 0
"moderate growth" → {1+, 2+}
"significant growth" → {3+, 4+}

Theoretically, it would be useful to extend the ERA mapping language to represent one-to-many mappings if the TransFER query-translation mechanism were sufficiently sophisticated to infer that a ReFER query requesting culture results with a value of 1+ or 2+ was semantically equivalent to a relational query requesting culture results with a value of "moderate growth." Such queries would be rare, however.

A many-to-many (or overlapping) correspondence between value sets presents an even more difficult mapping problem. For example, a many-to-many correspondence would exist if the relational value set were {"light or no growth," "moderate or significant growth"} and the FER value set were {"no growth," "light or moderate growth," "significant growth"}. In this case, one could not accurately map the terms "light or no growth" and "moderate or significant growth" to even a subset of the FER value set because the meanings of the terms in the value sets overlap. For example, none of the following mappings are semantically accurate:

"light or no growth" → "no growth"
"light or no growth" → "light or moderate growth"
"light or no growth" → {"no growth," "light or moderate growth"}

One-to-many and many-to-many correspondences between value sets present difficult mapping challenges because either no correct mapping exists or there is loss of information when the mapping is performed. Although the ERA mapping language is unable to map certain correspondences between value sets, no competing method for sharing database-intensive applications (including manual query translation) can generate such information either.

A.3.3.2 Differences in level of abstraction

An FER schema may represent information in more or less detail than the relational database to which the schema is mapped. For example, the FER schema may include an entity type that represents the total number of previous pregnancies of an obstetrics patient, whereas the relational database may list and describe each previous pregnancy in detail (containing information about the duration of the pregnancy, method of delivery, etc.). In this case, the FER schema represents information regarding previous pregnancies at a higher level of abstraction than the relational database. To map from the more detailed to the less detailed representation, an ERA operation must *abstract* the detailed list of previous pregnancies in the relational database into a single numerical count of previous pregnancies. The AGGREGATE operator of the ERA can perform this abstraction by invoking the SQL aggregation function "COUNT." In general, the AGGREGATE operator can invoke any SQL or external aggregation function to abstract multiple rows of a relational table into a single row or value; this capability gives the ERA language the power to map from more detailed to less detailed representations.

Conversely, mapping from less detailed to more detailed representations is often impossible because necessary information may not be available. For example, if a relational database stores only the total number of previous pregnancies, but the FER schema represents detailed information about each previous pregnancy (such as its duration), there is no operator that can generate the detailed information from the abstract information. In this case, the mapping fails and any applications that request detailed pregnancy information via the FER schema must be notified that the information is not available in the database (see the discussion of missing information in Section A.3.4.3). Although the ERA mapping language is unable to generate more detailed information than is represented in a relational database, no competing method for sharing database-intensive applications (including manual query translation) can generate such information either.

A.3.3.3 Differences in relationship cardinality

The nature of relationships between object classes in conceptual database schemas may be one to one, one to many, or many to many [Batini, et al. 1992]. This characteristics of relationships is termed *cardinality*. The cardinality of a relationship type represented in an FER schema may differ from the cardinality of a corresponding association represented in a relational database. The ERA language must resolve this semantic difference so that the meaning of an FER schema is faithfully preserved when it is mapped to a relational schema.

For example, an FER schema may specify that each patient is associated with only one physician (the patient's primary physician), whereas a relational database may associate a patient with all of the physicians who are caring for her (whether the physicians are primary caregivers or consulting specialists). In this case, the FER schema specifies a many-to-one relationship between patients and physicians (because many patients may be associated with a physician, but only one physician may be associated with each patient), whereas the relational database allows a many-to-many relationship (because many patients may be associated with a physician and *vice-versa*). To preserve the many-to-one semantics of the FER schema, a mapping operation must designate one of the physicians associated with each patient as that patient's primary physician. The AGGREGATE operator of the ERA mapping language can perform this operation by invoking an external aggregation function that selects one physician from the set of physicians caring for a patient. The aggregation function must apply appropriate selection criteria to identify the primary physician, based on data available in the relational database.

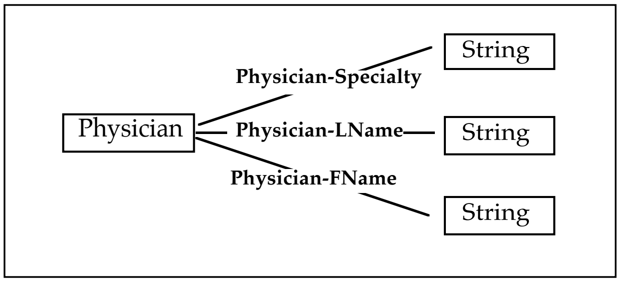
Conversely, if the FER schema specifies a many-to-many relationship between patients and physicians and the relational database represents a many-to-one relationship, no transformation is required in the mapping. This is because the relational database always will return one physician for each patient, which is a case subsumed by the many-to-many cardinality of the FER schema (that is, many to many implies many to one or more). The relational representation is, therefore, consistent with the semantics of the FER schema.

A.3.4 Content Differences

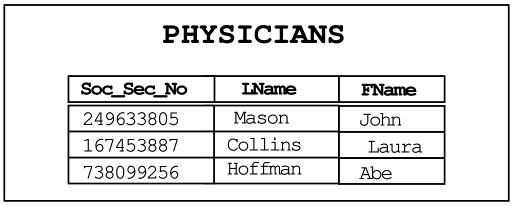
Content differences exist when information represented in the FER schema is not directly represented in a corresponding relational schema. Information that is not represented may be implicit, derivable, or simply missing.

A.3.4.1 Implicit information

Information that is obvious and not subject to change at a particular clinical site is often omitted from that site's database. The same information, however, may not be obvious or immutable at every database site, and therefore will appear in the global FER schema. For example, the global FER schema probably will represent the clinical specialties of physicians because most institutions have multispecialty medical staffs:

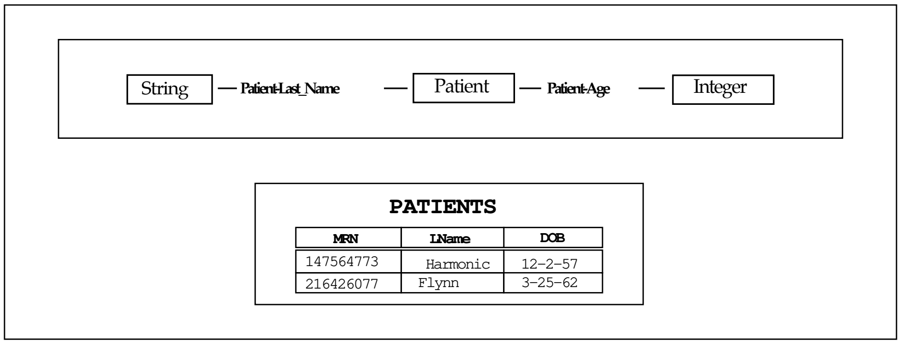


The clinical database at a psychiatric hospital, however, may not explicitly represent the specialties of its staff physicians, because the hospital employs only psychiatrists. The relational table that represents the physician roster at such a facility may be modeled as follows:



PHYSICIANS

Soc_Sec_No	LName	FName
249633805	Mason	John
167453887	Collins	Laura
738099256	Hoffman	Abe



PATIENTS

MRN	LName	DOB
147564773	Harmonic	12-2-57
216426077	Flynn	3-25-62

A.3.4.3 Missing information

Despite the power of the ERA mapping language, there may be information represented in the FER schema that simply cannot be generated from data implicitly or explicitly stored in the relational database. For example, a specific clinical facility may choose to omit patients' HIV statuses from its electronic database for purposes of confidentiality, whereas the FER schema may include HIV status as an entity type. Currently, mapping expressions specify that a NULL value be returned when missing information is requested by a ReFER query. However, this convention is inadequate, because the meaning of NULL values in databases is ambiguous [Date 1989b]. In the case of HIV status, a NULL value may signify that the status is negative, the status is unknown, or the status is known but unavailable (the appropriate meaning in this case). The TransFER methodology needs an improved mechanism to represent explicitly that information requested by ReFER queries is missing and why it is missing. Further research in this area is necessary so that application-specific processing can occur when requested data is unavailable.

A.3.5 Summary

The ERA mapping language has the power to resolve many kinds of structural, semantic, naming, and content differences between FER schemas and relational databases. The addition of operators, such as AGGREGATE and EXTEND, to the "standard" relational algebra allows the ERA mapping language to exploit the full power of modern SQL query engines to resolve differences. The availability of external functions via the virtual-relation mechanism allows the ERA mapping language to apply arbitrary transformation functions to resolve differences that the SQL language is unable to handle. Despite the power afforded by these features, there exist certain types of structural, semantic, naming, and content differences that the ERA mapping language cannot resolve. In the case of most of these differences, no other method for porting queries across database platforms, including manual query translation, can accommodate them. However, research in natural language processing, medical knowledge representation, and clinical vocabulary standardization is underway that may soon overcome the obstacles to sharing clinical decision-support applications. Because the architecture of the TransFER methodology is formally specified and theoretically sound, TransFER will be able to take advantage of new technologies when they become available.

Appendix B

The ReFER Attribute Grammar

This chapter lists the complete site-independent attribute grammar for the TransFER query-translation methodology. The implementation of the attribute grammar (augmented with the appropriate site-specific mappings) translates ReFER queries to site-specific relational queries encoded in the extended relational algebra. As I describe in Sections 4.7 and 4.8, the query translator subsequently optimizes the queries and translates them to SQL. Section B.1 contains the complete context-free grammar of the ReFER query language and serves as a reference for writing ReFER queries. Section B.2 repeats the context-free grammar rules of Section B.1, and lists the semantic functions that correspond to each grammar rule and that define the ReFER attribute grammar.

B.1 The ReFER context-free grammar

The context free grammar is predominantly a syntactic description of valid ReFER queries. However, several important semantic constraints also are noted where relevant.

```
<Query> ::= <Term>    % Note:  <Term> must have no free variables for
                               <Query> to be a a valid query

<Term> ::= [ <Termlist> | <Range> ]

<Term> ::= <Factor_Expr>
```

<Termlist$_1$> ::= <Term> , <Termlist$_2$>

<Termlist> ::= <Term>

<Range> ::= <Decllist> AND <Predlist>

<Range> ::= <Decllist>

<Factor_Expr$_1$> ::= <Factor> + <Factor_Expr$_2$>

<Factor_Expr$_1$> ::= <Factor> - <Factor_Expr$_2$>

<Factor_Expr$_1$> ::= <Factor> * <Factor_Expr$_2$>

<Factor_Expr$_1$> ::= <Factor> / <Factor_Expr$_2$>

<Factor_Expr> ::= <Factor>

<Factor> ::= <Variable>

<Factor> ::= <Constant>

<Factor> ::= <AggFunction> (<Term>)

<Factor> ::= (<Factor_Expr>)

<Factor$_1$> ::= <Function> (<Factor$_2$>)

<Factor$_1$> ::= <Variable> ^^ <Function> (<Factor$_2$>)

<Variable> ::= *string* % E.g. pt, md, drug1, drug2

<Constant> ::= *quoted_string* % E.g. "Jones"

<Constant> ::= *integer* % E.g. 145

<Constant> ::= *real* % E.g. 5.5

<Constant> ::= *quoted_datetime* % E.g. "93/12/02 0800"

<AggFunction> ::= *aggregate_function* % One of {AVG, MIN, MAX, SUM, CNT}

<Function> ::= *FER_function* % FER_function must exist in FER schema

<Decllist$_1$> ::= <Decl> , <Decllist$_2$>

<Decllist> ::= <Decl>

<Decl> ::= (<Variable> : <Entity>)

<Entity> ::= *FER_entity_type* % Entity type must exist in the FER schema

<Predlist$_1$> ::= <Predicate> AND <Predlist$_2$>

<Predlist$_1$> ::= <Predicate> OR <Predlist$_2$

<Predlist> ::= <Predicate>

<Predicate> ::= (<Predlist)

<Predicate$_1$> ::= NOT <Predicate$_2$>

<Predicate> ::= EXISTS [<Range>]

<Predicate> ::= <Factor_Expr$_1$> Operator <Factor_Expr$_2$>

<Predicate> ::= <Relationship> (<Factor$_1$> , <Factor$_2$>)

<Predicate> ::= <Variable> ^^ <Relationship> (<Factor$_1$> , <Factor$_2$>)

<Operator> ::= *operator_string* % One of { =, <>, >, ≥, <, ≤ }

<Relationship> ::= *FER_relationship* % Relationship must exist in FER schema

B.2 The ReFER attribute grammar

Each context-free grammar rule is associated with a set of semantic functions that define the attribute values of certain symbols in the grammar rule. In the semantic function definitions, each attribute of a symbol follows the symbol and is delimited by a period. For example, the term "<Predicate>.ERA" denotes the value of the attribute "ERA" for the symbol "<Predicate>". In certain cases, the grammar rules in the following attribute grammar do not match exactly the grammar rules presented in Section 4.6 of the text, because the latter rules were simplified for expository purposes. In the semantic functions that define ERA attributes, extended relational algebra operators are boldfaced for clarity.

```
<Query> ::= <Term>

   <Query>.ERA = MProj([<Term>.Id],<Term>.ERA)
   <Query>.Id = <Term>.Id
   <Query>.Vid = <Term>.Vid
   <Query>.Fid = <Term>.Fid


<Term> ::= [ <Termlist> | <Range> ]

   <Termlist>.Symbol_Table = <Range>.Symbol_Table
   <Term>.ERA = Join([videq(<Termlist>.Vid,<Range>.Vid)],
                     <Term>.ERA, <Range>.ERA)
   <Term>.Id = <Termlist>.Id
   <Term>.Vid = vid_diff(vid_union(<Termlist>.Vid, <Range>.Vid),
                         <Range>.Declared),
   <Term>.Fid = Ø


<Term> ::= <Factor_Expr>

   <Factor_Expr>.Symbol_Table = <Term>.Symbol_Table
   <Term>.ERA = <Factor_Expr>.ERA
   <Term>.Id = <Factor_Expr>.Id
   <Term>.Vid = <Factor_Expr>.Vid
   <Term>.Fid = <Factor_Expr>.Fid


<Termlist1> ::= <Term> , <Termlist2>

   <Term>.Symbol_Table = <Termlist1>.Symbol_Table
   <Termlist2>.Symbol_Table = <Termlist1>.Symbol_Table
   <Termlist1>.ERA = Join([videq(<Term>.Vid,<Termlist2>.Vid)],
                          <Term>.ERA, <Termlist2>.ERA)
   <Termlist1>.Id = concat(<Term>.Id,<Termlist2>.Id)
   <Termlist1>.Vid = vid_union(<Term>.Vid,<Termlist2>.Vid)
   <Termlist1>.Fid = Ø
```

```
<Termlist> ::= <Term>

   <Term>.Symbol_Table = <Termlist>.Symbol_Table
   <Termlist>.ERA = <Term>.ERA
   <Termlist>.Id = <Term>.Id
   <Termlist>.Vid = <Term>.Vid
   <Termlist>.Fid = <Term>.Fid


<Range> ::= <Decllist> AND <Predlist>

   <Range>.Symbol_Table = <Decllist>.Symbol_Table
   <Predlist>.Symbol_Table = <Decllist>.Symbol_Table
   <Range>.ERA = Join([videq(<Decllist>.Vid,<Predlist>.Vid)],
                     <Decllist>.ERA, <Predlist>.ERA)
   <Range>.Id = Ø
   <Range>.Vid = vid_union(<Decllist>.Vid,<Predlist>.Vid)
   <Range>.Fid = Ø
   <Range>.Declared = <Decllist>.Declared


<Range> ::= <Decllist>

   <Range>.Symbol_Table = <Decllist>.Symbol_Table
   <Range>.ERA = <Decllist>.ERA
   <Range>.Id = <Decllist>.Id
   <Range>.Vid = <Decllist>.Vid
   <Range>.Fid = <Decllist>.Fid
   <Range>.Declared = <Decllist>.Declared


<Factor_Expr1> ::= <Factor> + <Factor_Expr2>

   <Factor>.Symbol_Table = <Factor_Expr1>.Symbol_Table
   <Factor_Expr2>.Symbol_Table = <Factor_Expr1>.Symbol_Table
   <Factor_Expr1>.ERA = Join([videq(<Factor>.Vid, <Factor_Expr2>.Vid)],
                            <Factor>.ERA, <Factor_Expr2>.ERA)
   <Factor_Expr1>.Id = <Factor>.Id + <Factor_Expr2>.Id
   <Factor_Expr1>.Vid = vid_union(<Factor>.Vid, <Factor_Expr2>.Vid)
   <Factor_Expr1>.Fid = Ø
```

```
<Factor_Expr1> ::= <Factor> - <Factor_Expr2>

   <Factor>.Symbol_Table = <Factor_Expr1>.Symbol_Table
   <Factor_Expr2>.Symbol_Table = <Factor_Expr1>.Symbol_Table
   <Factor_Expr1>.ERA = Join([videq(<Factor>.Vid, <Factor_Expr2>.Vid)],
                             <Factor>.ERA, <Factor_Expr2>.ERA)
   <Factor_Expr1>.Id = <Factor>.Id - <Factor_Expr2>.Id
   <Factor_Expr1>.Vid = vid_union(<Factor>.Vid, <Factor_Expr2>.Vid)
   <Factor_Expr1>.Fid = Ø


<Factor_Expr1> ::= <Factor> * <Factor_Expr2>

   <Factor>.Symbol_Table = <Factor_Expr1>.Symbol_Table
   <Factor_Expr2>.Symbol_Table = <Factor_Expr1>.Symbol_Table
   <Factor_Expr1>.ERA = Join([videq(<Factor>.Vid, <Factor_Expr2>.Vid)],
                             <Factor>.ERA, <Factor_Expr2>.ERA)
   <Factor_Expr1>.Id = <Factor>.Id * <Factor_Expr2>.Id
   <Factor_Expr1>.Vid = vid_union(<Factor>.Vid, <Factor_Expr2>.Vid)
   <Factor_Expr1>.Fid = Ø


<Factor_Expr1> ::= <Factor> / <Factor_Expr2>

   <Factor>.Symbol_Table = <Factor_Expr1>.Symbol_Table
   <Factor_Expr2>.Symbol_Table = <Factor_Expr1>.Symbol_Table
   <Factor_Expr1>.ERA = Join([videq(<Factor>.Vid, <Factor_Expr2>.Vid)],
                             <Factor>.ERA, <Factor_Expr2>.ERA)
   <Factor_Expr1>.Id = <Factor>.Id / <Factor_Expr2>.Id
   <Factor_Expr1>.Vid = vid_union(<Factor>.Vid, <Factor_Expr2>.Vid)
   <Factor_Expr1>.Fid = Ø


<Factor_Expr> ::= <Factor>

   <Factor>.Symbol_Table = <Factor_Expr>.Symbol_Table
   <Factor_Expr>.ERA = <Factor>.ERA
   <Factor_Expr>.Id = <Factor>.Id
   <Factor_Expr>.Vid = <Factor>.Vid
   <Factor_Expr>.Fid = <Factor>.Fid
```

```
<Factor> ::= <Variable>

   <Variable>.Symbol_Table = <Factor>.Symbol_Table
   <Factor>.ERA = <Variable>.ERA
   <Factor>.Id = <Variable>.Id
   <Factor>.Vid = <Variable>.Vid
   <Factor>.Fid = <Variable>.Fid


<Factor> ::= <Constant>

   <Factor>.ERA = constant_relation(<Constant>.Value)
   <Factor>.Id = "constant_attribute_name"
   <Factor>.Vid = Ø
   <Factor>.Fid = Ø


<Factor> ::= <AggFunction> ( <Term> )

   <Term>.Symbol_Table = <Factor>.Symbol_Table
   <Factor>.ERA = Agg([<Term>.Id,                 %function argument
                       <Term>.Vid,                %grouping attributes
                       <AggFunction>.Value,       %function name
                       NewAttrList],              %automatically generated
                       <Term>.ERA)
   <Factor>.Id = NewAttrList
   <Factor>.Vid = <Term>.Vid
   <Entity>.Fid = Ø


<Factor1> ::= <Function> ( <Factor2> )

   <Factor2>.Symbol_Table = <Factor1>.Symbol_Table
   <Factor1>.ERA = Join([idpred(<Factor2>.Id,
                                 <Function>.Fid[source], =)],
                        <Factor2>.ERA, <Function>.ERA)
   <Factor1>.Id = <Function>.Fid[target]
   <Factor1>.Vid = <Factor2>.Vid
   <Factor1>.Fid = Ø


<Factor1> ::= <Variable> ^^ <Function> ( <Factor2> )

   <Factor2>.Symbol_Table = <Factor1>.Symbol_Table
```

<Factor$_1$>.ERA = **Join([**idpred(<Factor$_2$>.Id,
<Function>.Fid[source], =)**]**,
<Factor$_2$>.ERA, <Function>.ERA**)**
<Factor$_1$>.Id = <Function>.Fid[target]
<Factor$_1$>.Vid = <Factor$_2$>.Vid
<Factor$_1$>.Fid = Ø

<Factor> ::= (<Factor_Expr>)

<Factor_Expr>.Symbol_Table = <Factor>.Symbol_Table
<Factor>.ERA = <Factor_Expr>.ERA
<Factor>.Id = <Factor_Expr>.Id
<Factor>.Vid = <Factor_Expr>.Vid
<Factor>.Fid = <Factor_Expr>.Fid

<Variable> ::= *string*
<Variable>.ERA = lookup(*string*, ERA, <Variable>.Symbol_Table)
<Variable>.Id = lookup(<*string*, Id, <Variable>.Symbol_Table)
<Variable>.Id = vid(*string*,
lookup(*string*, Id, <Variable>.Symbol_Table))
<Variable>.Fid = lookup(*string*, Fid, <Variable>.Symbol_Table)

<Constant> ::= *quoted_string*

<Constant>.value = *quoted_string*

<Constant> ::= *integer*

<Constant>.value = *integer*

<Constant> ::= *real*

<Constant>.value = *real*

<Constant> ::= *quoted_datetime*

<Constant>.value = *quoted_datetime*

```
<AggFunction> ::= aggregate_function   % one of AVG, MIN, MAX, SUM, CNT

   <AggFunction>.Value = aggregate_function



<Function> ::= FER_function

   <Function>.Relationship = lookup_FER(FER_function, Relationship)
   <Function>.Source = lookup_FER(FER_function, Source_Entity_Type)
   <Function>.Target = lookup_FER(FER_function, Target_Entity_Type)
   <Function>.ERA = lookup_mapping(<Function>.Relationship, ERA)
   <Function>.Fid[source] = lookup_mapping(<Function>.Relationship,
                                           fid[<Function>.Source])
   <Function>.Fid[target] = lookup_mapping(<Function>.Relationship,
                                           fid[<Function>.Target])


<Decllist1> ::= <Decl> , <Decllist2>

   <Decllist1>.ERA = Join([videq(<Decl>.Vid,<Decllist2>.Vid)],
                          <Decl>.ERA, <Decllist2>.ERA)
   <Decllist1>.Id = Ø
   <Decllist1>.Vid = vid_union(<Decl>.Vid, <Decllist2>.Vid)
   <Decllist1>.Fid = Ø
   <Decllist1>.Symbol_Table  = decl_union(<Decl>.Symbol_Table,
                                          <Decllist2>.Symbol_Table)
   <Decllist1>.Declared = decl_union(<Decl>.Declared,
                                     <Decllist2>.Declared)


<Decllist> ::= <Decl>

   <Decllist>.ERA = <Decl>.ERA
   <Decllist>.Id = <Decl>.Id
   <Decllist>.Vid =<Decl>.Vid
   <Decllist>.Fid = <Decl>.Fid
   <Decllist>.Symbol_Table = <Decl>.Symbol_Table
   <Decllist>.Declared = <Decl>.Declared


<Decl> ::= ( <Variable> : <Entity> )

   <Decl>.ERA = <Entity>.ERA
```

<Decl>.Id = <Entity>.Id
<Decl>.Vid = vid(<Variable>.Value, <Entity>.Id)
<Decl>.Fid = <Entity>.Fid
<Decl>.Symbol_Table = entry(<Variable>.Value,
<Entity>.ERA,
<Entity>.Id,
vid(<Variable>.Value, <Entity>.Id),
<Entity>.Fid)
<Decl>.Declared = <Variable>.Value.

<Entity> ::= *string* % Must be one of the entities in the FER schema

<Entity>.ERA = lookup_mapping(*string*, ERA)
<Entity>.Id = lookup_mapping(*string*, Id)
<Entity>.Vid = Ø
<Entity>.Fid = Ø

<Predlist$_1$> ::= <Predicate> AND <Predlist$_2$>

<Predicate>.Symbol_Table = <Predlist$_1$>.Symbol_Table
<Predlist$_2$>.Symbol_Table = <Predlist$_1$>.Symbol_Table
<Predlist$_1$>.ERA = **Join([**videq(<Predicate>.Vid,<Predlist$_2$>.Vid)**],**
<Predicate>.ERA, <Predlist$_2$>.ERA**)**
<Predlist$_1$>.Id = Ø
<Predlist$_1$>.Vid = vid_union(<Predicate>.Vid, <Predlist$_2$>.Vid)
<Predlist$_1$>.Fid = Ø

<Predlist$_1$> ::= <Predicate> OR <Predlist$_2$

<Predicate>.Symbol_Table = <Predlist$_1$>.Symbol_Table
<Predlist$_2$>.Symbol_Table = <Predlist$_1$>.Symbol_Table
<Predlist$_1$>.ERA = **Union(**<Predicate>.ERA, <Predlist$_2$>.ERA**)**
<Predlist$_1$>.Id = Ø
<Predlist$_1$>.Vid = vid_union(<Predicate>.Vid, <Predlist$_2$>.Vid)
<Predlist$_1$>.Fid = Ø

<Predlist> ::= <Predicate>

<Predicate>.Symbol_Table = <Predlist>.Symbol_Table

```
   <Predlist>.ERA = <Predicate>.ERA
   <Predlist>.Id = <Predicate>.Id
   <Predlist>.Vid = <Predicate>.Vid
   <Predlist>.Fid = <Predicate>.Fid


<Predicate> ::= ( <Predlist> )

   <Predlist>.Symbol_Table = <Predicate>.Symbol_Table
   <Predicate>.ERA = <Predlist>.ERA
   <Predicate>.Id = <Predlist>.Id
   <Predicate>.Vid = <Predlist>.Vid
   <Predicate>.Fid = <Predlist>.Fid


<Predicate$_1$> ::= NOT <Predicate$_2$>

   <Predicate$_2$>.Symbol_Table = <Predicate$_1$>.Symbol_Table
   <Predicate$_1$>.ERA = Diff([],<Predicate$_2$>.ERA) %"raw" diff form
   <Predicate$_1$>.Id = Ø
   <Predicate$_1$>.Vid = Ø %"raw" Vid form; filled in later
   <Predicate$_1$>.Fid = Ø


<Predicate> ::= EXISTS [ <Range> ]

   <Range>.Symbol_Table = <Predicate>.Symbol_Table
   <Predicate>.ERA = Exists([],<Range>.ERA)  %"raw" Exists() form
   <Predicate>.Id = Ø
   <Predicate>.Vid = vid_diff(<Range>.Vid,<Range>.Declared)
   <Predicate>.Fid = Ø


<Predicate> ::= <Factor_Expr$_1$> <Operator> <Factor_Expr$_2$>

   <Factor_Expr$_1$>.Symbol_Table = <Predicate>.Symbol_Table
   <Factor_Expr$_2$>.Symbol_Table = <Predicate>.Symbol_Table
   <Predicate>.ERA = Join([idpred(<Factor_Expr$_1$>.Id, <Factor_Expr$_2$>.Id,
                             Operator.Value)],
                     <Factor_Expr$_1$>.ERA, <Factor_Expr$_2$>.ERA)
   <Predicate>.Id = Ø
   <Predicate>.Vid = vid_union(<Factor_Expr$_1$>.Vid, <Factor_Expr$_2$>.Vid)
   <Predicate>.Fid = Ø
```

```
<Operator> ::= operator_string   % One of { =, <>, >, ≥, <, ≤ }

   Operator.Value = operator_string

<Predicate> ::= <Relationship> ( <Factor$_1$> , <Factor$_2$> )

   <Factor$_1$>.Symbol_Table = <Predicate>.Symbol_Table
   <Factor$_2$>.Symbol_Table = <Predicate>.Symbol_Table
   <Predicate>.ERA = Join([idpred(<Factor$_1$>.Id, <Relationship>.Fid[1],
                                 =)],
                          <Factor$_1$>.ERA,
                          Join([idpred(<Factor$_2$>.Id,
                                       <Relationship>.Fid[2], =)],
                               <Factor$_2$>.ERA, <Relationship>.ERA))
   <Predicate>.Id = Ø
   <Predicate>.Vid = vid_union(<Factor$_1$>.Vid, <Factor$_2$>.Vid)
   <Predicate>.Fid = Ø


<Predicate> ::= <Variable> ^^ <Relationship> ( <Factor$_1$> , <Factor$_2$> )

   <Factor$_1$>.Symbol_Table = <Predicate>.Symbol_Table
   <Factor$_2$>.Symbol_Table = <Predicate>.Symbol_Table
   <Relationship>.Symbol_Table = <Predicate>.Symbol_Table
   <Predicate>.ERA = Join([idpred(<Factor$_1$>.Id,
                                 <Relationship>.Fid[1], =)],
                          <Factor$_1$>.ERA,
                          Join([idpred(<Factor$_2$>.Id,
                                       <Relationship>.Fid[2], =)],
                               <Factor$_2$>.ERA, <Relationship>.ERA))
   <Predicate>.Id = Ø
   <Predicate>.Vid = vid_union(<Factor$_1$>.Vid, <Factor$_2$>.Vid)
   <Predicate>.Fid = Ø

<Relationship> ::= FER_relationship

   <Relationship>.ERA = lookup(FER_relationship, ERA,
                               <Relationship>.Symbol_Table)
   <Relationship>.Id = lookup(FER_relationship, Id,
                              <Relationship>.Symbol_Table)
   <Relationship>.Vid = Ø
   <Relationship>.Fid[1] = lookup_FER(FER_relationship, Fid[1])
   <Relationship>.Fid[2] = lookup_FER(FER_relationship, Fid[2])
```

Certain operators that appear in the semantic functions of the attribute grammar require explanation:

- **vid_union**: computes the union of two sets of vids, as described in Section 4.6.1 of the text.

- **vid_diff**: computes the difference between two sets of vids, such that only the vids in the first set that are not in the second set are returned. For example, if vid_1 = { (pt, [`fname, lname`]), (md,[`first, last`])} and vid_2 = { (pt, [`fname, lname`])}, then vid_diff(vid_1, vid_2) = {(md,[`first, last`])}.

- **videq**: expands a join condition consisting of a comparison between two vids, so that the attributes in the vids are compared pairwise, as described in Section 4.6.3 of the text.

- **idpred**: expands a join condition consisting of a comparison between two *sets* of identifying attributes, so that the elements of the sets are compared pairwise, as described in Section 4.6.3 of the text.

- **decl_union**: merges two symbol tables.

- **lookup**: indexes into the symbol table (third argument) by a variable name (first argument) and retrieves the requested component (second argument). For example, lookup(pt, ERA, Symbol_Table_1) returns the ERA expression defined for the entity type that was declared as the variable pt in the symbol table Symbol_Table_1.

- **lookup_FER**: indexes into the coded representation of the current FER schema and retrieves the requested information. For example, lookup(age-of, Relationship) returns the relationship type that corresponds to the relationship function "age-of."

- **lookup_mapping**: indexes into the ERA mappings defined for the current site by an entity-type or relationship-type name and retrieves the requested mapping information. For example, lookup(Person, Id) might return the set of attributes { first_name, last_name } (that is, the key attributes of the entity type Person).

In addition to the attributes ERA, Id, Vid, and Fid that I discussed in Sections 4.2 and 4.3.2 of the text, the attribute grammar may also assign the following attributes to ReFER grammar symbols:

- **Symbol_Table**: A set of declared variables and their associated ERA mappings. The mappings corresponding to variables are assigned based on the entity types for which the variables are declared. For example, if the declaration (pt : Patient) appears in a ReFER query, the symbol table includes an entry that binds the variable "pt" to the ERA mapping for the entity type "Patient."

- **Declared**: The set of variables that were explicitly declared within an expression. For example, the set of declared variables in the ReFER expression "[name-of(pt) | (pt: Patient), (md: Physician) AND Treats(md,pt)]" is { pt, md }.

Appendix C

The Evaluation Data Set

This appendix contains documents that describe the evaluation experiment. Section C.1 contains the original call for participation that was circulated to solicit volunteer subjects for the experiment. Section C.2 contains the instructions that were provided to the volunteer subjects before they began the database-design task. Section C.3 contains the complete clinical data set which the database-design teams were asked to model in a relational database schema.

C.1 Call for Participation in Experiment

The following call for participation was circulated via electronic mail to the student bodies of the medical information sciences department at Stanford Medical School and the database research group at the Stanford Computer Science Department. The database-design teams for the experiment were composed of the respondents to this message.

> Greetings,
>
> I will be conducting an experiment as part of my thesis evaluation and need a few good women and men. You are qualified to participate if you have clinical experience and/or experience in relational database design.
>
> The experiment will involve several teams of two members--one knowledgeable in the medical domain, the other in database design--developing and implementing relational database schemas to represent a clinical data set that I will provide. The data set represents

clinical data for a single patient over a two day period and is of modest scope (roughly 40 data elements) and of modest size (roughly 150 recorded values). Hence, the database design task is much smaller than that of a full-blown clinical database, though it is representative of the modeling challenges inherent in the design of a real database. Each team will design a database schema independently to represent the clinical data set, and subsequently I will apply my abstract data model and query language to map across the different databases. Each team will also encode a small set of questions that I will provide into SQL queries that can execute against its database.

I've included an abstract describing my work and its objectives at the end of this message, and I would be glad to talk to anyone who has more questions. Your reward for participating in the experiment, in addition to my everlasting gratitude and an all-expense-paid trip to Tony and Alba's pizzeria, will be the opportunity to design your own clinical database schema with a partner whose expertise in this task complements your own. MIS students with a clinical background who are considering taking CS245 this winter (Database System Implementation) may be especially interested, as this course includes a database project.

If you decide to participate, you will specifically be asked to do the following:

1. Based on a clinical data set that I provide, design a relational database schema to capture the information in the data set and to support SQL queries against the data. The data that I provide will be in a very general format and you will need to make a number of modeling decisions in designing the relational schema.

2. Implement the schema you have designed in Sybase DBMS and populate the database with the clinical data that I have provided.

3. Translate a modest number (5-10) of informal questions that I provide into formal SQL queries that will answer the questions based on your relational database. For example, a question may be "what were all of the temperature readings for the patient named Chris Webber between 12:00 and 5:00 PM." The questions will be relatively simple.

Participating in the experiment will require approximately 10-15 hours of your time over a 3 month period beginning in November. Most of your help will be needed in January, so the experiment will not conflict with final exams and the holidays. Please contact me as soon as possible if you are interested in participating. Thank you.

ABSTRACT:

Bridging the Representational Heterogeneity of Clinical Databases

The goal of our work is to overcome the heterogeneity of clinical databases so as to provide medical decision-support applications with a uniform interface to patient data. Most decision-support applications require patient-specific data because they provide advice in the context of clinical cases. These applications include alerting systems, such as the HELP system and the Regenstrief Medical Record System; diagnostic systems, such as MYCIN and QMR; and information-retrieval systems, such as MEDLINE. The emergence and expansion of on-line clinical databases provides the potential for applications to access patient-specific data automatically and thereby to streamline the decision-support process by relieving users of the data-entry burden. The benefits of this mode of interaction have been demonstrated in existing systems [Kuperman, 1991, McDonald, 1984]. However, applications that automatically access patient data currently cannot be shared among health-care institutions because the means by which clinical data are represented and retrieved vary widely. For example, the help system's decision-support software in operation at the Latter-day Saints Hospital cannot be ported to other provider sites without significant modifications because the database queries generated by the help system are specific to the database structure, the query language, and the coding scheme of the help database. Disparities in the representation and retrieval of data across clinical databases exist, although the information stored at different sites is conceptually quite similar. The diverse implementation of databases containing conceptually similar information is termed representational heterogeneity. Representational heterogeneity requires that database queries be recoded manually when decision-support applications are ported from one provider site to another. The recoding of queries is necessary even for applications that use ostensibly portable knowledge-representation languages, such as the Arden Syntax [Hripcsak, 1990], because existing languages provide no standard model of clinical data. Manual recoding of queries not only entails significant effort, but also introduces a source of error because the semantics of queries may not be preserved when the queries are reformulated in a different environment.

We are developing a methodology, called TransFER, to bridge the representational heterogeneity of clinical databases and to obviate the need for manual recoding of database queries. The general strategy of TransFER is to define a global and uniform reference model of clinical information against which applications' requests for patient data may be formulated. The reference model is abstract, and is independent of any clinical database implementation. A formal mapping language allows database administrators to specify the correspondence between the clinical database implementation at their sites and the abstract reference model. Currently, only clinical databases implemented using the relational model can be accommodated by the mapping language. A translating compiler at each site uses the encoded mappings to translate automatically queries specified against the global model to queries that can be executed locally and that are semantically equivalent.

We propose to use the ICU data set to evaluate the generality and utility of the TransFER query translation methodology. Our research hypothesis is that the TransFER methodology can map the reference model effectively to three independently designed relational database schemata, all of which represent the same data set, and that the TransFER methodology can translate automatically queries formulated against the single reference model to queries that can be executed against all three schemata. The experiment will allow us to evaluate the TransFER methodology in the context of a realistic clinical data set and with respect to independently implemented databases.

References:

Hripcsak, G., Clayton, P. D., Pryor, T. A., Haug, P., Wigertz, O. B.& Van der lei, J. (1990). The Arden Syntax for Medical Logic Modules. In R. Miller (Ed.), Proceedings of the Fourteenth Symposium on Computer Applications in Medical Care . Washington, D.C., pp. 200-204.

Kuperman, G. J., Gardner, R. M.& Pryor, T. A. (1991). HELP: A Dynamic Hospital Information System. New York: Springer-Verlag.

McDonald, C. J. (1984). Reminders to Physicians from an Introspective Computer Medical Record. Annals of Internal Medicine, 100: 130-138.

C.2 Instructions to Database-design Teams

The following document contains the instructions that I provided to the volunteer subjects regarding the database-design task. Three database-design teams participated in the experiment; two of the teams consisted of one computer-science department student and one physician, and one of the teams consisted of a single physician with considerable experience in relational-database design.

The Clinical Database Design Derby

I. General instructions

The task is to represent in a relational database the clinical and demographic information for two inpatients. The data will be provided to you in a form resembling a patient chart. The objective is to generate a relational database schema that can capture the information in the "chart" and to populate the database with this information. The Sybase DBMS will be used for the implementation. The products of the design process include:

1. A set of SQL data definition language commands that define the relational database schema. These command will be specific to Sybase's "flavor" of SQL.
2. Documentation of the codes that are used in the populated database to represent the patient data (this may be part of the database or may be external to it)
3. An implemented and populated Sybase database that can be queried using SQL

II. Designing the database

- You may use any database design technique you wish to define the relational schema (ER diagrams, object-oriented analysis, etc.).
- The final relational schema should be represented as a set of commands in the Sybase data definition language. The commands are similar to the data definition language (DDL) commands of SQL. If you need to familiarize yourself with the Sybase DDL, Sybase documentation is available at the Section on Medical Informatics and elsewhere.
- You may use any names, codes, etc. you wish to represent the table and field names of the schema, consistent with the constraints of Sybase.
- Each table must have a primary key defined, consisting of one or more fields of that table. All foreign keys also must be explicitly defined. The Sybase DDL includes constructs to do this.

III. Populating the database

- After you have finished the database design, the laboratory and flowsheet data will be provided in electronic form. You may either write a small program to load the data into your database, write a program to generate the SQL commands to load the data, or manually write the SQL commands to load the data (or interactively load the data! -- not recommended).

- The history and physical, demographics, and medication list are not available in electronic form. However, this set of data is small.

- You may use any names, codes, etc. you wish to represent the values appearing in the database (except for numerical values, which must appear in their original integer or real form). These codes may be taken from standard terminologies, such as ICD9, CPT4, and SNOMED, or they may be of your own invention. For each name or code whose meaning is not obvious, you must specify either the standard vocabulary from which the code was taken or a more descriptive textual definition of the code.

IV. Rules of the game

1. The final populated relational schema must represent all of the data provided (i.e., *at least* all of the data — it may represent more data as specified below).

2. You may *enrich* the data set by adding types of objects that are not currently represented, as well as instances of those types. However, you may not *expand* the data set by adding instances of any object types that already are represented. For example, you could add data about each patient's insurance coverage because this type of information is not currently represented, but you could not add any chemistry lab results to the set of results already in the data. When in doubt, ask yourself "would adding these data change the results of any valid query issued against the original data set?" If the answer is yes, don't add them.

3. The objective is for the schemas to be designed independently. Therefore, you may not consult with any members of the other database design groups or with any of the physicians who will be formulating queries against the databases. The members of these groups include:

   <list of participants omitted>

V. The Data

The following documents will be provided to you.

1. Patient descriptions (demographic and encounter data)

2. Physician descriptions (demographic data)
3. Lab results (the results of chemistry and hematology lab tests)
4. ICU flowsheet data in graphical form (charted physiological parameters captured in the ICU)

If you have any questions at all about the data or about the database-design task in general, please contact me.

Thanks for your participation!

C.3 The Clinical Data Set

The clinical data set consists of demographic, laboratory, medication, and ICU-monitoring data for two patients over the course of approximately 24 hours in the hospital. The data are subdivided into a demographic and history section, a laboratory-test result section, and a charted flowsheet section. In addition to the patient data, there are also physician data that describe the patients' caregivers. The data for Patient #1 were provided by Isaac Kohane and Serdar Uckun as a test-data set to all participants in the 1993 AAAI Artificial Intelligence in Medicine conference.

C.3.1 Patient #1: Jessica Hepburn

Name: Jessica Hepburn
1229 Industrial Ave
Fremont, CA 94520
(415) 320-9934
Medical Record #: 238-94-4227
Attending: Dr. Bill Fold
Admit Date: 8/16/93 17:00

History and Physical

8.5 month old **asian female (date of birth: 12/28/92)** with **biliary atresia,** S/P (surgical procedure) **Kasai procedure on 5/22/93**, **ascending cholangitis** with a history of multiple liver abscesses and portal hypertension.

She was transferred to The Children's Hospital after a **15 day history of E. Coli sepsis**, worsening despite antibiotic therapy. She **currently is in liver failure with coagulopathy**. She has required **mechanical ventilatory support** due to the **ARDS** that followed the onset of sepsis.

Her **admission body weight is 5 kg**.

In addition to medications listed in the ICU flowsheet, the patient receives:

Oral nystatin
Ativan 0.5 mg IV every 6 hours
Amphotericin bladder irrigation every 12 hours
Nebulized ventolin PRN
Maalox 10cc by nasogastric tube every 6 hours for gastric pH < 5

LABS
08/16/93 18:13

Prothrombin Time	19.9 sec	Prothrombin Time - Control	12.0 sec
PTT	68.6 sec	PTT - Control	20 sec
Plasma Fibrinogen	100 mg/dl		

RBC	3.78 million/mm^3
MCV	91.9 fl
WBC	11.77 /mm^3
HCT	34.8 %
Mean Corpuscular Hemoglobin Conc	33.8 g/dl
Hemoglobin Conc	11.7 g/dl
Mean Corpuscular Hemoglobin	31.1 pg
Platelet Count	43 /mm^3

Blood Glucose	100 mg/dl
Plasma CO2	21 mEq/L
Serum CL$^-$	97 mEq/L
BUN	72 mg/dl
Serum K$^+$	4.2 mEq/L
Serum Creatinine	1.1 mEq/L
Serum NA$^+$	136 mEq/L

LABS
08/16/93 23:16

Serum Ionized Ca^{++}	0.78 mEq/L
PaO2	87 mmHg
PaCO2	44 mmHg
Arterial pH	7.28

LABS
08/17/93 04:30

Arterial O2 Saturation 96 %
Arterial Bicarb Conc 22 mEq/L
Arterial pH 7.23
PaCO2 53 mmHg
PaO2 93 mmHg
Serum Ionized Ca++ 0.90 mEq/L

LABS
08/17/93 04:31
Prothrombin Time 21.8 sec Prothrombin Time - Control 12.0 sec
PTT 76.6 sec PTT - Control 20 sec
Plasma Fibrinogen 89 mg/dl

WBC 13.72 /mm^3, RBC 3.80 million/mm^3
MCV 88.9 fl
Mean Corpuscular Hemoglobin Conc 33.4 g/dl
Hemoglobin Conc 11.3 g/dl
Mean Corpuscular Hemoglobin 29.7 pg
HCT 33.8

SGPT 33 U/L
SGOT 122 U/L
BUN 73 mg/dl
Blood Glucose 97 mg/dl
Plasma CO2 Content 20 mEq/L
Plasma Fibrinogen 89 mg/dl
Serum Albumin 3.4 g/dl
Serum Ca^{++} 9.2 mEq/L
Serum Cl^{-} 99 mEq/L
Serum Creatinine 1.0 mg/dl
Serum Inorganic Phosphate 4.9 mg/dl
Serum K^{+} 4.0 mEq/L
Serum Mg^{+} 2.4 mEq/L
Serum Na^{+} 137 mEq/L
Serum Total Protein 6.8 g/dl
Serum Triglycerdies 150 mg/dl
Total Bilirubin 29.8 mg/dl
Whole Blood Ammonia 382 mcg/dl

LABS
08/17/93 06:11
Arterial O2 Saturation 95 %
Arterial Bicarb Conc 21 mEq/L
Arterial pH 7.27
PaCO2 45 mmHg
PaO2 87 mmHg

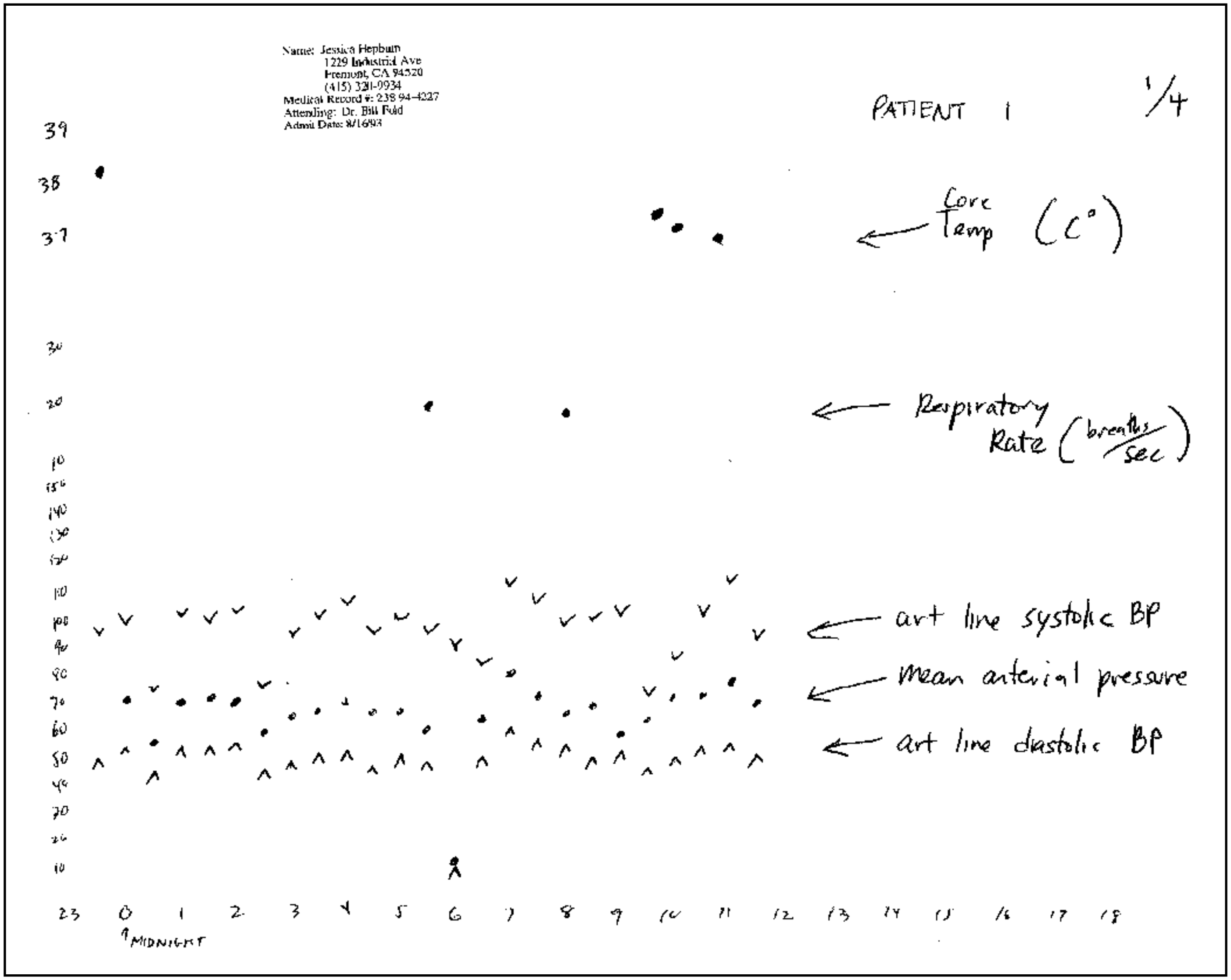
LABS
08/17/93 09:52
Arterial O2 Saturation 97 %
Arterial Bicarb Conc 20 mEq/L
Arterial pH 7.31
PaCO2 41 mmHg
PaO2 95 mmHg
Serum Ionized Ca^{++} 0.82 mEq/L

LABS
08/17/93 12:25
Arterial O2 Saturation 94 %
Arterial Bicarb Conc 21 mEq/L
Arterial pH 7.28
PaCO2 44 mmHg
PaO2 80 mmHg
Serum Ionized Ca^{++} 0.87 mEq/L

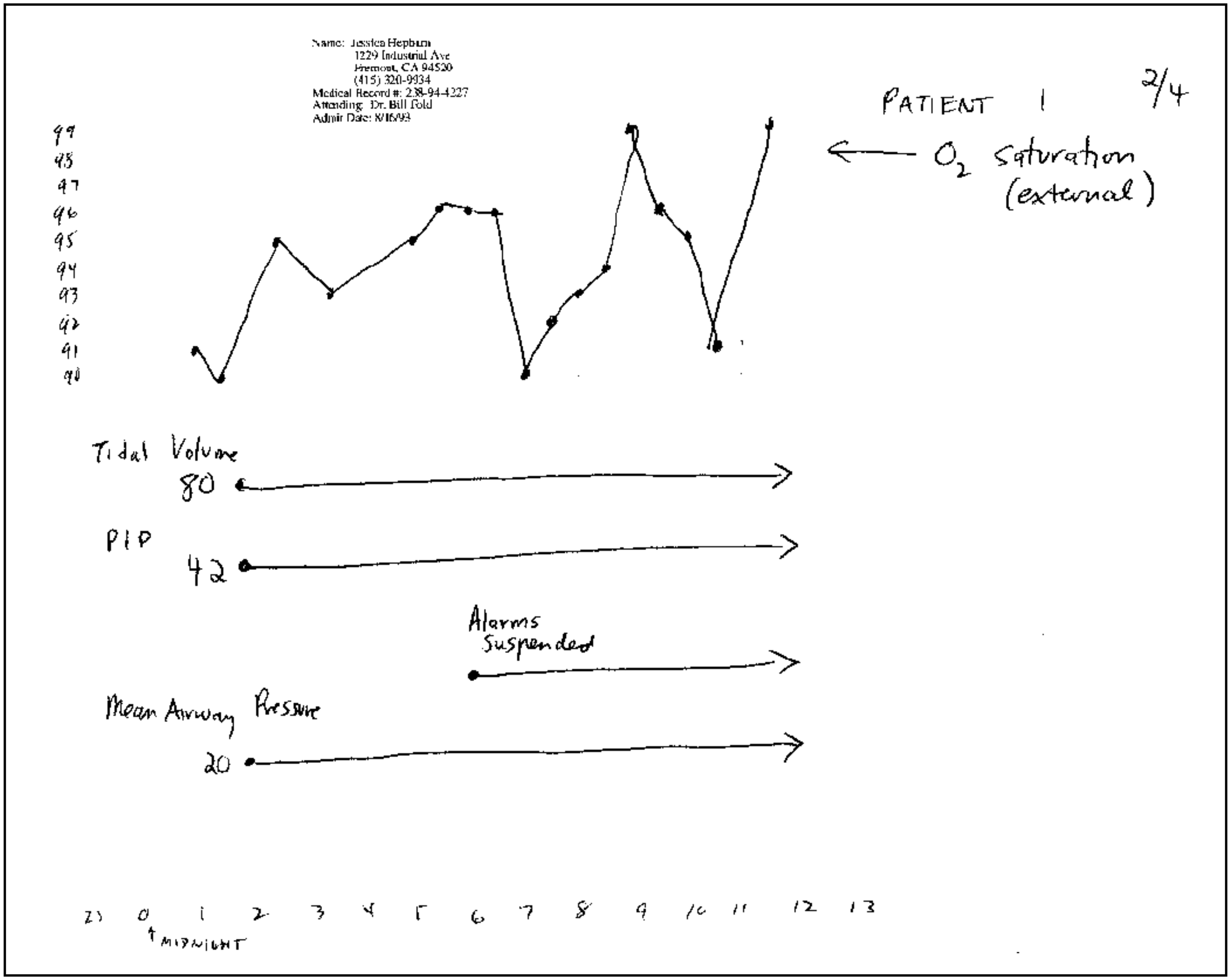
LABS
08/17/93 12:41

Platelet Count	108 /mm^3

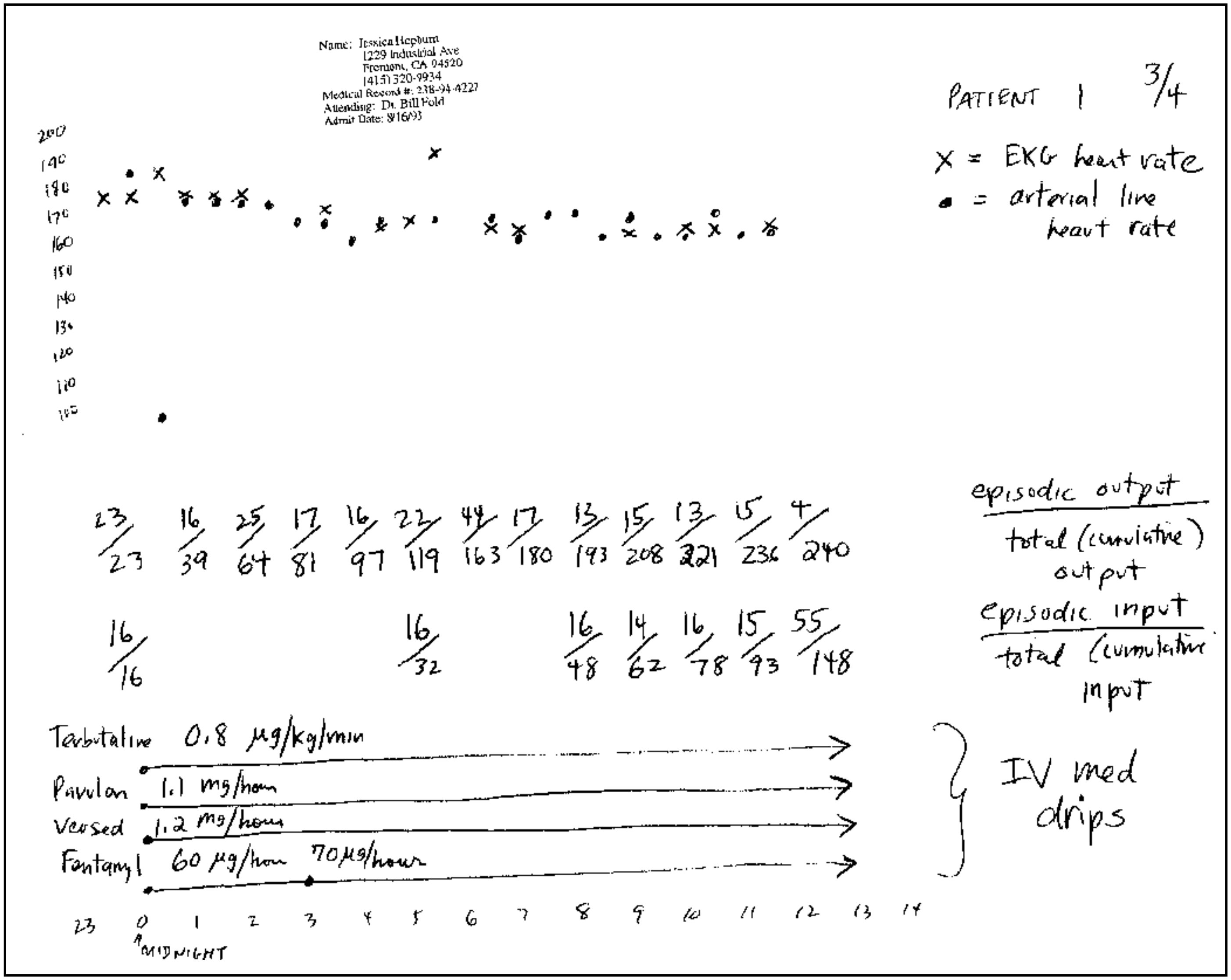
The following four pages contain flowsheet data captured during the patient's stay in the ICU. This data was provided to the database-design teams in exactly the form shown.



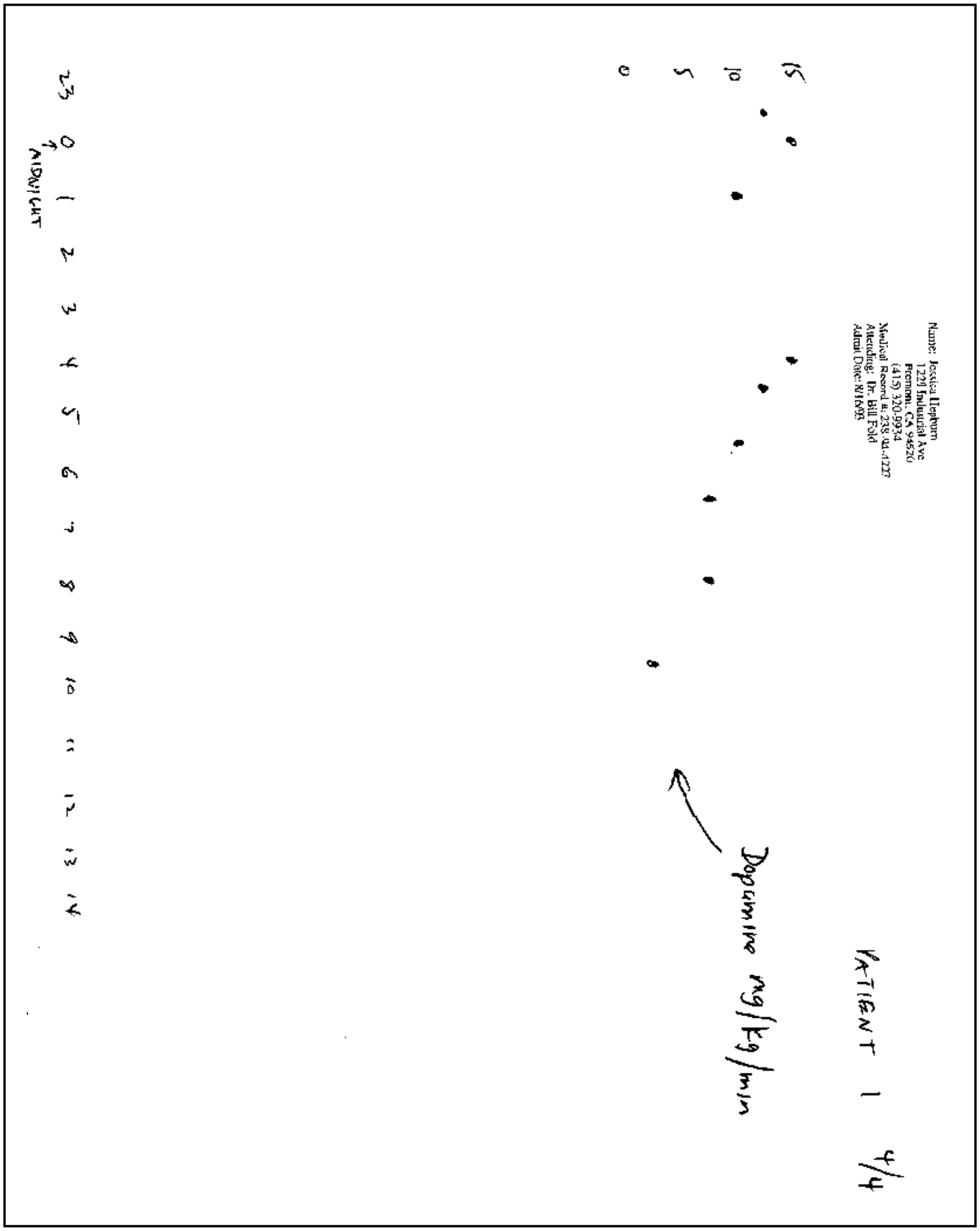
Name: Jessica Hepburn
1229 Industrial Ave
Fremont, CA 94520
(415) 321-9934
Medical Record #: 238 94-4227
Attending: Dr. Bill Field
Admit Date: 8/16/93



Name: Jessica Hepburn
1229 Industrial Ave
Fremont, CA 94530
(415) 320-9934
Medical Record #: 238-94-4227
Attending: Dr. Bill Fold
Admit Date: 8/16/93



Name: Jessica Hepburn
1229 Industrial Ave
Fremont, CA 94520
(415) 320-9934
Medical Record #: 238-94-4227
Attending: Dr. Bill Fold
Admit Date: 8/16/93


PATIENT 1 4/4
Name: Jessica Hepburn
1221 Industrial Ave
Fremont, CA 94520
(415) 320-9934
Medical Record #: 238-44-4227
Attending: Dr. Bill Fold
Admit Date: 8/16/93
Dopamine mg/Kg/min
15
10
5
0
23
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
MIDNIGHT

C.3.2 Patient #2: Anna Robik

Name: Anna Robik
415 Treeline St.
Los Altos, CA   94316
(415) 434-6733
Medical Record #: 145-78-3966
Attending: Seth O'Scope, MD
Admit Date: 5/24/93 06:00

History and Physical

Mrs. Robik is a 57 year old **caucasian woman** (**date of birth: 2/14/36**) admitted on 5/24/93 **for exacerbation of asthma**. At that time, her **respiratory rate was 36**, **pulse was 131** and **blood pressure was 150/100**.

Past history includes: **asthma since age 8**, **left nasal polyp removal 1985**, **chronic sinusitis since 1980**, **chronic renal insufficiency since 1988**, and **right renal cyst removal in 1993**.

She had a **normal renal ultrasound on 4/10/93**.

Medications on admission to hospital:

Albuterol MDI 4 puffs q 4
Theodur 100 mg po BID
Azmacort MDI 4 puffs q 4
Pulmoaid nebulizer prn .2 cc Albuterol /.2 cc NS
Brokosol prn
Cholorpheniramine prn

Allergies: Aspirin, non-steroidal antiinflammatory medications

Transfer Note

Her worsening **respiratory distress** led to a **transfer to the ICU on 5/25/93 10:00** where she remains.

ICU MEDS:

D5 NORMAL SALINE @ 100 CC/HOUR
Ranitidine 6 mg/min IV drip
Solumedrol 120 mg IV q 6 hours
Cefotetan 1 gm IV q 12 hours

Admission LABS
5/24/93 06:00 :

Serum NA^+ = 131 mEq/L, Serum K^+ = 4 mEq/L, Serum CL^-=104 mEq/L,
Plasma CO2 Content = 28 mEq/L, Serum Creatinine = 1.7 mEq/L,

Blood Glucose = 131 mg/dl

WBC=9.6 /mm^3, HCT = 45.8 %, Hemoglobin Conc 15.3 g/dl
39% Polymorphonuclear cells, 2% Band Cells, 34% Lymphocyes, 16% Monocytes, 8% Eosinophils

ABG ON 100% FACE MASK: Arterial pH = 7.26, PaCO2 = 56 mmHg,
PaO2 =232 mmHg

Notes in ICU

LABS
5/25/93 13:00
HCT =39 %, WBC = 28 /mm^3, Theophylline level = 1.8 µg/ml

ABG ON 100% FACE MASK: Arterial pH = 7.39, PaCO2 = 35 mmHg,
PaO2 =80 mmHg

LABS
5/25/93 15:00
Serum NA$^+$ = 142 mEq/L, Serum K$^+$ = 4.1 mEq/L, Serum CL$^-$=107 mEq/L,
Plasma CO2 Content = 22 mEq/L

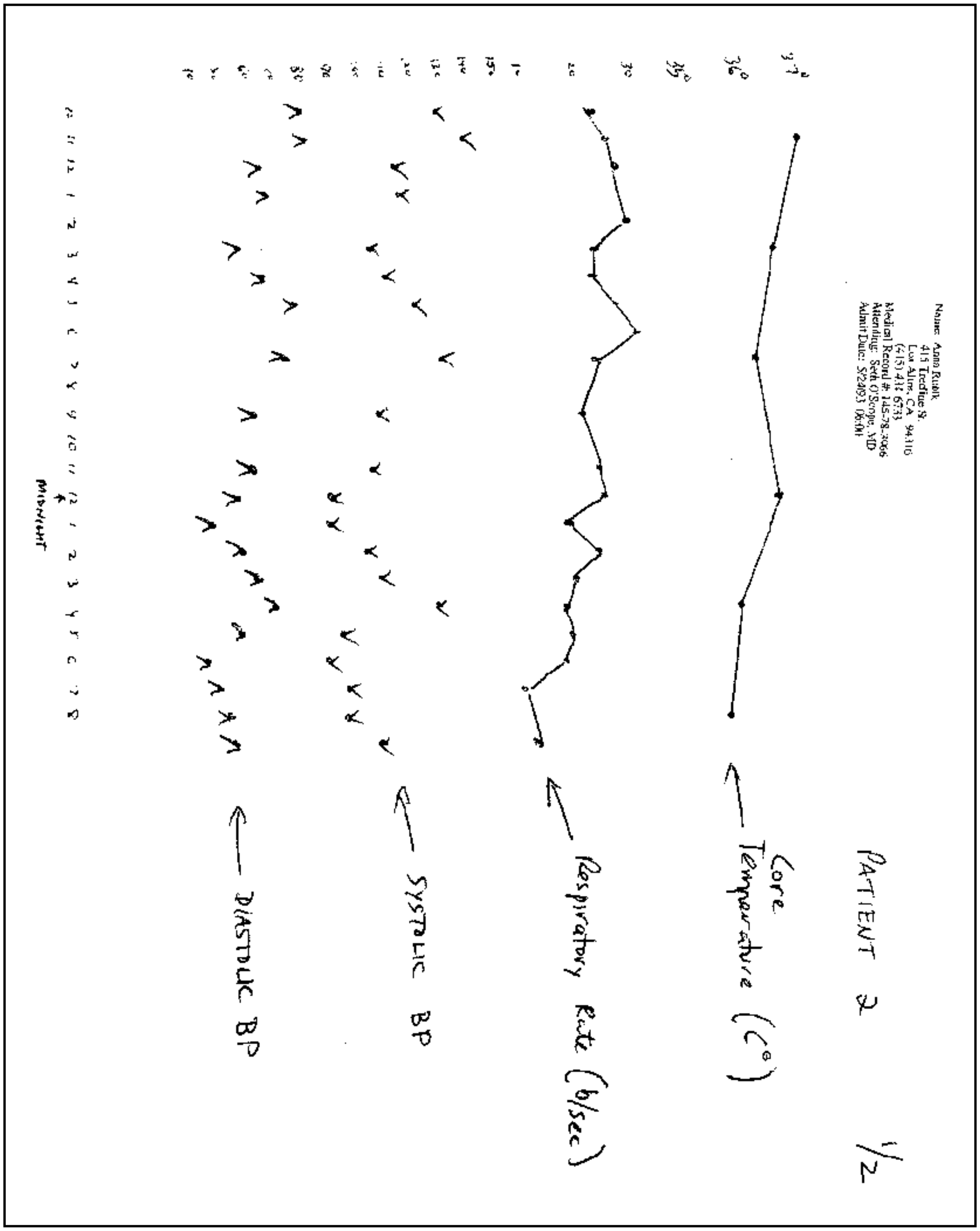
WBC = 19 /mm^3 Platelet Count = 138 /mm^3, Theophylline level = 1.8 µg/ml

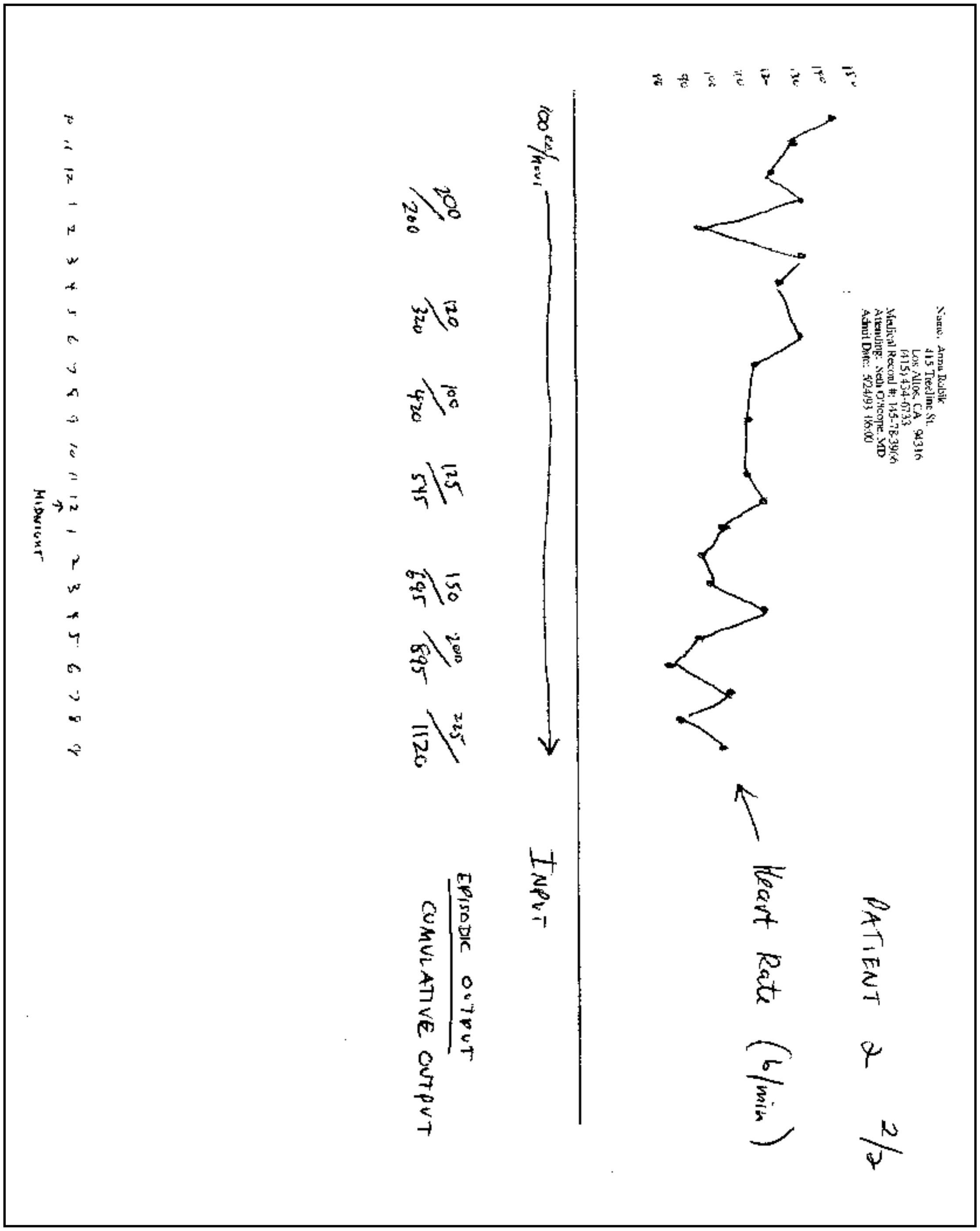
LABS
5/26/93 04:00
Serum NA$^+$ = 143 mEq/L, Serum K$^+$ = 4.0 mEq/L, Serum CL$^-$=103 mEq/L,
Plasma CO2 Content = 23 mEq/L, BUN = 19 mg/dl, Serum Creatinine = 1.3 mEq/L,
Blood Glucose = 147 mg/dl

WBC = 28.5 /mm^3, HCT = 37.8 %

Theophylline level = 5.7 µg/ml

The following two pages contain flowsheet data captured during the patient's stay in the ICU. This data was provided to the database-design teams in exactly the form shown.


PATIENT 2
1/2
Name: Anna Rubik
415 Tredfue St.
Los Alms, CA 94116
(415) 414 6733
Medical Record #: 145-78-3066
Attending: Seth O'Scope, MD
Admit Date: 5/24/93 06:00
Core Temperature (C°)
Respiratory Rate (b/sec)
SYSTOLIC BP
DIASTOLIC BP
37°
36°
35°
30
20
10 11 12 1 2 3 4 5 6 7 8 9 10 11 12 1 2 3 4 5 6 7 8
Midnight


Name: Anna Rubik
415 Treeline St.
Los Altos, CA 94316
(415) 434-6733
Medical Record #: 145-78-3966
Attending: Seth O'Scope, MD
Admit Date: 5/24/93 16:00
PATIENT 2 2/2
150
140
130
120
110
100
90
80
Heart Rate (b/min)
100 cc/hour
INPUT
200/200
120/320
100/420
125/545
150/695
200/895
225/1120
EPISODIC OUTPUT
CUMULATIVE OUTPUT
10 11 12 1 2 3 4 5 6 7 8 9 10 11 12 1 2 3 4 5 6 7 8 9
↑
MIDNIGHT

C.3.2 Physician data

Name: Bill Fold
Date of birth: 5/17/43

Office: 1212 Groupractice Way
Palo Alto, CA 94301
(415) 325-5322
Home: 34 Elm Drive
Los Altos, CA 94233
(415) 495-2144
DEA#: 478832
Specialty: Gastroenterology

Name: Seth O'Scope
Date of birth: 1/26/50

Office: 444 Tallbuilding St.
San Mateo, CA 95300
(415) 744-9003
Home: 12 Waterview Lane
Foster City, CA 94522
(415) 565-3800
DEA#: 597721
Specialty: Internal Medicine

Appendix D

The Evaluation Databases

This appendix contains the relational databases that each volunteer group designed and implemented based on the clinical data set. The definition of each database consists of the Sybase data-definition language commands that each group used to create the schema and the set of codes that each group used to populate the schema.

Note that each schema contains a set of table definitions that was generated by the database-design groups ("`Original Database Tables`") *plus* a set of table definitions that was added subsequently to facilitate the mapping of the relational schema to the FER schema ("Mapping Tables"). For example, each schema contains a table named "`DIAGNOSIS_CODES`" among the coding tables; this table was added *ex post facto* to map the terms that each group used to represent clinical diagnoses to the entity-type names that the FER-schema designer used to represent the same diagnoses. For example, a row in the `DIAGNOSIS_CODES` table might associate the site-specific diagnosis code "D2-51000" to the FER entity type "Asthma." Although tables were *added* to the relational schemas to facilitate the mapping process, none of the existing tables in the schemas were modified in any way.

D.1 Database 1

This database is notable in that (1) all of the allergies that a patient has are represented in the single text field "`allergy`" of the table "`wpatient`," and (2) the chief complaint and medical history for each patient are represented as free-text narratives in the fields "`cc`" and "`history`," respectively, of the table "`wadmin`." In each of these cases, the text strings stored in the database fields must be parsed in order to extract the individual allergies, presenting complaints, and past diagnoses.

D.1.1 Schema for database 1

```
/**********************************************************/
/* Original Database Tables */
/**********************************************************/

create table wpatient
(mrno             char(11)          not null,
 lname            varchar(20)       not null,
 fname            varchar(20)       not null,
 dob              datetime,
 sex              char(1),
 race             varchar(15),
 h_addr           varchar(50),
 h_phone          char(13),
 allergy          text              null)


create table wphysician
(dea              int               not null,
 lname            varchar(20)       not null,
 fname            varchar(20)       not null,
 dob              datetime,
 o_addr           varchar(50),
 o_phone          char(13),
 h_addr           varchar(50),
 h_phone          char(13),
 speciality       varchar(30))

create table wadmin
(mrno             char(11)    not null,
 date             datetime    not null,
 dea              int         not null,
 cc               text,
 history          text        null,
 location         varchar(10),
 tdate            datetime    null)

create table wmed
(mrno             char(11)    not null,
 date             datetime    not null,
 medcd            char(2)     not null,
 dose             float,
```

```
 route            varchar(18),
 frequency        varchar(20))

create table wmedcd
(medcd            char(2)     not null,
 drug             varchar(20),
 units            varchar(20) null)

create table wlabcd
(code             varchar(3)  not null,
 cdesc            varchar(35),
 units            varchar(20))


/************************************************************/
/* Mapping Tables*/
/************************************************************/

/* TABLE FOR ASSOCIATING CLOTTING TESTS WITH THEIR CONTROL TIMES */

create table CLOT_TEST_CONTROL (
   clot_code            char(3)     not null,
   control_code         char(3)     not null )


/* CODING TABLES */

create table DIAGNOSIS_CODES (
        dgx_code        varchar(50)             NOT NULL,
        ent_type        varchar(50)             NOT NULL)

create table PROCEDURE_CODES (
        proc_code       varchar(50)             NOT NULL,
        ent_type        varchar(50)             NOT NULL)

create table LAB_TEST_CODES (
        test_code       varchar(50)             NOT NULL,
        ent_type        varchar(50)             NOT NULL)

create table PHYSIOLOGIC_PARM_CODES (
        parm_code       varchar(50)             NOT NULL,
        ent_type        varchar(50)             NOT NULL)

create table MEDICATION_CODES (
        med_code        varchar(50)             NOT NULL,
        ent_type        varchar(50)             NOT NULL)
```

```
/** ADDITIONAL TABLES FOR VOCABULARY MAPPING AND TYPE HIERARCHY **/

create table FER_TYPE_HIERARCHY (
        ent_type                varchar(50)     not null,
        desc_ent_type           varchar(50)     not null)

create table ENTITY_TYPE_NAMES (
        ent_type                varchar(50)     not null,
        ent_type_name           varchar(50)     not null)
```

D.1.2 Codes for database 1

Observation-Code	Observation	Units
4	ventilator alarms suspended	NA
7	EKG heart rate	bpm
19	mean arterial pressure	mm Hg
20	arterial line systolic BP	mm Hg
21	arterial line diastolic BP	mm Hg
22	arterial line heart rate	bpm
59	arterial O2 saturation	%
80	respiratory rate	breaths/min
81	tidal volume	cc/breath
83	PIP	cm H2O
84	mean airway pressure	cm H2O
135	SGPT	U/L
124	Arterial O2 Saturation	%
125	Arterial Bicarb Conc	mEq/L
123	Arterial pH	NA
128	SGOT	U/L
101	BUN	mg/dl
103	Blood Glucose	mg/dl
112	HCT	%
117	Hemoglobin Conc	g/dl
118	Mean Corpuscular Hemoglobin	pg
114	Mean Corpuscular Hemoglobin Conc	g/dl
108	MCV	fl
102	PTT-Control	sec
106	PTT	sec
122	PaCO2	mmHg
121	PaO2	mmHg
104	Plasma CO2 Content	mEq/L
109	Plasma Fibrinogen	mg/dl
115	Platelet Count	/mm^3
116	Prothrombin Time-Control	sec
100	Prothrombin Time	sec

107	RBC	million/mm^3
132	Serum Albumin	g/dl
134	Serum Ca++	mEq/L
105	Serum Cl-	mEq/L
113	Serum Creatinine	mg/dl
130	Serum Inorganic Phosphate	mg/dl
120	Serum Ionized Ca++	mEq/L
111	Serum K+	mEq/L
126	Serum Mg+	mEq/L
119	Serum Na+	mEq/L
127	Serum Total Protein	g/dl
131	Serum Triglycerides	mg/dl
133	Total Bilirubin	mg/dl
110	WBC	/mm^3
129	Whole Blood Ammonia	mcg/dl
140	Diff Polymorphonuclear cells	%
141	Diff Band cells	%
142	Diff Lymphocytes	%
143	Diff Monocytes	%
144	Diff Eosinophils	%
145	Theophylline assay	mcg/ml
200	Terbutaline	mcg/kg/min
201	Pavulon	mg/hr
202	Versed	mg/hr
203	Fluid Out	cc/hr
204	Dopamine	mg/kg/min
205	Fluid In	cc
206	Fentanyl	mcg/hr
207	Temp	C
208	Zantac	mg/hr
90	systolic BP	mm Hg
91	diastolic BP	mm Hg

Medication-Code	Medication	Units
14	nystatin	
4	ativan	mg
3	Amphotericin	
20	ventolin	
12	Maalox	cc
19	terbutiline	microgram/kg/min
15	Pavulon	mg/hour
21	versed	mg/hour
11	fentanyl	microgram/hour
10	dopamine	mg/kg/hr
2	Albuterol MDI	puffs
18	Theodur	mg

```
5                        Azmacort                 puffs
1                        Albuterol                cc
13                       NS                       cc
6                        Brokosol
8                        Cholorpherinamine
9                        D5NS                     cc/hr
16                       Ranitidine               mg/min
17                       Solumedrol               mg
7                        Cefotetan                gm
```

D.2 Database 2

This database is notable in that (1) there is no explicit representation of admission and discharge diagnoses, only the interval of each diagnosis (in the table "`Diagnosis`") and the interval of each admission (in the table "`Admission`"), and (2) the medications for each patient are *normalized* across the tables "`Therapy`" (which contains the identities of the medications taken by each patient) and "`Therapy_Types`" (which contains the characteristics of all medications, regardless of the patients who are taking them).

D.2.1 Schema for database 2

```
/*************************************************************/
/*  Original Database Tables */
/*************************************************************/

create table Patient (
        mr_number               char(11)        not null,
        last_name               varchar(15)     null,
        first_name              varchar(15)     null,
        street_address          varchar(30)     null,
        city                    varchar(15)     null,
        state                   char(2)         null,
        zip                     char(5)         null,
        tel_number              char(12)        null,
        birthdate               datetime        null,
        race                    varchar(15)     null,
        gender                  char(1)         null )

create table Physician (
        dea_number              char(6)         not null,
        last_name               varchar(15)     null,
        first_name              varchar(15)     null,
```

```
        wk_street_address       varchar(30)     null,
        wk_city                 varchar(15)     null,
        wk_state                char(2)         null,
        wk_zip                  char(5)         null,
        wk_tel_number           char(12)        null,
        hm_street_address       varchar(30)     null,
        hm_city                 varchar(15)     null,
        hm_state                char(2)         null,
        hm_zip                  char(5)         null,
        hm_tel_number           char(12)        null,
        birthdate               datetime        null,
        specialty               varchar(20)     null )

create table Admission (
        mr_number               char(11)        not null,
        dea_number              char(6)         not null,
        admit_datetime          datetime        not null,
        location                varchar(10)     null,
        discharge_datetime      datetime        null )

create table Diagnosis (
        mr_number               char(11)        not null,
        diagnosis_name          varchar(30)     not null,
        start_datetime          datetime        not null,
        end_datetime            datetime        null )
create table Therapy (
        mr_number               char(11)        not null,
        med_code                char(3)         not null,
        start_datetime          datetime        not null,
        end_datetime            datetime        null,
        quantity                varchar(6)      null,
        frequency               varchar(15)     null,
        condition               varchar(15)     null )

create table Therapy_Types (
        med_code                char(3)         not null,
        med_name                varchar(30)     null,
        route                   varchar(10)     null,
        units                   varchar(15)     null )

create table Allergy (
        mr_number               char(11)        not null,
        allergy_name            varchar(30)     not null,
        start_date              datetime        null,
        manifestation           varchar(30)     null )
```

```
create table Observations (
        mr_number               char(11)        not null,
        obs_datetime            datetime        not null,
        obs_code                varchar(3)      not null,
        obs_value               varchar(10)     null )

create table Observation_Types (
        obs_code                char(3)         not null,
        obs_text                varchar(40)     not null,
        obs_units               varchar(15)     null )

/*****************************************************************/
/* Mapping Tables */
/*****************************************************************/

/* TABLE FOR ASSOCIATING CLOTTING TESTS WITH THEIR CONTROL TIMES */

create table CLOT_TEST_CONTROL (
        clot_code      char(3)     not null,
        control_code   char(3)     not null )

/* CODING TABLES */

create table DIAGNOSIS_CODES (
        dgx_code     varchar(50)       not null,
        ent_type     varchar(50)       not null)

create table PROCEDURE_CODES (
        proc_code    varchar(50)       not null,
        ent_type     varchar(50)       not null)

create table LAB_TEST_CODES (
        test_code    varchar(50)       not null,
        ent_type     varchar(50)       not null)

create table PHYSIOLOGIC_PARM_CODES (
        parm_code    varchar(50)       not null,
        ent_type     varchar(50)       not null)

create table MEDICATION_CODES (
        med_code     varchar(50)       not null,
        ent_type     varchar(50)       not null)
```

```
/** ADDITIONAL TABLES FOR VOCABULARY MAPPING AND TYPE HIERARCHY **/

create table FER_TYPE_HIERARCHY (
        ent_type            varchar(50)     not null,
        desc_ent_type       varchar(50)     not null)

create table ENTITY_TYPE_NAMES (
        ent_type            varchar(50)     not null,
        ent_type_name       varchar(50)     not null)
```

D.2.2 Codes for database 2

Observation-Code	Observation	Units
4	ventilator alarms suspended	NA
6	Pulse	bpm
7	EKG heart rate	bpm
19	mean arterial pressure	mm Hg
20	arterial line systolic BP	mm Hg
21	arterial line diastolic BP	mm Hg
22	arterial line heart rate	bpm
59	arterial O2 saturation	%
80	respiratory rate	breaths/min
81	tidal volume	cc/breath
83	PIP	cm H2O
84	mean airway pressure	cm H2O
90	systolic BP	mm Hg
91	diastolic BP	mm Hg
100	Prothrombin Time	sec
100	Weight	kg
101	BUN	mg/dl
102	PTT-Control	sec
103	Blood Glucose	mg/dl
104	Plasma CO2 Content	mEq/L
105	Serum Cl-	mEq/L
106	PTT	sec
107	RBC	million/mm^3
108	MCV	fl
109	Plasma Fibrinogen	mg/dl
110	WBC	/mm^3
111	Serum K+	mEq/L
112	HCT	%
113	Serum Creatinine	mg/dl
114	Mean Corpuscular Hemoglobin Conc	g/dl
115	Platelet Count	/mm^3
116	Prothrombin Time-Control	sec
117	Hemoglobin Conc	g/dl

118	Mean Corpuscular Hemoglobin	pg
119	Serum Na+	mEq/L
120	Serum Ionized Ca++	mEq/L
121	PaO2	mmHg
122	PaCO2	mmHg
123	Arterial pH	NA
124	Arterial O2 Saturation	%
125	Arterial Bicarb Conc	mEq/L
126	Serum Mg+	mEq/L
127	Serum Total Protein	g/dl
128	SGOT	U/L
129	Whole Blood Ammonia	mcg/dl
130	Serum Inorganic Phosphate	mg/dl
131	Serum Triglycerides	mg/dl
132	Serum Albumin	g/dl
133	Total Bilirubin	mg/dl
134	Serum Ca++	mEq/L
135	SGPT	U/L
140	Diff Polymorphonuclear cells	%
141	Diff Band cells	%
142	Diff Lymphocytes	%
143	Diff Monocytes	%
144	Diff Eosinophils	%
145	Theophylline assay	mcg/ml
200	Terbutaline	mcg/kg/min
201	Pavulon	mg/hr
202	Versed	mg/hr
203	Fluid Out	cc/hr
204	Dopamine	mg/kg/min
205	Fluid In	cc
206	Fentanyl	mcg/hr
207	Temp	C
208	Zantac	mg/hr
300	Renal ultrasound	

Medication-Code	Medication	Route	Units
200	Terbutaline	IV	mcg/kg/min
201	Pavulon	IV	mg/hr
202	Versed	IV	mg/hr
204	Dopamine	IV	mg/kg/min
206	Fentanyl	IV	mcg/hr
208	Ranitidine	IV	mg/min
210	Nystatin	Oral	
211	Ativan	IV	mg
212	Amphotericin Bladder Wash	Via Foley	
213	Ventolin	Nebulizer	

214	Maalox	Per NG	cc
215	Albuterol MDI	po	puffs
216	Theodur	po	mg
217	Azmacort MDI	po	puffs
218	Pulmonaid Nebulizer	po	cc/ .2 cc NS
219	Brokosol	po	
220	Chlorpheniramine	po	
221	D5 Normal Saline	IV	cc/hr
222	Solumedrol	IV	mg
223	Cefotetan	IV	g

D.3 Database 3

This database is notable in that it explicitly represents admission diagnoses, discharge diagnoses, problem lists, and procedures in separate tables; SNOMED codes are used to represent instances of each of these categories.

D.3.1 Schema for database 3

```
/****************************************************************/
/*  Original Database Tables */
/****************************************************************/

create table SNOMED_CODES (
        CODE                    char(10)        not null,
        ECLASS                  char(2),
        ENOMEN                  char(200) )

create table patient_demographics (
        Med_Rec_No              char(10) not null,
        last                    char(20),
        first                   char(20),
        home_address            char(20),
        home_city               char(20),
        home_state              char(2),
        home_zip                char(20),
        race                    char(10),
        sex                     char(10),
        DOB                     char(20),
        home_phone              char(20),
        work_phone              char(20) null )
```

```
create table physician_demographics (
        DEA_Num                 char(10) not null,
        DOB                     datetime,
        last                    char(20),
        first                   char(20),
        Office_address          char(20),
        office_city             char(20),
        office_state            char(2),
        office_zip              char(20),
        office_phone            char(20),
        specialty               char(20),
        home_address            char(20),
        home_city               char(20),
        home_state              char(2),
        home_zip                char(20),
        home_phone              char(20) )
create table hospitalizations (
        Med_Rec_No              char(10)         not null,
        unit                    char(20),
        AdmitDate               datetime,
        DischargeDate           datetime         null,
        AttendingDEA            char(20) )

create table AdmitDiagnosis (
        Med_Rec_No              char(10)         not null,
        AdmitDate               datetime,
        Diagnosis               char(10) )

create table DischargeDiagnosis (
        Med_Rec_No              char(10)         not null,
        AdmitDate               datetime,
        Diagnosis               char(10) )

create table Procedures (
        Med_Rec_No              char(10)         not null,
        ProcedureDate           datetime,
        proced                  char(10) )

create table Problems (
        Med_Rec_No              char(10)         not null,
        ProblemStart            datetime         null,
        ProblemEnd              datetime         null,
        Problem                 char(10) )
```

```
create table observations (
        Med_Rec_No              char(10)         not null,
        ObservationDate         datetime,
        ObservationID           char(10),
        ObservationValue        char(10),
        ObservationUnits        char(10) )

create table observation_codes (
        ObservationID           char(10)         not null,
        ObservationDesc         char(40) )

create table Medications (
        Med_Rec_No              char(10)         not null,
        MedDate                 datetime,
        MedID                   char(10),
        MedAmount               char(10),
        MedUnits                char(10)         null,
        MedRoute                char(10),
        MedFrequency            char(20) )

create table Med_codes (
        ObservationID           char(10)         not null,
        ObservationDesc         char(20) )

/*****************************************************************/
/* Mapping Tables */
/*****************************************************************/

/* TABLE FOR ASSOCIATING CLOTTING TESTS WITH THEIR CONTROL TIMES */

create table CLOT_TEST_CONTROL (
     clot_code  char(3)  not null,
     control_code  char(3)  not null )

/* CODING TABLES */

create table DIAGNOSIS_CODES (
        dgx_code   varchar(50)       not null,
        ent_type   varchar(50)       not null)

create table PROCEDURE_CODES (
        proc_code  varchar(50)       not null,
        ent_type   varchar(50)       not null)
```

```
create table LAB_TEST_CODES (
        test_code  varchar(50)       not null,
        ent_type   varchar(50)       not null)

create table PHYSIOLOGIC_PARM_CODES (
        parm_code  varchar(50)       not null,
        ent_type   varchar(50)       not null)

create table MEDICATION_CODES (
        med_code   varchar(50)       not null,
        ent_type   varchar(50)       not null)

/** ADDITIONAL TABLES FOR VOCABULARY MAPPING AND TYPE HIERARCHY **/

create table FER_TYPE_HIERARCHY (
        ent_type           varchar(50)     not null,
        desc_ent_type      varchar(50)     not null)

create table ENTITY_TYPE_NAMES (
        ent_type           varchar(50)     not null,
        ent_type_name      varchar(50)     not null)
```

D.3.2 Codes for database 3

SNOMED-Code	Modifier	Definition
L-85B30	01	Asian race
L-85B30	02	Mongoloid
T-D0AA0	01	Female, NOS
T-D0AA0	02	Female individual, NOS
L-85B50	01	European race
L-85B50	02	Caucasian
L-85B50	02	Caucasoid
D5-81100	01	Hepatic failure, NOS
D5-81100	02	Hepatic insufficiency, NOS
D5-81100	02	Liver function failure, NOS
DC-60000	01	Blood coagulation disorder, NOS
DC-60000	02	Coagulation disorder, NOS
DC-60000	02	Coagulopathy, NOS
D2-60100	01	Adult respiratory distress syndrome, NOS
D2-60100	02	ARDS
D2-60100	02	Congestive atelectasis
D2-60100	02	DaNang lung
D2-60100	02	Post-traumatic pulmonary insufficiency
D2-60100	02	Pulmonary insufficiency following shock
D2-60100	02	Pulmonary insufficiency following surgery
D2-60100	02	Pulmonary insufficiency following trauma
D2-60100	02	Shock lung

D2-60100	02	Traumatic wet lung
D2-60100	02	Vietnam lung
D2-51000	01	Asthma, NOS
D2-51000	02	Allergic bronchitis, NOS
D2-51000	02	Asthmatic bronchitis, NOS
D2-51000	02	Bronchial asthma, NOS
D2-60250	01	Acute respiratory distress, NEC
D2-60250	02	Acute respiratory insufficiency, NEC
P1-KC001	01	Kasai procedure
P1-KC002	01	Renal Cyst Removal
P1-21310	01	Simple excision of nasal polyps
P1-21310	02	Nasal polypectomy
D2-01150	01	Chronic sinusitis, NOS
D2-01150	02	Chronic infection of sinus, NOS
D7-11013	01	Chronic renal failure syndrome, NOS
D4-57010	01	Congenital biliary atresia
P2-22200	01	Mechanical ventilation, NOS
DE-19000	01	Bacterial septicemia, NOS
D5-86430	01	Ascending cholangitis

Observation-Code	Observation
4	ventilator alarms suspended
7	EKG heart rate
19	mean arterial pressure
20	arterial line systolic BP
21	arterial line diastolic BP
22	arterial line heart rate
59	arterial O2 saturation
80	respiratory rate
81	tidal volume
83	PIP
84	mean airway pressure
90	systolic BP
91	diastolic BP
100	Prothrombin Time
101	BUN
102	PTT-Control
103	Blood Glucose
104	Plasma CO2 Content
105	Serum Cl-
106	PTT
107	RBC
108	MCV
109	Plasma Fibrinogen
110	WBC
111	Serum K+

112	HCT
113	Serum Creatinine
114	Mean Corpuscular Hemoglobin Conc
115	Platelet Count
116	Prothrombin Time-Control
117	Hemoglobin Conc
118	Mean Corpuscular Hemoglobin
119	Serum Na+
120	Serum Ionized Ca++
121	PaO2
122	PaCO2
123	Arterial pH
124	Arterial O2 Saturation
125	Arterial Bicarb Conc
126	Serum Mg+
127	Serum Total Protein
128	SGOT
129	Whole Blood Ammonia
130	Serum Inorganic Phosphate
131	Serum Triglycerides
132	Serum Albumin
133	Total Bilirubin
134	Serum Ca++
135	SGPT
140	Diff Polymorphonuclear cells
141	Diff Band cells
142	Diff Lymphocytes
143	Diff Monocytes
144	Diff Eosinophils
145	Theophylline assay
208	Zantac
200	Terbutaline
201	Pavulon
202	Versed
203	Fluid Out
204	Dopamine
205	Fluid In
206	Fentanyl
207	Temp

Medication-Code	Medication
1	Albuterol
2	Albuterol MDI
3	Amphotericin
4	ativan
5	Azmacort

6	Brokosol
7	Cefotetan
8	Cholorpherinamine
9	D5NS
10	dopamine
11	fentanyl
12	Maalox
13	NS
14	nystatin
15	Pavulon
16	Ranitidine
17	Solumedrol
18	Theodur
19	terbutiline
20	ventolin
21	versed

Appendix E

The Evaluation FER Schema

This appendix contains the FER schema that I designed to model the clinical data set used in the evaluation experiment. The constructs of the FER data model are represented graphically as follows:

- Entity types appear in large bold-faced text; for example ,

Patient.

- Relationship types appear in small plain text; For example,

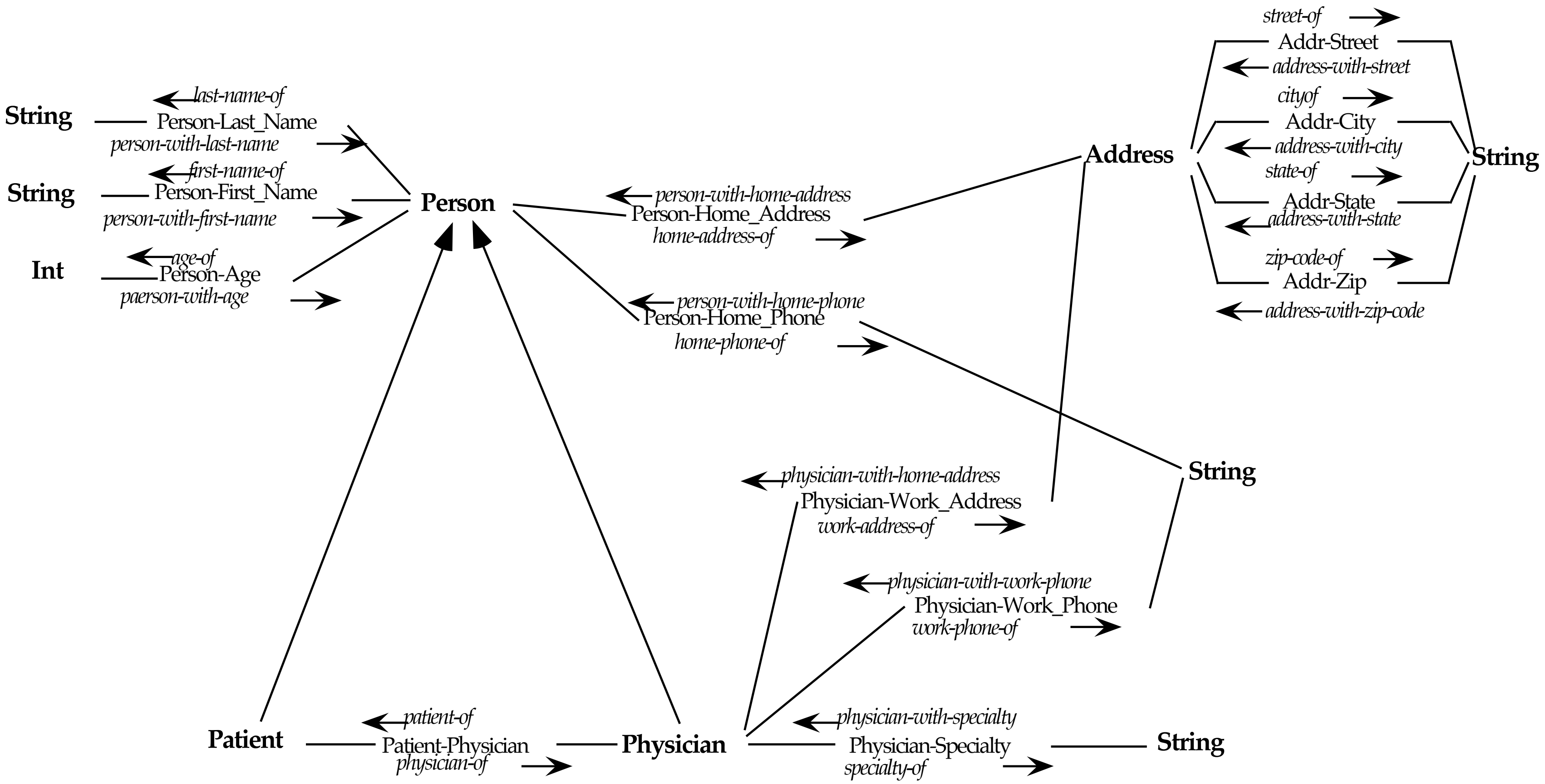
Patient-Physician.

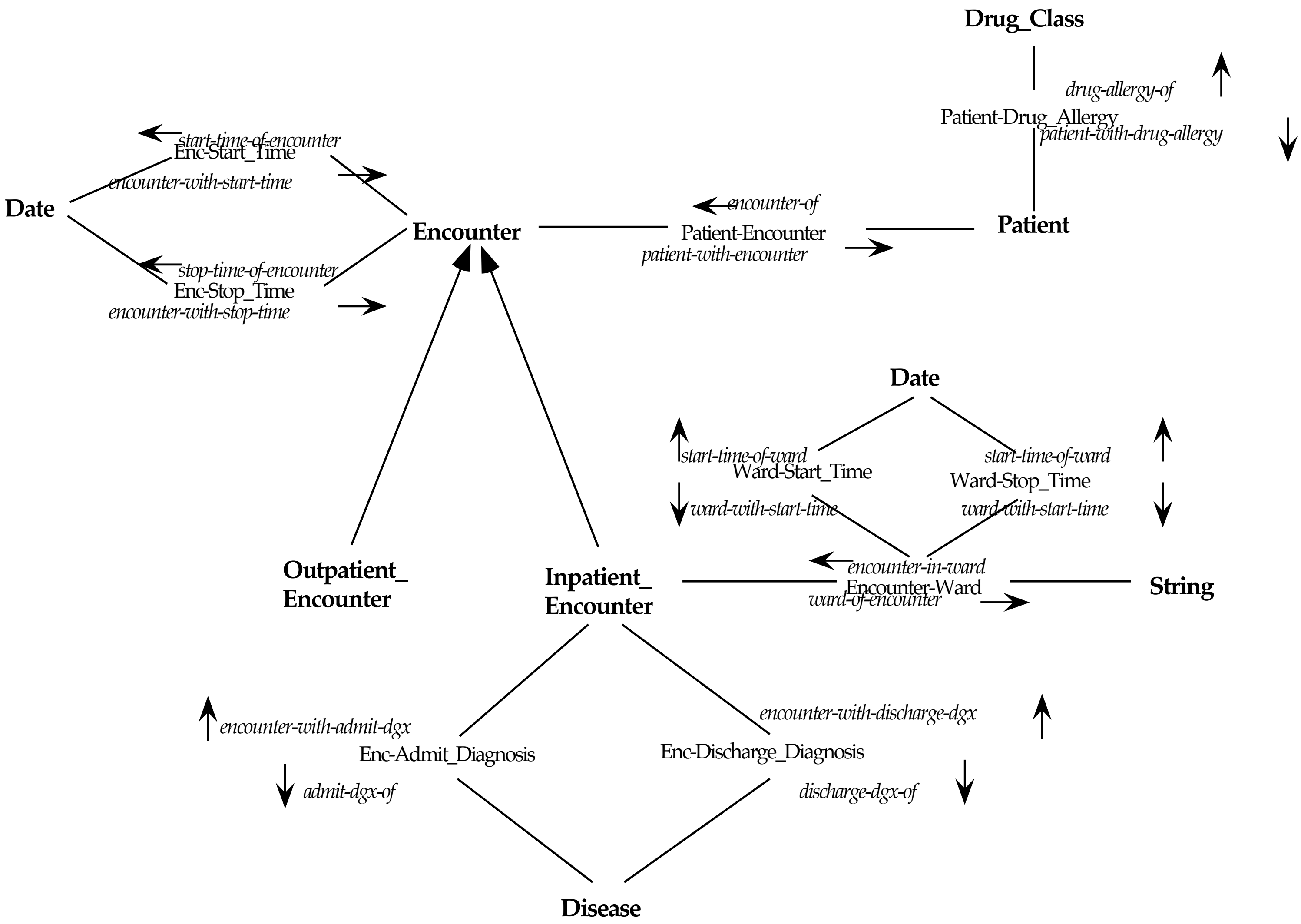
- Relationship functions appear in small italicized text; for example

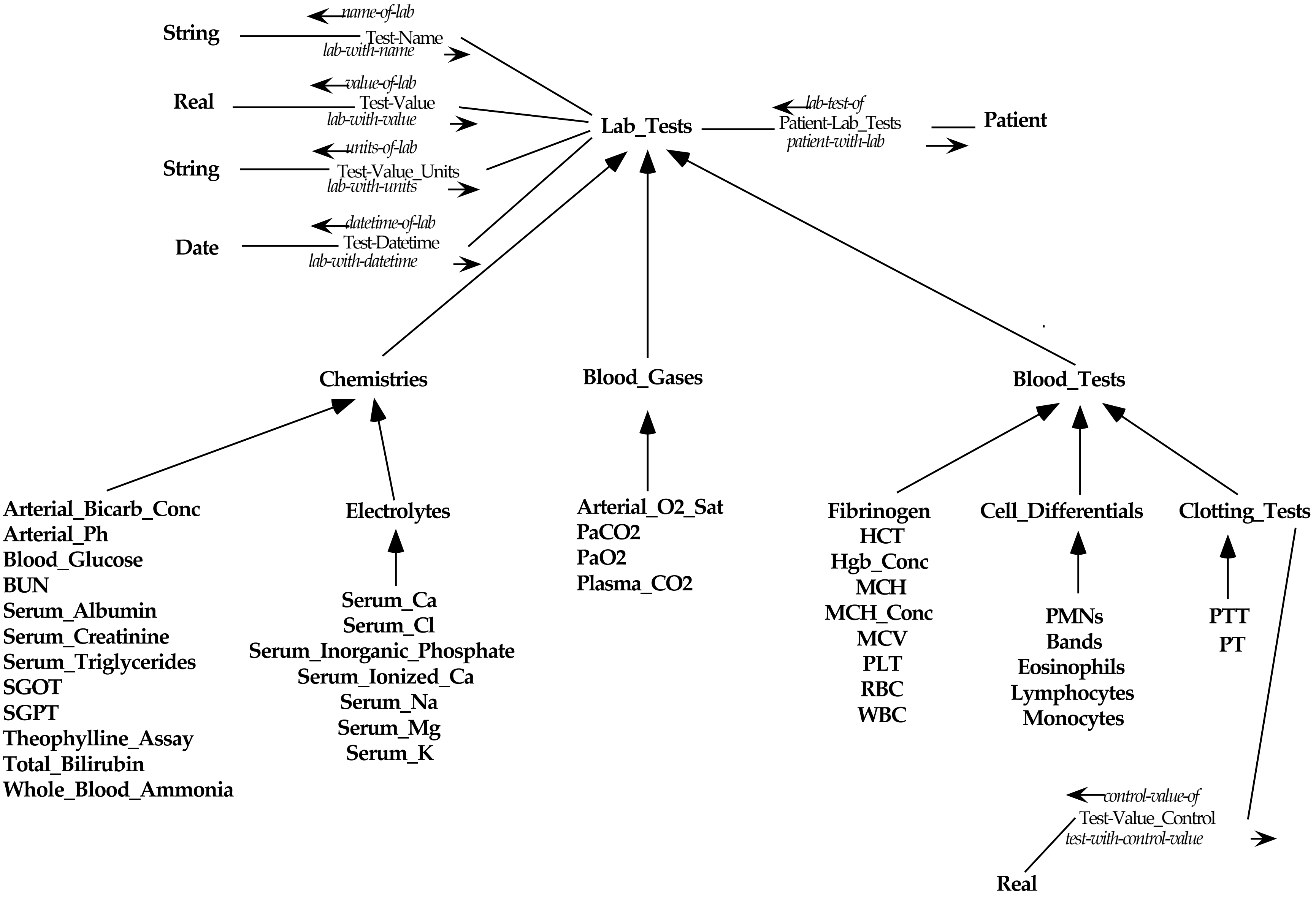
patient-of.

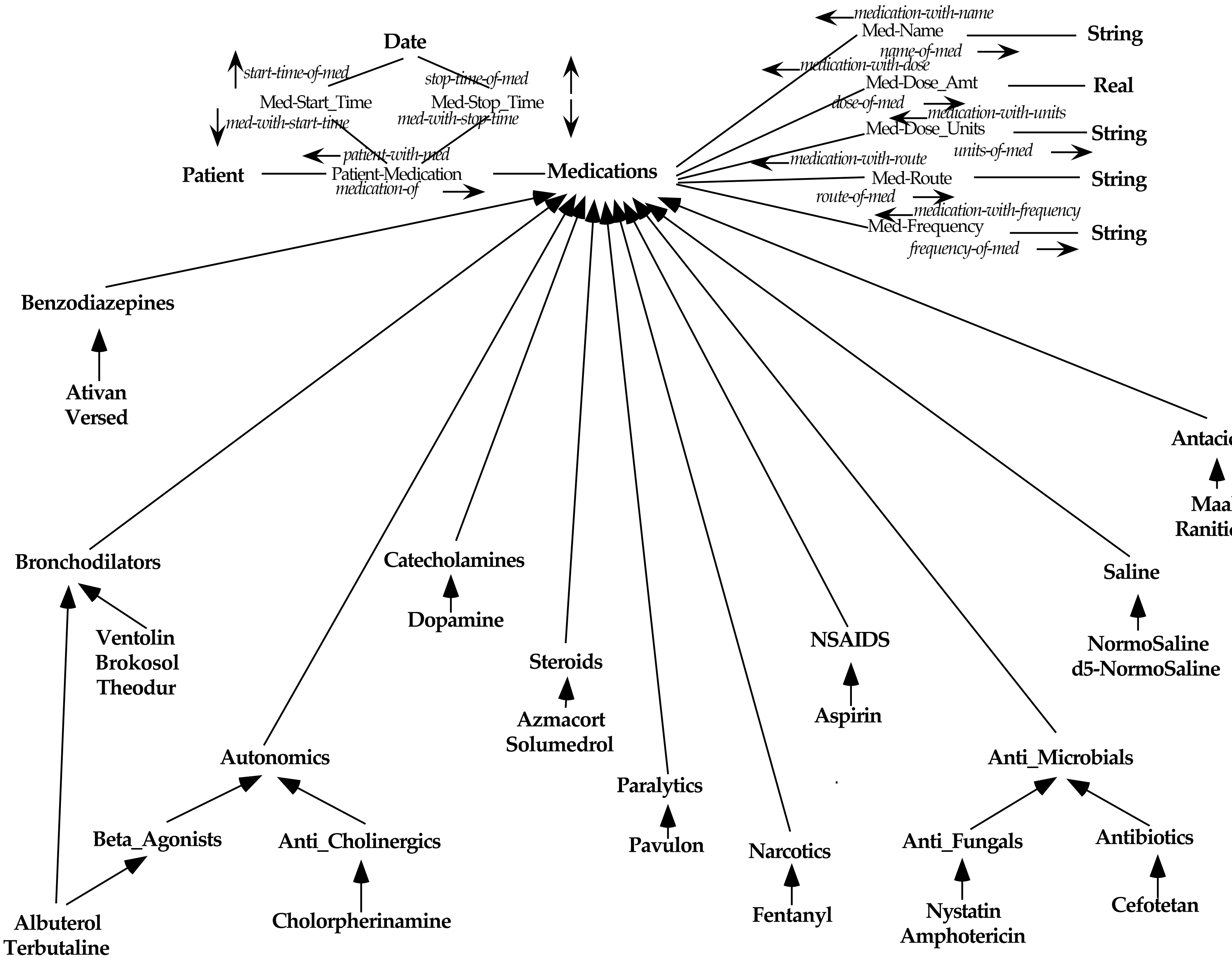
- IS-A connections appear as arrows with no annotations; for example, the arrow from the Patient to the Person entity type on the next page.

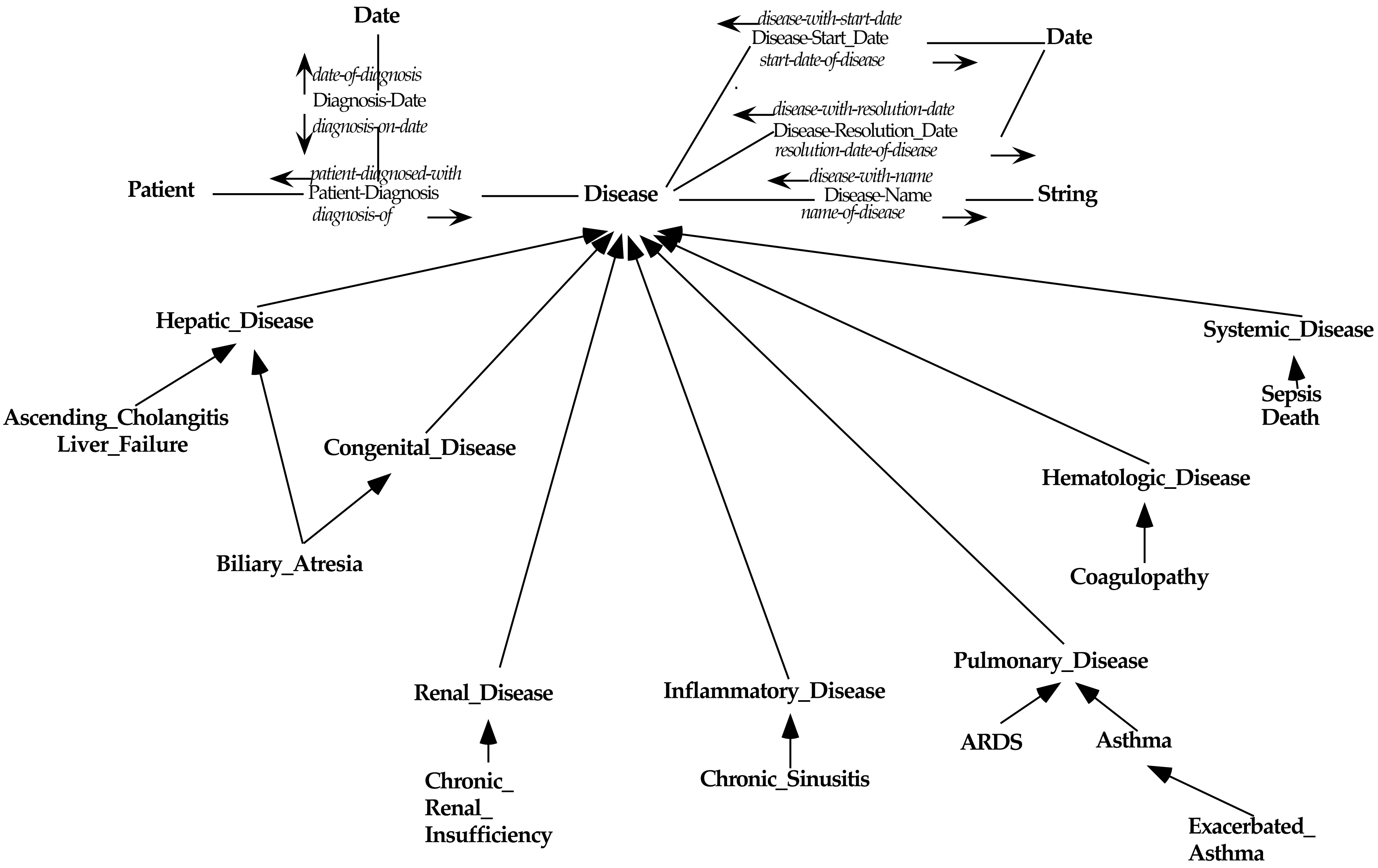
The FER schema represents one formal conceptualization of the real-world semantics represented by the clinical data set in Appendix C. Many other equally valid schemas for the same data set exist, none of which is necessarily "correct." It is important, however, for any conceptualization to have a clear and unambiguous description of its *meaning*. For example, in my conceptualization of the clinical data set, each instance of the entity type Lab_Tests represents a specific lab *result* for a particular patient at a particular time. It does not represent the *kinds* of lab tests that may exist, such as glucose, serum sodium, and PTT. The implication of this modeling decision is that Blood_Glucose, Serum_Na, and PTT are *not instances* of the entity type Lab_Tests, but rather *sub-types*. Any application developer who uses the FER schema and any database programmer who maps the FER schema must be aware of and agree upon this meaning; otherwise, the meanings of certain queries may change when the queries are translated by the TransFER methodology.

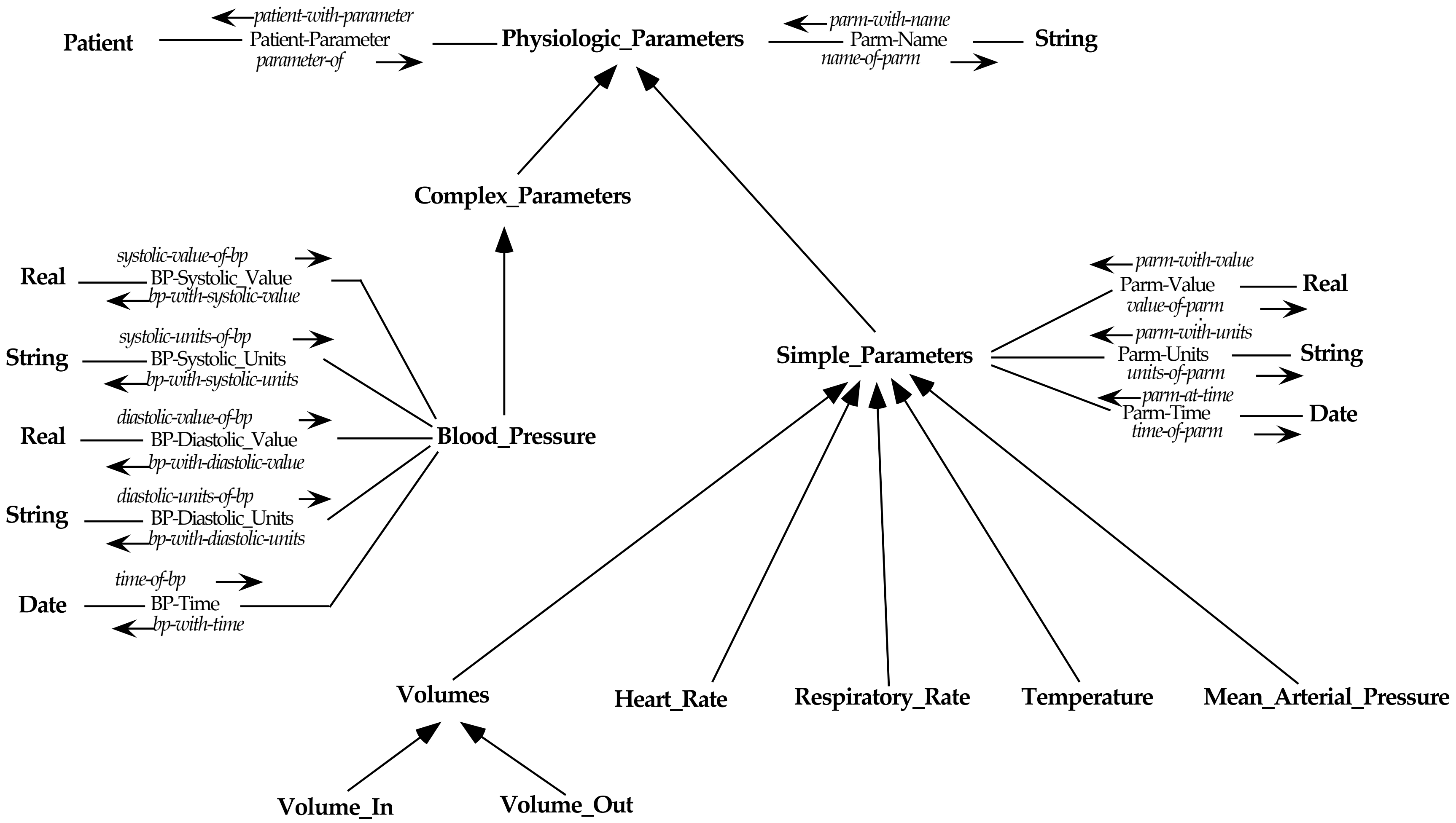

String
last-name-of
Person-Last_Name
person-with-last-name
String
first-name-of
Person-First_Name
person-with-first-name
Int
age-of
Person-Age
paerson-with-age
Person
person-with-home-address
Person-Home_Address
home-address-of
person-with-home-phone
Person-Home_Phone
home-phone-of
Address
street-of
Addr-Street
address-with-street
cityof
Addr-City
address-with-city
state-of
Addr-State
address-with-state
zip-code-of
Addr-Zip
address-with-zip-code
String
String
physician-with-home-address
Physician-Work_Address
work-address-of
physician-with-work-phone
Physician-Work_Phone
work-phone-of
Patient
patient-of
Patient-Physician
physician-of
Physician
physician-with-specialty
Physician-Specialty
specialty-of
String

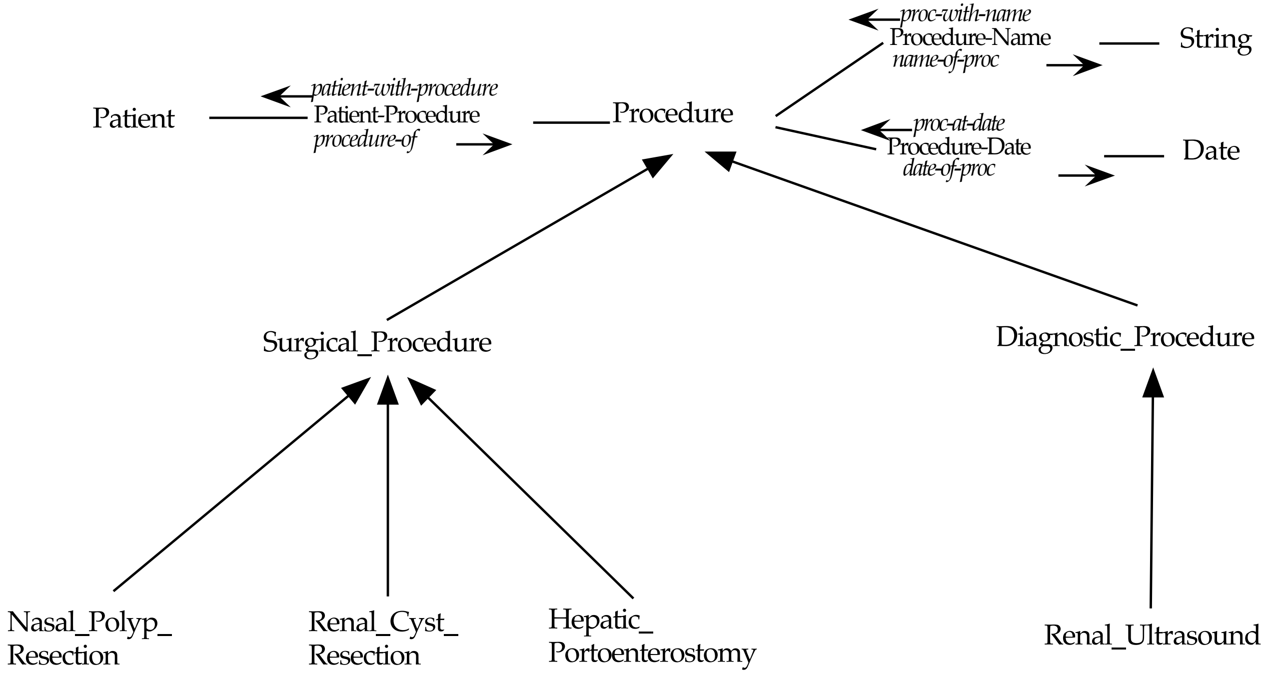

Drug_Class
drug-allergy-of
Patient-Drug_Allergy
patient-with-drug-allergy
start-time-of-encounter
Enc-Start_Time
encounter-with-start-time
Date
encounter-of
Encounter
Patient-Encounter
Patient
patient-with-encounter
stop-time-of-encounter
Enc-Stop_Time
encounter-with-stop-time
Date
start-time-of-ward
Ward-Start_Time
start-time-of-ward
Ward-Stop_Time
ward-with-start-time
ward-with-start-time
encounter-in-ward
Outpatient_
Encounter
Inpatient_
Encounter
Encounter-Ward
String
ward-of-encounter
encounter-with-admit-dgx
encounter-with-discharge-dgx
Enc-Admit_Diagnosis
Enc-Discharge_Diagnosis
admit-dgx-of
discharge-dgx-of
Disease


String
name-of-lab
Test-Name
lab-with-name
Real
value-of-lab
Test-Value
lab-with-value
String
units-of-lab
Test-Value_Units
lab-with-units
Date
datetime-of-lab
Test-Datetime
lab-with-datetime
Lab_Tests
lab-test-of
Patient-Lab_Tests
patient-with-lab
Patient
Chemistries
Blood_Gases
Blood_Tests
Arterial_Bicarb_Conc
Arterial_Ph
Blood_Glucose
BUN
Serum_Albumin
Serum_Creatinine
Serum_Triglycerides
SGOT
SGPT
Theophylline_Assay
Total_Bilirubin
Whole_Blood_Ammonia
Electrolytes
Serum_Ca
Serum_Cl
Serum_Inorganic_Phosphate
Serum_Ionized_Ca
Serum_Na
Serum_Mg
Serum_K
Arterial_O2_Sat
PaCO2
PaO2
Plasma_CO2
Fibrinogen
HCT
Hgb_Conc
MCH
MCH_Conc
MCV
PLT
RBC
WBC
Cell_Differentials
PMNs
Bands
Eosinophils
Lymphocytes
Monocytes
Clotting_Tests
PTT
PT
control-value-of
Test-Value_Control
test-with-control-value
Real


Date
start-time-of-med
stop-time-of-med
Med-Start_Time
Med-Stop_Time
med-with-start-time
med-with-stop-time
Patient
patient-with-med
Patient-Medication
medication-of
Medications
medication-with-name
Med-Name
name-of-med
String
medication-with-dose
Med-Dose_Amt
Real
dose-of-med
medication-with-units
Med-Dose_Units
String
units-of-med
medication-with-route
Med-Route
String
route-of-med
medication-with-frequency
Med-Frequency
String
frequency-of-med
Benzodiazepines
Ativan
Versed
Antacids
Maalox
Ranitidine
Bronchodilators
Ventolin
Brokosol
Theodur
Catecholamines
Dopamine
Steroids
Azmacort
Solumedrol
NSAIDS
Aspirin
Saline
NormoSaline
d5-NormoSaline
Autonomics
Beta_Agonists
Anti_Cholinergics
Cholorpherinamine
Albuterol
Terbutaline
Paralytics
Pavulon
Narcotics
Fentanyl
Anti_Microbials
Anti_Fungals
Nystatin
Amphotericin
Antibiotics
Cefotetan


Date
date-of-diagnosis
Diagnosis-Date
diagnosis-on-date
Patient
patient-diagnosed-with
Patient-Diagnosis
diagnosis-of
Disease
disease-with-start-date
Disease-Start_Date
start-date-of-disease
Date
disease-with-resolution-date
Disease-Resolution_Date
resolution-date-of-disease
disease-with-name
Disease-Name
name-of-disease
String
Hepatic_Disease
Ascending_Cholangitis
Liver_Failure
Congenital_Disease
Biliary_Atresia
Renal_Disease
Chronic_
Renal_
Insufficiency
Inflammatory_Disease
Chronic_Sinusitis
Pulmonary_Disease
ARDS
Asthma
Exacerbated_
Asthma
Hematologic_Disease
Coagulopathy
Systemic_Disease
Sepsis
Death


Patient
patient-with-parameter
Patient-Parameter
parameter-of
Physiologic_Parameters
parm-with-name
Parm-Name
name-of-parm
String
Complex_Parameters
Simple_Parameters
parm-with-value
Parm-Value
value-of-parm
Real
parm-with-units
Parm-Units
units-of-parm
String
parm-at-time
Parm-Time
time-of-parm
Date
Real
systolic-value-of-bp
BP-Systolic_Value
bp-with-systolic-value
String
systolic-units-of-bp
BP-Systolic_Units
bp-with-systolic-units
Real
diastolic-value-of-bp
BP-Diastolic_Value
bp-with-diastolic-value
String
diastolic-units-of-bp
BP-Diastolic_Units
bp-with-diastolic-units
Date
time-of-bp
BP-Time
bp-with-time
Blood_Pressure
Volumes
Heart_Rate
Respiratory_Rate
Temperature
Mean_Arterial_Pressure
Volume_In
Volume_Out


proc-with-name
Procedure-Name
name-of-proc
String
patient-with-procedure
Patient
Patient-Procedure
procedure-of
Procedure
proc-at-date
Procedure-Date
date-of-proc
Date
Surgical_Procedure
Diagnostic_Procedure
Nasal_Polyp_ Resection
Renal_Cyst_ Resection
Hepatic_ Portoenterostomy
Renal_Ultrasound

Appendix F

The Evaluation Queries

This appendix contains the 23 queries that the medical domain experts formulated based on the clinical data set in Appendix C. The domain experts formulated each query in natural language. I subsequently encoded each query in the ReFER query language, based on the FER schema that I designed for the clinical data set and I used the queries for the evaluation experiment. The ReFER encoding of each query follows the natural-language formulation of the query in the following sections. In certain cases, I changed the meaning of the query slightly when I encoded it in the ReFER language. These changes usually entailed adding date information to the query that the domain expert had omitted. In several cases, however, the query was modified in order to be compatible with the FER schema. All changes to the original natural-language queries are noted and explained below. The original query appears in bold-faced text and my ReFER translation of the query appears in fixed-width text.

1. What are the admission diagnoses for Jessica Hepburn?

```
[ name-of-disease(admit-dgx-of(ie)) | (ie:Inpatient_Encounter),
                                       (pt:Patient) AND
                                      Patient-Encounter(pt,ie) AND
                                      last-name-of(pt) = "Hepburn" AND
                                      first-name-of(pt) = "Jessica" ]
```

2. What are the medications for Jessica Hepburn at the time she was admitted to the ICU?

```
[  name-of-med(m) | (m:Medications),(pt:Patient),
                    (ie:Inpatient_Encounter) AND
                    Patient-Encounter(pt,ie) AND
                    ew^^ward-of-encounter(ie) = "ICU" AND
                    pm^^Patient-Medication(pt,m) AND
                    start-time-of-med(pm) <= start-time-of-ward(ew) AND
                      (stop-time-of-med(pm) >= start-time-of-ward(ew) OR
                       stop-time-of-med(pm) IS NULL ) AND
                    last-name-of(pt) = "Robik" ]
```

3. What are the allergies for Anna Robik?

```
[ drug-allergy-of(pt) | (pt:Patient) AND
                        last-name-of(pt) = "Robik" AND
                        first-name-of(pt) = "Anna" ]
```

4. At 8:00 a.m. , what is the lowest pO2 for Jessica Hepburn?

```
MIN ([value-of-lab(po2) | (po2:PaO2),(pt:Patient) AND
                          Patient-Lab_Tests(pt,po2) AND
                          datetime-of-lab(po2) <= "1993/08/17 0800" AND
                          last-name-of(pt) = "Hepburn" AND
                          first-name-of(pt) = "Jessica" ] )
```

Note: I modified this query to retrieve the lowest p02 at 8:00 a.m. *on the day of August 17, 1993.*

5. What is the mean O2 saturation for Jessica Hepburn over the last 24 hours at 8:00 a.m. ?

```
AVG( [ value-of-lab(ao2s) | (ao2s:Arterial_O2_Sat),(pt:Patient) AND
                            Patient-Lab_Tests(pt,ao2s) AND
                            datetime-of-lab(ao2s) <= "1993/08/17 0800"
                                    AND
                            datetime-of-lab(ao2s) >= "1993/08/16 0800"
                                    AND
                            last-name-of(pt) = "Hepburn" AND
                            first-name-of(pt) = "Jessica" ] )
```

Note: I modified this query to retrieve the mean 02 saturation over the previous 24 hours at 8:00 a.m. *on the day of August 17, 1993.*

6. What are the last laboratory results at 8:00 a.m. for Anna Robik?

Note: I could not encode this query because the ReFER query language currently has no *ordering* operator to determine the latest date in a sequence of dates.

7. What is the total input for the past 24 hours at 9:00 a.m. and the total output for the past 24 hours for Anna Robik?

```
[ SUM([value-of-parm(i) | (i:Volume_In) AND Patient-Parameter(pt,i) AND
                              time-of-parm(i) <= "1993/04/11 0900" AND
                              time-of-parm(i) >= "1993/04/10 0900" ]),
  SUM([value-of-parm(o) | (o:Volume_Out) AND Patient-Parameter(pt,o) AND
                              time-of-parm(o) <= "1993/04/11 0900" AND
                              time-of-parm(o) >= "1993/04/10 0900" ])
          | (pt: Patient) AND last-name-of(pt) = "Robik" AND
                             first-name-of(pt) = "Anna" ]
```

Note: This query was modified to compute the total input and output for the past 24 hours at 9:00 a.m. *on April 11, 1993.*

8. **What were the last three arterial blood gas readings for Jessica Hepburn at 1:00 p.m. 8/17/93?**

Note: I could not encode this query because the ReFER query language has no ordering operator to compute the last *N* instances in a set.

9. **What is Jessica Hepburn's average respiratory rate over the past 12 hours at 8:00 a.m.?**

```
AVG ( [ value-of-parm(rr) | (rr:Respiratory_Rate),(pt:Patient) AND
                            Patient-Parameter(pt,rr) AND
                            time-of-parm(rr) <= "1993/08/17 0800" AND
                            time-of-parm(rr) >= "1993/08/16 2000" AND
                            last-name-of(pt) = "Hepburn" AND
                            first-name-of(pt) = "Jessica" ] )
```

Note: I modified this query to retrieve the average respiratory rate over the previous 12 hours at 8:00 a.m. *on the day of August 17, 1993.*

10. **What is Anna Robik's maximum temperature over the past 12 hours at 8:00 a.m. on May 11, 1994?**

```
MAX ([ value-of-parm(t) | (t:Temperature),(pt:Patient) AND
                          Patient-Parameter(pt,t) AND
                          last-name-of(pt) = "Robik" AND
                          first-name-of(pt) = "Anna" AND
                          time-of-parm(t) <= "1993/04/11 0800" AND
                          time-of-parm(t) >= "1993/04/10 2000" ] )
```

11. Who is the attending physician for Anna Robik?

```
[ first-name-of(md),
  last-name-of(md)   | (md:Physician),(pt:Patient) AND
                      Patient-Physician(pt,md) AND
                      last-name-of(pt) = "Robik" AND
                      first-name-of(pt) = "Anna" ]
```

Note: Because the FER schema does not distinguish between attending and consulting physicians, I modified this query to retrieve *all* of the physicians for Anna Robik.

12. What is Jessica Hepburn's home phone number?

```
[ home-phone-of(pt) | (pt:Patient) AND
                      last-name-of(pt) = "Hepburn" AND
                      first-name-of(pt) = "Jessica" ]
```

13. Names of all patients who have pH < 7.25 in last week

```
[ last-name-of(pt),
  first-name-of(pt) | (ph:Arterial_Ph), (pt:Patient) AND
                       Patient-Lab_Tests(pt,ph) AND
                       datetime-of-lab(ph) >= "1993/08/11 0000" AND
                       value-of-lab(ph) < 7.25 ]
```

Note: I modified this query to retrieve the patients with a pH < 7.25 in the previous week *as of August 17, 1993*.

14. Names of all patients currently on Lasix and their lowest value of potassium within the last 24 hours

```
[  last-name-of(pt),
   MIN( [ value-of-lab(k) | (k:Serum_K) AND
                             Patient-Lab_Tests(pt,k) AND
                             datetime-of-lab(k) >= "1993/05/25 0000" ] )
   | (pt:Patient),(med:Ranitidine) AND
          pm^^Patient-Medication(pt,med) AND
          stop-time-of-med(pm) IS NULL ]
```

Note: Because none of the patients in the data set were on Lasix, I modified this query to find the lowest potassium value for the patients on *ranitidine* over the last 24 hours *as of midnight on May 25, 1993.*

15. Names of any patients whose current weight is 4 kg less than their admission weight or 4 kg greater than their admission weight

Note: I could not encode this query because no weight data was represented in the FER schema.

16. Name of any patient who currently has HCT < 30 and is not on ranitidine or cimetidine or an antacid

```
[ last-name-of(pt) | (pt:Patient) AND
                     EXISTS [ (hct:HCT) AND
                              Patient-Lab_Tests(pt,hct) AND
                              value-of-lab(hct) < 30 AND
                              datetime-of-lab(hct) >= "5/25/1993 00:00" ]
                        AND
                     NOT EXISTS [ (med:Antacids) AND
                                  pm^^Patient-Medication(pt,med) AND
                                  stop-time-of-med(pm) IS NULL ] ]
```

17. Names of all patients on a medication to which they are allergic

Note: I could not encode this query because the ReFER query language has no constructs for relating data and metadata (i.e. there is no operator to test if a medication prescribed to a patient is *an instance of* a class of medications to which the patient is allergic).

18. Provide the ABG and respiratory rate within 6 hours before and 1 hour after for any patient receiving mechanical ventilation

Note: I could not encode this query because the ReFER query language has no constructs for performing arithmetic operations on temporal data.

19. How many hours after Anna is placed on cefotetan does her temperature fall below 101?

Note: I could not encode this query because the ReFER query language has no constructs for performing arithmetic operations on temporal data.

20. Name of any patient taking Digoxin and Quinidine on the same day

Note: I could not encode this query because the semantics of "on the same day" are ambiguous: within the same 24 hour period or on the same calendar day?

21. Name of any patient on Theodur with a theophylline level < 10 or > 20

```
[ last-name-of(pt),
  first-name-of(pt) | (pt:Patient), (med:Theodur) AND
                       pm^^Patient-Medication(pt,med) AND
                       stop-time-of-med(pm) IS NULL AND
                       EXISTS [ (ta:Theophylline_Assay) AND
                                Patient-Lab_Tests(pt,ta) AND
                                datetime-of-lab(ta) >
                                            start-time-of-med(pm) AND
                                ( value-of-lab(ta) < 10   OR
                                  value-of-lab(ta) > 20 ) ] ]
```

22. Name of any patient currently on a ventilator who has gone > 4 hours without an ABG reading

```
[ last-name-of(pt) | (pt:Patient) AND
                      NOT EXISTS [ (abg:Blood_Gases) AND
                                   Patient-Lab_Tests(pt,abg) AND
                                   datetime-of-lab(abg) >=
                                   "1993/05/25 1600" ]   ]
```

Note: I modified this query to retrieve the last name of any patient who has gone > 4 hours without an ABG reading *as of 4:00 p.m. on May 25, 1993*, because no data regarding ventilator status was available in the FER schema..

23. Name of any patient with temperature > 101 who has not had a WBC within the last 8 hours

```
[ last-name-of(pt) | (pt:Patient),(t:Temperature) AND
                     Patient-Parameter(pt,t) AND
                     value-of-parm(t) >= 38.3 AND
                     time-of-parm(t) >= "8/16/1993 20:00" AND
                     NOT EXISTS [ (lab:WBC) AND
                                  Patient-Lab_Tests(pt,lab) AND
                                  datetime-of-lab(lab) >
                                               "8/16/1993 16:00"] ]
```

Note: I modified this query to retrieve the last name of any patient with a temperature > 38.3 centigrade who has not had a white blood count within the last 8 hours *as of 4:00 p.m. on August 16, 1993*.

Bibliography

J. B. Adams. 1986. Three surveillance and query languages for medical care. MD *Computing*, 3(1):11-19.

S. S. Agarwal. 1991. *Flexible Relations: A Model for Data in Distributed, Autonomous, and Heterogeneous Databases*. Ph.D. Thesis, Stanford University.

American Society for Testing and Materials. 1992. E 1460 Standard Specification for Defining And Sharing Modular Health Knowledge Bases (Arden Syntax for Medical Logic Modules). In *ASTM Standards, v 14.01*:539-87; ASTM.

J. Annevelink, C. Y. Young and T. P.C. 1992. Heterogeneous database integration in a physician workstation. In P. D. Clayton, ed., *Proceedings of the Fifteenth Symposium on Computer Applications in Medical Care*:368-72. New York; McGraw-Hill.

ANSI-X3H4. 1993. *IRDS Conceptual Schema*. Technical Report 93-196, American National Standards Institute, Inc.

Y. Arens, C. Y. Chee, C. Hsu and C. A. Knoblock. 1993. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Information Systems*, 2(2):127-58.

M. P. Atkinson and O. P. Buneman. 1987. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105.

G. O. Barnett, R. D. Zeilstorff, J. Piggins and J. McLatchey. 1982. COSTAR: a comprehensive information system for ambulatory care. In B. I. Blum, ed., *Proceedings of the Sixth Symposium on Computer Applications in Medical Care*:8-18; IEEE Press.

T. Barsalou and D. Gangopadhyay. 1992. M(DM): An Open Framework for Interoperation of Multimodel Multidatabase Systems. In F. Golshani, ed., *Proceedings of the Eighth International Conference on Data Engineering*:218-27; IEEE Computer Society Press.

C. Batini, S. Ceri and S. B. Navathe. 1992. *Conceptual Database Design, an Entity Relationship Approach*. Redwood City, CA: The Benjamin/Cummings Publishing Comapany, Inc.

R. L. Blum. 1981. Displaying clinical data from a time-oriented database. *Computers in Biomedical Research*, 11:197.

K. E. H. Bradshaw. 1988. *A Computerized Laboratory Alerting System to Warn of Life-Threatening Events*. Ph.D. Thesis, University of Utah.

J. D. Bronzino, R. A. Morelli and J. W. Goethe. 1989. OVERSEER: A prototype expert system for monitoring drug treatment in the psychiatric clinic. *IEEE Transactions on Biomedical Engineering*, 36(5):533-40.

B. G. Buchanan and E. H. Shortliffe. 1984. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading, MA: Addison-Wesley.

D. Burleson. 1995. APIs: linchpin of database connectivity. *Database Programming and Design*, 8(6):39-45.

S. Ceri and G. Pelagatti. 1984. Heterogeneous Distributed Database Systems. In S. Ceri and G. Pelagatti, eds., *Distributed Databases, Principles and Systems*:Chapter 15. New York; McGraw-Hill.

P. P.-S. Chen. 1976. The Entity-Relationship Model—Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9-36.

E. F. Codd. 1970. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377-87.

E. F. Codd. 1972. Relational completeness of data base sublanguages. In R. Rustin, ed., *Data Base Systems*. New York; Prentice-Hall.

E. F. Codd. 1979. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):393-434.

C. J. Date. 1989a. *A Guide to the SQL Standard*. Reading, MA: Addison-Wesley.

C. J. Date. 1989b. Three-valued logic and the real world. *InfoDB,* Winter.

S. M. Deen, R. R. Amin and M. C. Taylor. 1987. Data integration in distributed databases. *IEEE Transactions on Software Engineering,* 13(7):860-64.

R. S. Dick and E. B. Steen, eds. 1991. *The Computer-Based Patient Record*. Washington, D.C.: National Academy Press.

R. Elmasri, J. Weeldreyer and A. Hevner. 1985. The category concept: An extension to the entity-relationship model. *Data and Knowledge Engineering,* 1:75-116.

R. Elmasri and G. Wiederhold. 1979. Data model integration using the structural model. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*.

G. Engels, M. Gogolla, U. Hohenstein, K. Hyulsmann, P. Lohr-Richter, G. Saake and H.-D. Ehrich. 1992. Conceptual modelling of database applications using an extended ER model. In *Proceedings of the Ninth International Conference on Data and Knowledge Engineering*:157-204; Elsevier Science Publishing, Inc.

D. A. Evans, J. J. Cimino, W. R. Hersh, S. M. Huff and D. S. Bell. 1994. Toward a medical-concept representation language. *Journal of the American Medical Informatics Association*, 1:207-17.

R. S. Evans, S. L. Pestotnik and J. P. Burke. 1990. Reducing the duration of prophylactic antibiotic use through computer monitoring of surgical patients. *DICP Annals of Parmacotherapy*, 24:351-54.

M. J. Field and K. N. Lohr, eds. 1992. *Guidelines for Clinical Practice*. Washington, D.C.

R. Fikes, M. Cutkosky, T. Gruber and J. van Baalen. 1991. *Knowledge Sharing Technology Project Overview*. Technical Report KSL 91-71, Knowledge Systems Laboratory, Stanford University.

G. Gardarin and P. Valduriez. 1989. *Relational Databases and Knowledge Bases*. Reading, MA: Addison-Wesley.

M. Genesereth and R. Fikes. 1991. *Knowledge Interchange Format Version 2.2 Reference Manual*. Technical Report Logic-90-4, Computer Science Department, Stanford University.

T. R. Gruber. 1993. *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*. Technical Report KSL 93-4, Knowledge Systems Laboratory, Stanford University.

J. J. Harrington, T. J. R. Benson and A. L. Spector. 1990. IEEE P1157 Medical Data Interchange (MEDIX) Committee Overview and Status Report. In R. Miller, ed., *Proceedings of the Fourteenth Symposium on Computer Applications in Medical Care*; IEEE Computer Society Press.

HL-7 Working Group. 1989. *Health Level 7 Interface Standard Version 2.1*. Philadelphia.

U. Hohenstein. 1990. Automatic tranformation of an entity-relationship query language into SQL. In F. H. Lochovsky, ed., *Proceedings of the Eighth International Conference on Entity-Relationship Approach*:303-21. Toronto, Canada; Elsevier Science Publishing, Inc.

U. Hohenstein and G. Engels. 1990. Formal semantics of an entity-relationship-based query language. In H. Kangassalo, ed., *Proceedings of the Ninth International Conference on the Entity-Relationship Approach*:177-94. Lausanne, Switzerland; Elsevier Science Publishing, Inc.

U. Hohenstein and M. Gogolla. 1989. A calculus for an extended entity-relationship model incorporating arbitrary data operations and aggregate functions. In C. Batini, ed., *Proceedings of the Seventh International Conference on Entity-Relationship Approach*:129-48. Rome, Italy; Elsevier Science Publishing, Inc.

G. Hripcsak. 1994. Desperately seeking data: Knowledge base-database links. In C. Safran, ed., *Proceedings of the Seventeenth Symposium on Computer Applications in Medical Care*:639-43; McGraw-Hill.

G. Hripcsak, P. D. Clayton, T. A. Pryor, P. Haug, O. B. Wigertz and J. Van der lei. 1990. The Arden Syntax for Medical Logic Modules. In R. Miller, ed., *Proceedings of the Fourteenth Symposium on Computer Applications in Medical Care*:200-204; IEEE Computer Society Press.

G. Hripcsak, P. Ludemann, T. A. Pryor, O. B. Wigertz and P. D. Clayton. 1994. Rationale for the Arden Syntax. *Computers and Biomedical Research*, 27:291-324.

R. Hull and R. King. 1987. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3).

J. T. Kelly and J. E. Swartwout. 1990. Development of practice parameters by physician organizations. *Quality Review Bulletin*, February:54-57.

W. Kim and S. Jungyun. 1991. Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer*, December.

A. Klug. 1982. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the Association for Computing Machinery*, 29(3):699-717.

D. E. Knuth. 1968. The semantics of context-free languages. *Mathematical Systems Theory*, 2:127-45.

G. J. Kuperman, R. M. Gardner and T. A. Pryor. 1991. *HELP: A Dynamic Hospital Information System*. New York: Springer-Verlag.

B. J. MacLennan. 1987. *Principle of Programming Languages*. Fort Worth: Harcourt Brace Jovanovich College Publishers.

D. Maier. 1983. *The Theory of Relational Databases*. Rockville, MD: Computer Science Press.

V. M. Markowitz and A. Shoshani. 1993. Object Queries over Relational Databases: Language, Implementation, and Applications. In *Proceedings of the Ninth International Conference on Data Engineering*.

K. A. Marrs, S. A. Steib, C. A. Abrams and M. G. Kahn. 1994. Unifying Heterogeneous Distributed Clinical Data in a Relational Database. In C. Safran, ed., *Proceedings of the Sumposium on Computer Applications in Medical Care*:655-648. New York; McGraw-Hill.

C. J. McDonald. 1984. Reminders to physicians from an introspective computer medical record. *Annals of Internal Medicine*, 100:130-38.

C. J. McDonald, D. K. Martin and J. M. Overhage. 1991. Standards for the electronic transfer of clinical data: Progress and promises. *Topics in Health Record Management*, 11(4):1-16.

S. McLinden, G. D'Ascenzo Carlos and C. E. Oleson. 1990. The evolution of a standard for patient record communication: a case study. In R. Miller, ed., *Proceedings of the Fourteenth Symposium on Computer Applications in Medical Care*:239-43; IEEE Computer Society Press.

R. A. Miller and F. E. Masarie. 1990. Quick Medical Reference (QMR): A microcomputer-based diagnostic decision support system for general internal medicine. In R. Miller, ed., *Proceedings of the Fourteenth Symposium on Computer Applications in Medical Care*:986-88; IEEE Computer Society Press.

M. Musen. 1992. Dimensions of knowledge sharing and reuse. *Computers and Biomedical Research*, 25:435-67.

J. Mylopoulos, P. A. Bernstein and H. K. T. Wong. 1980. A language facility for designing database-intensive applications. *ACM Transactions on Database Systems*, 5(2):185-207.

D. V. Ostler, J. J. Harrington and G. Hannemyr. 1990. A common reference model for healthcare data exchange - P1157 MEDIX system architecture. In R. Miller, ed., *Proceedings of the Fourteenth Symposium on Computer Applications in Medical Care*:235-38; IEEE Computer Society Press.

J. Peckham and F. Maryanski. 1988. Semantic Data Models. *ACM Computing Surveys*, 20(3):153-89.

A. Pirotte. 1978. High-level data base query languages. In H. Gallaire and J. Minker, eds., *Logic and Data Bases*:409-33; Plenum Press.

T. A. Pryor and G. Hripcsak. 1994. Sharing MLMs: An experiment between Columbia-Presbyterian and LDS Hospital. In C. Safran, ed., *Proceedings of the Seventeenth Symposium on Computer Applications in Medical Care*:399-403; McGraw-Hill.

A. L. Rector, W. A. Nowlan and A. J. Glowinski. 1993. Goals for concept representation in the GALEN project. In *Proceedings of the Seventeenth Symposium on Computer Applications in Medical Care*:414-418.

W. Rishel. 1989. Pragmatic Considerations in the design of the HL7 protocol. In L. C. Kingsland, ed., *Proceedings of the Thirteenth Symposium on Computer Applications in Medical Care*:687-90; IEEE Computer Society Press.

N. Sager, M. Lyma, C. Bucknall, N. Nhan and L. J. Tick. 1994. Natural language processing and the representation of clinical data. *Journal of the American Medical Informatics Association*, 1(2):142-60.

J. E. Sammet. 1969. *Programming Languages: History and Fundamentals*. Englewood Cliffs, N.J.: Prentice-Hall.

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

D. Shipman. 1981. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140-73.

E. H. Shortliffe. 1981. ONCONCIN: An aid for the outpatient management of cancer patients. In *Proceedings of the Society for Computer Medicine and the Society for Advanced Medical Systems*:57-58. Washinton, D.C.

J. M. Smith, P. A. Bernstein, N. G. Dayal, T. Landers, K. W. T. Lin and E. Wong. 1986. Multibase--integrating heterogeneous distributed database systems. In A. Gupta, ed., *Integration of Information Systems: Bridging Heterogeneous Databases*:163-75. New York; IEEE Press.

J. Sowa. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley.

W. W. Stead and W. E. Hammond. 1988. Computer-based medical records: The centerpeice of TMR. *MD Computing*, 5(5):48-62.

M. C. Toepp and N. Kuznets, eds. 1993. *Directory of Practice Parameters*: American Medical Association.

J. D. Ullman. 1988. *Principles of Database and Knowledge-base Systems*. Volume 1. Rockville, MD: Computer Science Press.

J. D. Ullman. 1989. *Principles of Database and Knowledge-base Systems*. Volume 2. Rockville, MD: Computer Science Press.

J. van der Lei and M. A. Musen. 1991. A model for critiquing based on automated medical records. *Computers and Biomedical Research*, 24:344-78.

M. VanAmringe and T. E. Shannon. 1992. Awareness, assimilation, and adoption: The challenges of effective dissemination and the first AHCPR-sponsored Guidelines. *Quality Review Bulletin*, 18(12):397-404.

G. Wiederhold. 1983. *Database Design*. New York: McGraw-Hill.

G. Wiederhold. 1992. Mediators in the Architecture of Future Information Systems. *IEEE Computer*, March:38-49.

K. Wilkinson, P. Lyngbaek and W. Hasan. 1989. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, December.

J. Yen, H. L. Juang and R. MacGregor. 1991. Using polymorphism to improve expert system maintainability. *IEEE Expert*, 6(1):48-55.

A FORMAL MODEL FOR BRIDGING HETEROGENEOUS RELATIONAL DATABASES IN CLINICAL MEDICINE

A DISSERTATION
SUBMITTED TO THE PROGRAM IN MEDICAL INFORMATION SCIENCES
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

By
Walter Sujansky
January 1996