Facilitating Transformations in a Human Genome Project Database [*]

S. B. Davidson, A. S. Kosky
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
Email: susan@cis.upenn.edu, kosky@saul.cis.upenn.edu

B. Eckman
Department of Genetics
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6145
Email: eckman@cbil.humgen.upenn.edu

Contact author: Susan B. Davidson, Phone (215) 898-3490, Fax (215) 898-0587

Abstract

Human Genome Project databases present a confluence of interesting database challenges: rapid schema and data evolution, complex data entry and constraint management, and the need to integrate multiple data sources and software systems which range over a wide variety of models and formats. While these challenges are not necessarily unique to biological databases, their combination, intensity and complexity are unusual and make automated solutions imperative. We illustrate these problems in the context of the Philadelphia Genome Center for Human Chromosome 22, and describe a new approach to a solution for these problems, by means of a deductive language for expressing database transformations and constraints.

1 Introduction

Human Genome Project databases present a confluence of interesting database challenges: rapid schema and data evolution, complex data entry and constraint management, and the need to integrate multiple data sources and software systems which range over a wide variety of models and formats. These challenges are particularly common to laboratory notebook databases, within the Human Genome Project as well as within the broader realm of biological databases, where their combination, intensity and complexity make automated solutions imperative. Furthermore, techniques to aid in their solution either do not exist or are inadequate in this domain. This paper illustrates these problems in the context of the database developed at the Philadelphia Genome Center for Chromosome 22, and describes a first step to solving what we perceive to be the core of these problems: a language in which to express data transformations and constraints.

The goal of the Human Genome Project (HGP) is to sequence the 24 distinct chromosomes comprising the human genome. Each chromosome is composed of a long, double-stranded molecule of DNA (deoxyribonucleic acid) made up of complementary pairs of four different nucleotides or *bases* (A, G, C, T), arranged like beads on a string. *Sequencing* DNA means discovering the exact sequence of A's, C's, T's, and G's on the string. Although there are techniques for directly sequencing short DNA strings (approximately 400 bases), current methods are not practical for sequencing the entire genome (3 billion bases) at one time. Consequently the HGP has set mapping the chromosomes as a less ambitious intermediate goal. *Mapping* involves the ordering of identifiable DNA fragments as markers along the chromosome, and anchoring markers at known positions to serve as landmarks. The database discussed in this paper, Chr22DB, is the laboratory notebook for the Philadelphia Genome Center for Chromosome 22, located at the University of Pennsylvania and Children's Hospital of Philadelphia.

One of the major problems faced in HGP databases is rapid schema evolution and the resulting need to modify existing applications. New and better experimental techniques are constantly being developed and the experimental data being modeled is constantly changing, forcing evolution of the laboratory notebook database schema and related applications. This process must occur extremely rapidly, since investigators consult the database to plan and guide ongoing experimentation. Furthermore data integrity is crucial. However, the data is very complex, hierarchically organized, and contains an unusually large number of links among tables (inclusion dependencies). This gives rise to a number of complex, non-standard constraints that need to be specified and enforced in order for the data to be correct.

[*] This research was supported in part by the following grants: NSF IRI9004137, ARO DAAH0493G0, NIH P50-HG-00425, NSF BIR-9402292 and ARO DAAL03-89-C0031 PRIME.

Another major problem is that access to multiple, heterogeneous remote databases and software packages is frequently needed to augment the contents of the laboratory notebook databases and to answer queries posed by researchers. These include archival databases, such as the nucleic acid sequence database, Genbank, the protein sequence data base, PIR [1], the biomedical bibliographic data base, Medline, and the human genome map data base, GDB [2]; a growing number of laboratory notebook databases; as well as software systems such as BLAST [3], FASTA [4], and Staden, which perform complex data analysis involving such computational problems as pattern-matching, search and string comparison. These databases include flat-relational databases (Sybase), object-oriented databases (Object Store, GemStone), complex-relational databases (ASN.1), and personal-computer-based databases.

This heterogeneity in schemas and models within the HGP is likely to persist. As data complexity increases, different databases may capture only partial, and perhaps significantly different views of the data as a whole; as analysis tasks increase in complexity beyond simple queries, it is often necessary to organize the data to optimize a specific application to achieve acceptable system performance. Thus, we find numerous independent structurings of the same or similar information. The GenBank family is a case in point: there is the "standard" flat-file version with numerous trivial syntactic variants, a relational version developed at the Los Alamos National Laboratory [6], the ASN.1 version developed at NCBI, a relational version developed from the ASN.1 version by the Philadelphia Center for Chromosome 22 [7], and at least one knowledge base version, also developed within our group [8], which transforms the data from a sequence entry view to a biological concept view. Each of these has its own advantages and disadvantages that include issues of representation, query language expressiveness, and portability, among others.

Two recent papers underscore the problems that we have been alluding to, and indicate that they are pervasive to HGP databases:

- A recent report of a Department of Energy Informatics "summit" [9] listed a number of simple queries that were impossible to answer with the current data sources, because the sources are distributed among various databases, programs and structured files, and there is no effective technique for combining them.
- Goodman et al [10] in an appraisal of their attempt to create a genome information system listed two major issues that they faced: (a) the lack of an adequate query language for the DBMS they were using; and (b) the fact that the underlying schema was constantly evolving.

An important part of this is the problem of *transforming data* into some form that is understandable by users, a query language, or an applications program. The problem of schema or data evolution calls for flexible tools for rapidly re-mapping databases. We need a principled approach to data transformations: transformations between schemas in a single data model (as with schema evolution), between different data models (as with data entry screens, and as in the Genbank family of databases), or across multiple data models (as in the integration of data from multiple sources).

The purpose of this paper is to describe our approach to specifying data transformations, and illustrate it using a sample problem of data transformations that has arisen in Chr22DB. Our approach is based on a declarative language, TSL, for specifying transformations and constraints. While declarative query languages (datalog and its extensions) have not yet gained universal acceptance as query languages, we believe they *are* the right approach to data transformations. The reason for this is that, while a data transformation can be thought of as a query, it is one in which the computational forms used are very simple and whose output is rather large and structurally complex – a whole database rather than a single relation. Furthermore, it is highly desirable to have the query in a form that is easy to analyze and to reason about in light of rapid evolution.

The remainder of this paper is organized as follows: Section 2 illustrates a part of Chr22DB and a data transformation problem that has been encountered. Section 3 describes our transformation language and shows how it is used to capture the sample problem. We conclude by arguing how current techniques fail to address the problems we have encountered, and discussing future research.

2 A Sample Data Transformation in Chr22DB

The data and schemas in the archival and laboratory notebook databases for the HGP are highly complex and difficult to understand, especially for those who know little to nothing about molecular biology. We will therefore start off by explaining a bit about what is being modeled and what some of the terms used mean.

2.1 A Databaser's View of the Biological Background

The HGP's intermediate goal is *mapping*: ordering markers (fragments of DNA) along the human chromosome and locating them at known positions. A variety of techniques are used to anchor markers to specific locations on the chromosome. For the sake of simplicity, we will consider only one: *physical mapping* using cloned probes and Sequence Tag Sites (STS's).

The chromosome of interest is cut randomly into overlapping pieces of experimentally manipulable size (50,000-1 million bases). These pieces are then reassembled into a linear ordering representing their order in the original DNA string. To discover the relative ordering of fragments, it is crucial to be able to ascertain when the sequence of two pieces of DNA overlaps, that is, when the pieces come from neighboring sites in the original string. Sequence overlap between two pieces of DNA can be detected by showing that their sequences contain the sequence of a third, much shorter fragment, called a *probe*. The linear ordering on the pieces yields a linear ordering on the probes whose sequence is contained in them, and vice versa. The probes then become the desired map landmarks and may be used to sequence areas of special interest, such as regions thought to be related to inheritable disease.

Physical mapping and its relationship to DNA sequence is illustrated in Figure 1. At the top of this figure, a chromosome is depicted with the banding patterns visible under a microscope, which themselves function as landmarks at the coarsest level of granularity. Vertical lines denote markers (probes). Horizontal lines denote larger, overlapping DNA fragments whose sequence contains marker sequence. Below, the sequence of a tiny substring of DNA is shown.

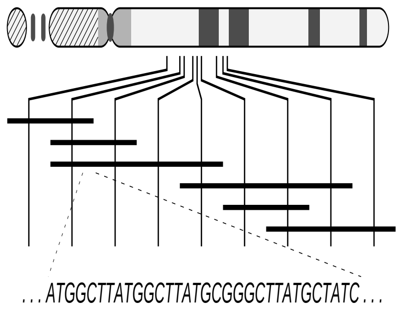


Figure 1: Physical Mapping of a Chromosome

Two types of probes used in physical mapping and represented in Chr22DB are: 1) Cloned probes and 2) Sequence Tag Sites (STS's). Cloned probes are actual physical reagents stored in freezers, and STS's are information stored in a database. In what follows, we briefly describe some of the information maintained about probes by Chr22DB.

Cloned Probes. In cloning, a fragment or *interval* of human DNA is inserted into carrier or *vector* DNA in bacterial or yeast cells. When the host cells are cultured, many exact replicas of the human DNA are produced, to be used in future experiments.

Sequence Tag Sites (STS's). An STS is an interval of DNA defined by a *primer pair*: a pair of sequenced nucleic acid intervals used as primers to start a chemical reaction called *amplification by the polymerase chain reaction* (PCR amplification). The entire reaction comprises several stages, each proceeding at a different temperature. An amplification reaction will not occur unless the primer sequences are found, properly spaced, within the test sequence; therefore, a successful reaction demonstrates sequence containment. Important data items about an STS are: its name, including the laboratory which named it; the name, sequence, and melting temperature of each of the primers; the expected size range of the amplified product; the temperature and time required for each stage of the process (PCR conditions); a cross-reference to GDB; the name of the cloned probe from which the primers were derived; and the chromosomal location of the site.

2.2 A Sample Database Transformation

The data in Chr22DB comes from a variety of sources: archival databases such as GDB, preexisting spreadsheet databases, object-oriented laboratory notebook databases from other centers, as well as directly from experiments being carried out at the center. Importing data from these sources involves data transformations, and has to date been done largely by hand. Though not particularly glamorous, data entry is a special case of a data transformation, and provides a good illustration of some of the complexity of structural transformations. Rewriting the data entry applications is enormously time consuming since application generator tools can not handle the complexity of the data involved, and the modification has to be done by hand.

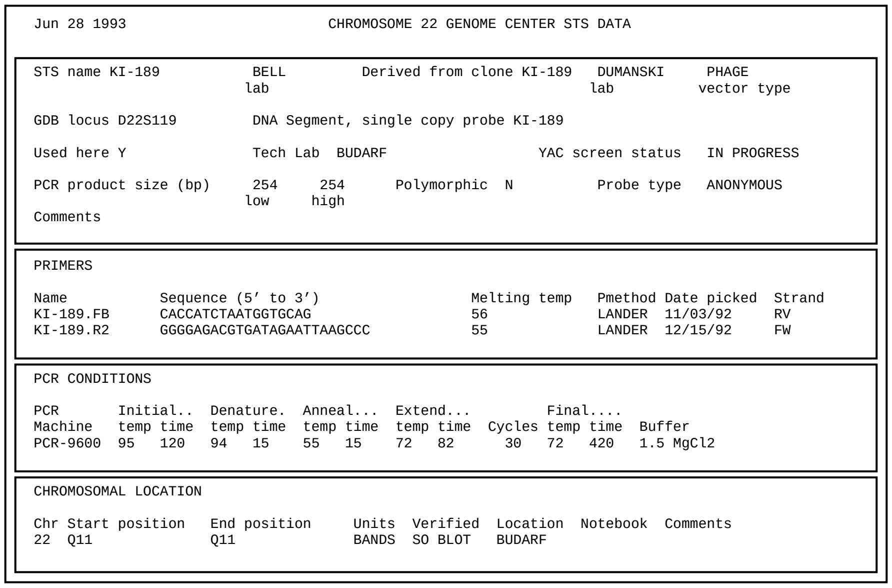
In data entry applications, the data are captured on a screen form that provides a specialized view of the underlying database. The view and the database may differ widely in structure, and the application must map between these two schemas. An example of a form used to enter STS lab notebook data is shown in Figure 2. It consists of a complex relation (`STS`) with three subrelations (`Primers`, `PCR_conditions`, `Location`). Since each screen enters a single STS, there will be one row in the `STS` relation, two rows in the `Primers` relation denoting the primer pair, and multiple rows in the `PCR_conditions` and `Location` relations.

Data entered at the data-entry screen must be transformed to the underlying (relational) Chr22DB database. A conceptual (EER) schema[1] of the relevant portion of Chr22DB is shown in Figure 3; this is merely introduced to convey the linkages between relations rather than to give a precise semantics of the schema. Some of the relevant relational tables and attributes for this schema are given below. Uppercase attribute names denote primary keys.

```
names(MATERIAL_ID, LAB_CODE, NAME, public_name)
primer(ID, pname, picked_from_na_interval_id,
   melting_temp, pick_method, date_picked, strand)
STS(ID, pr1_primer_id, pr2_primer_id,
   PCR_prod_size_lo, PCR_prod_size_hi)
PCR_conditions(STS_ID, AMPL_MACHINE, ANNEAL_TEMP,
   init_denat_time, denat_temp, denat_time, ...)
```

In this database transformation a complex relation with nested subrelations is flattened into a standard relational schema with value-based pointers linking related tables. The atomic attributes of the top-level

[1]The schemas in this paper were all drawn using ERDRAW [13].



```
Jun 28 1993                        CHROMOSOME 22 GENOME CENTER STS DATA

STS name KI-189           BELL         Derived from clone KI-189   DUMANSKI     PHAGE
                         lab                                       lab          vector type

GDB locus D22S119         DNA Segment, single copy probe KI-189

Used here Y               Tech Lab  BUDARF                  YAC screen status   IN PROGRESS

PCR product size (bp)     254     254      Polymorphic  N          Probe type   ANONYMOUS
                         low     high
Comments

PRIMERS

Name          Sequence (5' to 3')                  Melting temp   Pmethod Date picked  Strand
KI-189.FB     CACCATCTAATGGTGCAG                   56             LANDER  11/03/92     RV
KI-189.R2     GGGGAGACGTGATAGAATTAAGCCC            55             LANDER  12/15/92     FW

PCR CONDITIONS

PCR       Initial..  Denature.  Anneal...  Extend...          Final....
Machine   temp time  temp time  temp time  temp time  Cycles temp time  Buffer
PCR-9600  95   120   94   15    55   15    72   82       30   72   420   1.5 MgCl2

CHROMOSOMAL LOCATION

Chr Start position   End position     Units  Verified  Location  Notebook  Comments
22  Q11              Q11              BANDS  SO BLOT   BUDARF
```

Figure 2: STS Data Entry Screen

screen relation are distributed over 6 relational tables in the target schema: names, lab, material, interval, na_interval, and STS. The Primers subrelation is decomposed into 5 target tables: na_interval, interval, material, primer, sequence. The two name fields in the entry screen (STS_name and GDB_locus) are mapped to two separate rows in the target names table, which are linked by the internal identifier of the object being inserted.

In order to accomplish the data transformations, appropriate insert statements must be generated. The normalized target schema relies on internal system-generated identifiers to accomplish the links among related tables.

To maintain data integrity, the transformed data must conform to the integrity constraints of the target database. Preeminent are key and inclusion dependency constraints, but more complex constraints may also hold. For example, each material must have at least one GDB name (i.e., names.lab_code = "GDB") and at least one non-GDB name (i.e, names.lab_code $\neq$ "GDB").

3 A Language for Database Transformations and Constraints

We believe that a deductive approach is the best choice for expressing data transformations. There are several reasons for this: transformations should be easy to modify and reason about; the language should be able to easily express the structural manipulation of complex data types, though it does not need to have the computational expressiveness of a general purpose programming language; and finally, the language should unify transformations and integrity constraints since there is a significant level of interaction between the two. Not only do constraints play a part in determining transformations between databases, but a transformation may imply certain constraints on the source and target database.

Our language is based on Horn-clause logic and allows for formal reasoning about database transformations, constraints and the interactions between the two. Not only can transformations be expressed in this language, but unambiguous and nonrecursive transformation programs can be implemented using code generators for a variety of database programming languages. The proposed code generators will work in two stages: First rules are converted to a *normal form*, each rule specifying how a complete entry for the target database is generated from the source database. The normalised rules are then converted into code for the appropriate DBMS. This approach means that logical inferences are performed only once at the rule level, rather than many times at the data level. Further it is straightforward to re-use the core of the program, allowing easy adaptation of the code generator for a variety of database systems.

We will start by explaining the underlying data model, then giving the syntax of the language with several examples of constraints and transformation clauses that have been generated for the data entry application described in the previous section. Finally, we describe normal forms, and how they are used in implementing transformation programs.

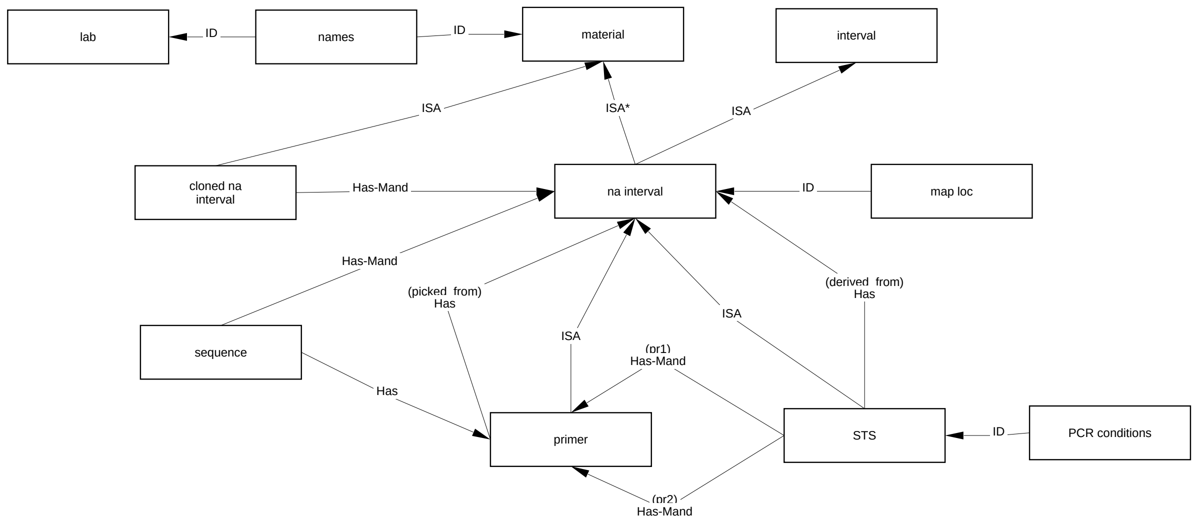


Figure 3: Schema of Target Database for STS's.

3.1 Data Model

The language is based around a nested relational data model which allows us to describe and implement transformations between a wide range of data models. The model is similar to that of [14], and allows for arbitrary nesting of set and tuple constructors, and for a representation of object identities. It is a natural extension of the relational data model, but also allows us to represent the complex data structures found in the various semantic and object-oriented data models that are currently gaining popularity. In addition, the model allows the attributes of records to be either *required* or *optional* (*not-null* or *null*).

We use Skolem functions to generate object identities as in [12]. Skolem functions can be applied to a group of values in order to create an entirely new value, which can then be used as an object identifier, or as a way of referencing a row in some particular relation from other relations. The type system for the language ensures that the values generated by two distinct Skolem functions can not be confused.

Many established data models incorporate various kinds of constraints as primitives: relational data models may support keys or other functional dependencies, certain existence dependencies and inclusion constraints, while object-oriented databases may support some concept of inheritance and object identity. However there remain many important dependencies which occur in biological and other databases but do not fall into any of these categories. Rather than making any such classes of constraints primitive in our data model, our language provides a means to express a very general family of constraints, including but not limited to those mentioned above.

3.2 The Language

Our language allows independent access to the the individual components of a relation or tuple, and variables can be bound to simple values, tuples in a relation or entire relations (sets of tuples). Individual clauses describe one conceptual part of a transformation, rather than describing the construction of an entire tuple in a relation. In this respect it differs from established logic-based database query languages, such as Datalog and ILOG, in which relation names are used as predicates, and variables are bound to base values only; a practice which becomes awkward when relations with many attributes are being considered. When limited to the flat relational case, the expressive power of the language can be seen to be greater than that of Datalog or ILOG without negation, but strictly less than that of Datalog with stratified-negation. When dealing with database transformations we are concerned with *non-recursive* transformation programs only, though our concept of recursion is weaker than that for Datalog ([15]), so that clauses which we do not consider to be recursive could not be expressed in non-recursive Datalog.

Though the language does incorporate some novel syntactic features, it could basically be thought of as an evolution of existing deductive languages: for example it could be considered to be an extension of ILOG ([12]) to the nested relational model, or as a restriction of IQL ([11]) for dealing with structural data manipulations. The most significant contributions of this work are in the way the language is used to express and implement transformations and constraints. In established deductive query languages, though they may involve some degree of rule rewriting and optimization, the rules are basically used to describe the manipulations of data from a database necessary to satisfy a query. Here we are concerned primarily with the manipulation of the rules themselves, and with converting them from a form in which the transformations and constraints are logically specified in a clear and meaningful manner, into a form where the rules can then be easily translated into some efficient database programming language.

3.2.1 Types

Types in our language are given by the following abstract syntax:

$$\begin{array}{rcll} t & ::= & \{t\} & \text{— \textit{set type}} \\ & | & \langle a_1 :^* t_1, \ldots, a_n :^* t_n \rangle & \text{— \textit{record type}} \\ & | & \textit{int} \mid \textit{string} \mid \ldots & \text{— \textit{base types}} \end{array}$$

A set type, $\{t\}$, represents finite sets of values of type t. A tuple type $\langle a_1 :^* t_1, \ldots, a_n :^* t_n \rangle$ represents *tuples* or *records* with required attributes $a_1, \ldots, a_n$, of types $t_1, \ldots, t_n$ respectively: each symbol $:^*$ representing one either $:$, for a required attribute, or $:^{op}$, for an optional attribute. Base types *int*, *string* and so on represent simple atomic values.

A relation is considered to be a set of tuples, so a *relation type* is a type of the form $\{\langle a_1 :^* t_1, \ldots, a_n :^* t_n \rangle\}$ In a *flat-relation type* the types $t_1, \ldots, t_n$ are all base types.

As well as individual relations, we consider databases as a whole to have types. A *database type* is a tuple type with attributes for each of the relations or classes in the database and going to the types of those relations. For example a database type for the database whose schema is shown in Figure 3, with relations such as `STS`, `primer`, `sequence` and `na-interval`, will be a tuple type with attributes `STS`, `primer` and so on, each of which would be of an appropriate relation type.

Our language is strongly typed, in that, given types for the source and target databases of a transformation, a unique type can be inferred for each term in a transformation program.

3.2.2 Terms and Atomic Formulae

The main syntactic elements of our language are *terms*, ranged over by $P, Q, \ldots$, and *atoms*, ranged over by $\phi, \psi, \ldots$. *Terms* represent values in a database, while *atoms* are the basic building blocks of formulae. They are defined by the following abstract syntax:

$$\begin{array}{rcll} P & ::= & \mathsf{Src} & \text{— \textit{source database}} \\ & | & \mathsf{Tgt} & \text{— \textit{target database}} \\ & | & \mathsf{c} & \text{— \textit{constant}} \\ & | & X & \text{— \textit{variable}} \\ & | & a & \text{— \textit{attribute}} \\ & | & P.a & \text{— \textit{projection}} \\ & | & f(P_1, \ldots, P_{r_f}) & \text{— \textit{Skolem function}} \\ & | & P\langle \phi_1, \ldots, \phi_k \rangle & \text{— \textit{compound term}} \end{array}$$

and

$$\begin{array}{rcll} \Phi & ::= & P \doteq Q & \text{— \textit{equality}} \\ & | & P \dot{\neq} Q & \text{— \textit{inequality}} \\ & | & P \dot{\in} Q & \text{— \textit{set-inclusion}} \\ & | & P \dot{\leq} Q \mid P \dot{\geq} Q & \text{— \textit{arithmetic predicates}} \\ & | & \mathsf{Undef}(a) & \text{— \textit{undefined optional}} \\ & & & \quad \text{\textit{attribute}} \\ & | & \mathsf{False} & \text{— \textit{contradiction}} \end{array}$$

Here the Src and Tgt represent the source and target databases in a transformation, which are regarded as tuples of relations. Constants are always of base type, while variables can be bound to sets and tuples as well as to values of base type.

If P is a term representing a tuple, then $P.a$ is the value of the a attribute of the tuple (if it is defined).

A compound term of the form $P\langle \phi_1, \ldots, \phi_n \rangle$ has the same value as the term P but carries with it the atoms $\phi_1, \ldots, \phi_n$ which are to be interpreted relative to P: so any attribute term, a, occurring in one of $\phi_1, \ldots, \phi_n$ (but not in any smaller compound term) is evaluated as $P.a$. An attribute term, a, must occur within some compound term. For example if the variable X represents a tuple in the target relation `STS`, and the attribute term `Id` occurs in an atom ϕ_i of the compound term $X\langle \phi_1, \ldots, \phi_k \rangle$, then the term `Id` represents the `Id` field of the tuple X, or, equivalently, has the same value as that of the term $X.$`Id`.

Atoms are built using the binary predicates $=$ (equality), $\neq$ (inequality) and $\in$ (set inclusion), and the Undef predicates which check for the definedness of an optional attribute of a tuple. The nullary predicate False represents an error situation, and is used in checking the validity of a transformation.

For example an atom $X \in$ `STS` would mean that X is a tuple in the relation `STS`. We could use a compound term to put further restrictions on X:

$$X\langle \mathtt{id} = I, \mathtt{pr1\_primer\_id} = P1, \\ \mathtt{pr2\_primer\_id} = P2 \rangle \in \mathsf{Tgt}.\mathtt{STS}$$

which means that X is a tuple in the target relation `STS` with `id` attribute I, `pr1_primer_id` attribute $P1$ and `pr2_primer_id` attribute $P2$.

3.2.3 Clauses

A *clause* has the form

$$\psi \leftarrow \phi_1, \ldots, \phi_n$$

The atom ψ is called the *head* of the clause, while $\phi_1, \ldots, \phi_2$ form the *body* of the clause.

Not all syntactically correct clauses are meaningful. A clause is said to be *well-formed* for source database type T_{src} and target database type T_{tgt} if it is *well-typed* with respect to T_{src} and T_{tgt}, meaning that all the types of terms occurring in the clause make sense when we take the term Src to have the type T_{src} and

Tgt to have the type T_{tgt}, and it is *range-restricted*. The concept of range-restriction is taken from Datalog ([16]), and means that each variable in the clause is restricted to range over some finite set of values. The formal definitions of these restrictions, together with a more detailed presentation of the semantics of the language, can be found in [17]. All the clauses considered in this paper will be well-formed for the relevant types.

The meaning of a well-formed clause, $\psi \leftarrow \phi_1, \ldots, \phi_n$, is that if, for some instantiation of the variables in the body, $\phi_1, \ldots, \phi_n$ are true, then there is an instantiation of the remaining variables in the head of the clause such that ψ is also true. Clearly the truth of a clause is dependent on the values of the source and target databases for which it is being evaluated. A pair of database values μ and ν, of types T_{src} and T_{tgt} respectively, are said to *satisfy* a clause if it is true when we take the term Src to denote the value μ and Tgt to denote ν.

For example the clause

$$X = Y \;\leftarrow\; X\langle \texttt{id} = I\rangle \in \mathsf{Tgt}.\texttt{STS},\; Y\langle \texttt{id} = I\rangle \in \mathsf{Tgt}.\texttt{STS}$$

says that, for any two tuples X and Y in the relation `STS`, if X and Y have the same value, I, on their `id` attributes then they are equal. In other words the attribute `id` is a key for `STS`. This clause is an example of a *constraint*: a clause which concerns only one database rather than the connection between a pair of databases.

The terms in a clause can be classified as *source terms* which denote values in the source database, and *target terms* which denote values in the target database. A *target constraint* is then a clause containing only target terms, while a *source constraint* contains only source terms. Constraints may be tested after a transformation is carried out in order to ensure the validity of the transformation. Constraints may also play a significant part in determining transformations, and be counted as part of a transformation program.

We will now look at some more examples of constraints for the database shown in Figure 3. Firstly an inclusion dependency, that for every `primer id` in the `STS` table there is a corresponding entry in the `primer` table:

$$\begin{aligned}&X\langle \texttt{id} = P\rangle \in \mathsf{Tgt}.\texttt{primer} \;\leftarrow\\ &\qquad Y\langle \texttt{pr1\_primer\_id} = P\rangle \in \mathsf{Tgt}.\texttt{STS}\end{aligned}$$

Next that each material has exactly one GDB name:

$$\begin{aligned}&X = Y \;\leftarrow\\ &\qquad X\langle \texttt{material\_id} = M, \texttt{lab\_code} = \text{“GDB”}\rangle\\ &\qquad\qquad \in \mathsf{Tgt}.\texttt{names},\\ &\qquad Y\langle \texttt{material\_id} = M, \texttt{lab\_code} = \text{“GDB”}\rangle\\ &\qquad\qquad \in \mathsf{Tgt}.\texttt{names}\end{aligned}$$

And, finally, that a public name cannot be a GDB name:

$$\begin{aligned}&\mathsf{False} \;\leftarrow\; \langle \texttt{public\_name} = \text{“Yes”}, \texttt{lab\_code} = \text{“GDB”}\rangle\\ &\qquad\qquad \in \mathsf{Tgt}.\texttt{names}\end{aligned}$$

Note that the last two of these constraints could not be expressed using the traditional functional and existence dependencies for the relational model.

In determining a transformation between two databases, we are interested in a special class of clauses called *transformation clauses*. A *transformation clause* is one which contains only target terms in its head, and which does not contain any Undef atoms for target terms.

For example, the following is a transformation clause generating part of the `STS` relation of the schema shown in Figure 3 from the data entry screen shown in Figure 2:

$$\begin{aligned}&\langle \texttt{id} = \texttt{f\_STS}(PI1, PI2),\\ &\;\texttt{pr1\_primer\_id} = PI1,\\ &\;\texttt{pr2\_primer\_id} = PI2,\\ &\;\texttt{PCR\_prod\_size\_lo} = SL,\\ &\;\texttt{PCR\_prod\_size\_hi} = SH\rangle \in \mathsf{Tgt}.\texttt{STS}\\ &\qquad \leftarrow\; \langle\langle \texttt{pname} = PN1\rangle \in \texttt{Primers},\\ &\qquad\qquad \langle \texttt{pname} = PN2\rangle \in \texttt{Primers},\\ &\qquad\qquad \texttt{PCR\_prod\_size\_lo} = SL,\\ &\qquad\qquad \texttt{PCR\_prod\_size\_hi} = SH\rangle \in \mathsf{Src}.\texttt{STS\_screen},\\ &\qquad\quad \langle \texttt{pname} = PN1, \texttt{id} = PI1\rangle \in \mathsf{Tgt}.\texttt{primer},\\ &\qquad\quad \langle \texttt{pname} = PN2, \texttt{id} = PI2\rangle \in \mathsf{Tgt}.\texttt{primer},\\ &\qquad\quad PI1 \leq PI2\end{aligned}$$

There are several points about this clause that deserve comment. Firstly notice that the Skolem function `f_STS` is used to generate ids for the `STS` relation. Also although the `STS_screen` relation has only one attribute `Primers`, it occurs in two separate atoms in the description of a tuple in the `STS_screen` relation. This is because the attribute is *set valued* and each of the two atoms asserts the presence of a different tuple in the `Primers` sub-relation.

The body of this clause makes use of the target database relation `primer` in order to look up the `primer_id`'s. The tuples for this relation are in turn generated by another clause:

$$\begin{aligned}&\langle \texttt{id} = \texttt{f\_primer}(PN), \texttt{pname} = PN,\\ &\;\texttt{melting\_temp} = MT, \texttt{pick\_method} = PM,\\ &\;\texttt{date\_picked} = DP, \texttt{strand} = ST\rangle\\ &\qquad\qquad\qquad \in \mathsf{Tgt}.\texttt{primer}\\ &\qquad \leftarrow\; \langle\langle \texttt{pname} = PN, \texttt{melting\_temp} = MT,\\ &\qquad\qquad \texttt{pmethod} = PM, \texttt{date\_picked} = DP,\\ &\qquad\qquad \texttt{strand} = ST\rangle \in \texttt{Primers}\rangle\\ &\qquad\qquad\qquad \in \mathsf{Src}.\texttt{STS\_screen}\end{aligned}$$

We will see in Section 3.3 that it is necessary to unfold clauses like this, in order to get a clause that refers only to source relations in its body and only to target relations in its head. Clauses of this form can be processed in one-pass without referring to the target database.

3.2.4 Transformation Programs

A *transformation program*, from database type T_{src} to database type T_{tgt}, consists of a set Δ of transformation clauses that are well formed for T_{src} and T_{tgt}.

If Δ is such a transformation program, and μ is a database value of type T_{src} and ν is a database value of type T_{tgt}, then ν is said to be a Δ-*transformation* of μ iff, for each clause $C \in \Delta$, μ and ν satisfy C.

A transformation program Δ from T_{src} to T_{tgt} is said to be *complete* iff, for any database value μ of type T_{src}, if there exists a Δ-transformation of μ then there is a *unique smallest* such transformation. The *smallest* Δ-transformation is important because it represents the data generated by the transformation program Δ from the source database: in general a transformation program will imply that certain data should be in the target database but does not exclude other additional data from being in the database as well. If a transformation program is complete then there is no ambiguity about what this smallest transformation is. It is these *unique smallest* transformations that we wish to compute.

We are particularly interested in *non-recursive* transformation programs. These describe transformations that can be done in "one pass": that is, they can be carried out by reading the source database and inserting values into the target database, as opposed to *recursive* transformations in which data which is inserted in to the target database is then used to create more data for the target database. The problem of testing whether a transformation program is recursive in our nested relational model is a little more delicate than the problem for the flat relational model and Datalog. Details can be found in [17].

3.3 Normal Forms

We now limit our attention to the special case of database transformations where the target database is flat relational. In this case, we first convert a transformation program into a *normal form*, which can in turn easily be converted into a program in some (non-recursive) query language.

Suppose our target database contains a relation R. A transformation clause is said to be in *normal form* if it has the form

$$\begin{aligned} X\langle a_1 = P_1, \ldots, a_k = P_k, b_1 = Q_1, \ldots, b_l = Q_l\rangle \in R \\ \leftarrow\ \phi_1, \ldots, \phi_n \end{aligned}$$

where $a_1, \ldots, a_k$ are the required attributes of the relation R, and $b_1, \ldots, b_l$ are a subset of the optional attributes of the relation R; the atoms $\phi_1, \ldots, \phi_n$ contain only source terms and constants; and the terms $P_1, \ldots, P_k, Q_1, \ldots, Q_l$ are built using only variables, constant symbols and function symbols.

A transformation program is said to be in normal form if all its clauses are in normal form.

We have an algorithm [17] which given a non-recursive transformation program for a flat relational target database type, if the program is complete will return an equivalent program in normal form, and if the program is not complete will fail, reporting an error. This algorithm forms the central part of our code generators for transformation programs. If the source database type is also flat relational then clauses in normal form can be directly translated into a join-and-project expression in relational calculus or a "select-from-where" expression in SQL. If the source database is not flat-relational then normal form clauses can be converted into CPL ([18, 19]) or some other suitable query language.

The normal-form clauses are built by combining and unfolding clauses of a transformation, in order to form clauses which provide a complete description of a tuple in the target database in terms of the elements of the source database. Because our transformation programs are not recursive it follows that this process will terminate. If it is possible to build only a partial description of a tuple for some relation, then it follows that the transformation program is not complete.

For example a normal-form clause for the `STS` table in the transformation from the STS data-entry screen (Figure 2) to Ch22DB (Figure 3) formed from the clauses in section 3.2.3 would be:

$$\begin{aligned} &\langle \mathtt{id} = \mathtt{f\_STS}(PI1, PI2), \\ &\ \mathtt{pr1\_primer\_id} = PI1, \\ &\ \mathtt{pr2\_primer\_id} = PI2, \\ &\ \mathtt{PCR\_prod\_size\_lo} = SL, \\ &\ \mathtt{PCR\_prod\_size\_hi} = SH\rangle \in \mathsf{Tgt}.\mathtt{STS} \\ &\qquad \leftarrow\ \langle\langle \mathtt{pname} = PN1\rangle \in \mathtt{primers}, \\ &\qquad\qquad \langle \mathtt{pname} = PN2\rangle \in \mathtt{primers}, \\ &\qquad\qquad \mathtt{PCR\_prod\_size\_lo} = SL, \\ &\qquad\qquad \mathtt{PCR\_prod\_size\_hi} = SH\rangle \\ &\qquad\qquad\qquad \in \mathsf{Src}.\mathtt{STS\_screen}, \\ &\qquad\quad PI1 = \mathtt{f\_primer}(PN1), \\ &\qquad\quad PI2 = \mathtt{f\_primer}(PN2), \\ &\qquad\quad PI1 < PI2 \end{aligned}$$

Notice that this clause gives a complete description of a tuple in the `STS` relation, and does not call on any of the target relations in the body of the clause. In particular the calls to the `primer` relation which were in the body of the clause in section 3.2.3 have been replaced by applications of the Skolem function `f_primer`.

3.4 Transformation Tools

The transformation process is automated by means of code-generators for a variety of database programming languages. Initial implementation efforts are for a code generator for CPL ([18]), since this language has interfaces to SYBASE (the most immediate requirement for Chr22DB) and several other biological data-sources. However the core of the tool is an implementation of the convert-to-normal-form algorithm, and further interfaces to convert normal form programs to other database programming languages can be constructed easily.

In addition tools to read meta-data from various database systems, such as SYBASE, and schema-design tools, such as ER-draw ([13]), and convert it into TSL types and constraints are being developed. These take much of the load of entering constraints off the users, and allows them to concentrate on specifying the substantial part of a transformation. Ultimately we would like to build graphical schema-manipulation tools which automatically generate the relevant constraints and transformation clauses for a schema evolution.

4 Conclusions

The complexity of the data structures involved in Human Genome Project databases, together with the frequency of schema evolutions and the large number of incompatible heterogeneous databases with which data must be exchanged, necessitates the development of new tools and methodologies.

Although much has been written on the subject of schema evolution (see [20]), existing works do not address the problem of performing the corresponding transformations on the underlying data. However it is essential to have all the previously entered data available for the current schema, and consequently these transformations are necessary.

Some of these issues have also been addressed in [21], although in the context of database integration and for a more limited data model. Our proposed language allows us to specify database transformations in a clear and formal manner, and then implement the transformation for a variety of database systems by means of code generators. In addition our language allows for the specification of constraints that arise from Human Genome data which are not representable using established constraint languages.

Related work also includes that of schema merging in heterogeneous databases (see [22, 23, 24, 25, 26]). Central to all of these approaches is the need to have some user manipulation of the underlying schemas to indicate how the underlying databases are related to the merged schema. To our knowledge, there has been no principled, systematic approach proposed to do this other than our proposed constraint language.

There are many areas of future research, some of which we have already indicated, such as the completion of normal form algorithms for target databases which are not flat relational; the implementation of code generators from normal form transformation programs to languages of interest, and the eventual development of a window driven interface for specifying transformation programs.

Another issue is that of *composing transformations*: while some transformations will be applied only once, many transformation programs will be applied repeatedly. The most frequent of these is probably data-entry transformation programs; others involve transformation programs which import data from other archival databases, such as GDB, which are run routinely in order to reflect the continuous updates of the archival databases. We do not want to rewrite these transformation programs every time there is a minor schema evolution on the Chromosome 22 database, hence the need to *compose* the transformations.

We have currently completely specified the data entry transformation, and have partially specified a transformation from an archival genomic database, GDB, to Chr22DB. Our experience is that the approach is extremely useful, since the relationships between structures in the source and target is clearly indicated in the clauses of the program. Knowing first-hand how laborious it was to transform SYBASE code for data entry as Chr22DB evolved, this is an extremely important practical gain.

Acknowledgements: We are indebted to Peter Buneman, Chris Overton and David Searls for their help and advice in developing and presenting these ideas. The really cool postscript diagram in Figure 1 is also due to David Searls.

References

[1] W. Barker, D. George, L. Hunt, and J. Garavelli, "The PIR protein sequence database," *Nucleic Acids Research*, vol. 19, pp. 2231–2236, 1991.

[2] P. Pearson, "The genome data base (GDB), a human genome mapping repository," *Nucleic Acids Research*, vol. 19, pp. 2237–2239, 1991.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–410, 1990.

[4] W. R. Pearson, "Rapid and sensitive sequence comparison with FASTP and FASTA," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 85, pp. 2444–2448, 1990.

[5] National Center for Biotechnology Information, National Library of Medicine, Bethesda, MD, *ENTREZ: Sequences Users' Guide*, 1992. Release 1.0.

[6] M. J. Cinkosky, J. Fickett, D. Nelson, and T. G. Marr, "The restructuring of GenBank," October 1987.

[7] K. Hart, D. B. Searls, and G. C. Overton, "SORTEZ: A relational translator for NCBI's ASN.1 database," *Computer Applications in the Biosciences*, vol. 10, no. 3, 1994. To appear. See also UPenn Technical Report CBIL-9203.

[8] G. C. Overton, J. Aaronson, J. Haas, and J. Adams, "QGB: A system for querying sequence database fields and features," *Computational Biology*, 1994. To appear.

[9] Department of Energy, *DOE Informatics Summit Meeting Report*, April 1993. Available via gopher at `gopher.gdb.org`.

[10] N. Goodman, S. Rozen, and L. Stein, "Requirements for a deductive query language in the MapBase genome-mapping database," in *Proceedings of Workshop on Programming with Logic Databases, Vancouver, BC*, October 1993.

[11] S. Abiteboul and P. Kanellakis, "Object identity as a query language primitive," in *Proceedings of ACM SIGMOD Conference on Management of Data*, (Portland, Oregon), pp. 159–173, 1989.

[12] R. Hull and M. Yoshikawa, "ILOG: Declarative creation and manipulation of object identifiers," in *Proceedings of 16th International Conference on Very Large Data Bases*, pp. 455–468, 1990.

[13] E. Szeto and V. M. Markowitz, "Erdraw 4.0: A graphical editor for extended entity-relationship schemas. reference manual," Tech. Rep. LBL–PUB–3084, Lawrence Berkeley Laboritory, Berkeley, California, 1993.

[14] S. Abiteboul and C. Beeri, "On the power of languages for the manipulation of complex objects," in *Proceedings of International Workshop on Theory and Applications of Nested Relations and Complex Objects*, (Darmstadt), 1988. Also available as INRIA Technical Report 846.

[15] J. D. Ullman, *Principles of Database and Knowledgebase Systems I*. Rockvill, MD 20850: Computer Science Press, 1989.

[16] J. D. Ullman, *Principles of Database and Knowledgebase Systems II: The New Technologies*. Rockvill, MD 20850: Computer Science Press, 1989.

[17] A. S. Kosky, "A language for database transformations and constrains," 1993. Manuscript available from `kosky@saul.cis.upenn.edu`.

[18] L. Wong, "Querying nested collections: A dissertation proposal," August 1993. Manuscript available from `limsoon@saul.cis.upenn.edu`.

[19] V. Breazu-Tannen, P. Buneman, and L. Wong, "Naturally embedded query languages," in *LNCS 646: Proceedings of 4th International Conference on Database Theory, Berlin, Germany, October, 1992* (J. Biskup and R. Hull, eds.), pp. 140–154, Springer-Verlag, October 1992. Available as UPenn Technical Report MS-CIS-92-47.

[20] J. F. Roddick, "Schema evolution in database systems — An annotated bibliography," *SIGMOD Record*, vol. 21, pp. 35–40, December 1992.

[21] S. Widjojo, D. S. Wile, and R. Hull, "Worldbase: A new approach to sharing distributed information," tech. rep., USC/Information Sciences Institute, February 1990.

[22] A. Motro, "Superviews: Virtual integration of multiple databases," *IEEE Transactions on Software Engineering*, vol. SE-13, pp. 785–798, July 1987.

[23] A. Sheth, J. Larson, J. Cornellio, and S. Navethe, "A tool for integrating conceptual schemas and user views," in *Proceedings of 4th International Conference on Data Engineering*, pp. 176–183, 1988.

[24] S. Navathe, R. Elmasri, and J. Larson, "Integrating user views in database design," *IEEE Computer*, vol. 19, pp. 50–62, January 1986.

[25] C. Batini, M. Lenzerini, and S. Navathe, "A comparative analysis of methodolgies for database schema integration," *ACM Computing Surveys*, vol. 18, pp. 323–364, December 1986.

[26] A. Sheth and J. Larson, "Federated database systems for managing distributed heterogeneous and autonomous databases," *ACM Computing Surveys*, vol. 22, pp. 183–236, September 1990.