SCHEMA COERCION: USING DATABASE META-INFORMATION TO FACILITATE DATA TRANSFER

by

Terence Critchlow

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

June 1997

ABSTRACT

As more information becomes available, the ability to quickly incorporate new and diverse data sources into existing database systems becomes critical. Schema coercion addresses this need by defining the mapping between databases as a collection of mappings between corresponding constructs. This work defines a comprehensive schema coercion tool: it transforms schemata into corresponding ER representations, identifies correspondences between them, and uses these correspondences to generate a program that automatically transfers data between the databases. In addition to creating a useful tool, this work addresses the significant theoretical problems associated with resolving representational and semantic conflicts between heterogeneous data sources. The approach advocated by this dissertation associates confidences with correspondences, and meta-information with schemata. This approach has successfully reduced the amount of interaction required to define several coercions, including a complex coercion between diverse genetics databases.

To Heidi

For Heidi

Because of Heidi

Again

Still

Always

ACKNOWLEDGMENTS

The author wishes to acknowledge the tremendous amount of support received from the Utah Center for Human Genome Research. In particular, I would like to thank Peter Cartwright for providing motivation and funding from the National Institute of Health under the Utah Genome Center Grant.

CONTENTS

CHAPTER 1

INTRODUCTION

As computer networks become larger and more information becomes available, the ability to quickly incorporate new and diverse sources of information into existing database systems has become critical for many organizations. Successfully incorporating new data requires resolving representational and semantic conflicts between heterogeneous data sources. Several different approaches have been developed to address this problem, three of that are described below.

The use of federated databases is a popular solution to this problem. In a federated database, a unified view of several independent databases is developed. This view is then used as if it represented a single coherent database. The database interface automatically decomposes queries, sends subqueries to the appropriate local databases, and recombines the results. In theory, federation allows each individual database to maintain its autonomy by allowing arbitrary modifications and the exportation of a possibly restricted view of the schema and data contained in the database to the federation. However, due to external demands, in practice there may be limits on what modifications the federation administrator will allow a local administrator to make and remain part of the federation. Federated databases work well when all participants actively contribute to the federation. For example, when the independent databases are within a single company, or when the federated database represents a catalog containing products sold by several different companies in a related industry.

There are situations in that the federated database approach is not appropriate. In particular, in a large community such as the genetics community, many of the local databases would not export information to the community since either the information has already been submitted to a community database or it is viewed as proprietary. In addition, many of the local databases use proprietary or privately developed database management systems. As such, the local databases have nothing to contribute to, and are often unable to participate in, a federation. The approach often used in this case is to develop specialized conversion routines within the local organization to translate the data stored in the global database into the local format. When either the global or local schema changes, the conversion routine must be adapted to the new format.

The final approach to integrate additional information into a database is schema evolution. This approach is used when the data are required to be represented in the local database, but cannot be expressed in the current schema. In this case, the existing database schema is modified to incorporate the new information. The modification to the database may be as simple as expanding a table by providing an additional field, or may entail a complete restructuring of the database, and a rewrite of many existing applications. Then, the data are either imported into the database using a specialized conversion program, or manually entered.

In all of these solutions, a tool that could aid in data and schema manipulation would be extremely useful. It would reduce the interaction required to define the initial manipulation, the interaction required to adapt an existing manipulation to local schema modifications, and the number of errors in the manipulation definition. In the first case, a way to combine a variety of schemata is required. In the second and third cases, a tool to perform conversions between two, possibly very different, schemata would significantly reduce the amount of work required.

Traditionally, schema manipulation has taken three forms: transformation, integration, and evolution. Converting a schema between different data models, such as between an object-oriented representation and an entity-relationship representation, is described by a transformation manipulation. This manipulation is usually assumed to have been performed before others are applied. Schema integration involves merging two or more separate database schemata into a single global schema. Integration is most often used to combine all of the information contained in several autonomous databases into a single federated database. Schema evolution is the process of modifying the schema of an existing database. Several aids have been developed to decrease the difficulty associated with this process. They ensure the correct translation of the information currently in the database from the old schema to the new schema; however, most restrict the translation operations that may occur.

This work defines and addresses a third form of schema manipulation: schema coercion. Schema coercion is the translation from one database schema, the source schema, into another, the reference schema. Schema coercion can be thought of as a mix of schema integration and schema evolution that is applicable to situations where the schemata of interest are not equal partners. If the desired global schema is known in advance, coercion can be used to solve the schema integration process, by translating each of the local schemata into the global schema. For example, in the Genome Topographer

(GT) [78], a global schema has already been decided upon. The problem facing GT is translating interesting information from local (lab) databases into the GT global schema. Currently, the relevant information is manually selected from the lab schema, and a translator is hand written to resolve conflicts and transfer the data to the GT database. This problem cannot be handled directly by an integration algorithm, since the global schema does not, and should not, include all of the information represented by the local schema. Rather a coercion algorithm must be used to filter out the unwanted information, and convert the desired information to the reference schema. Schema coercion can also be used as a primitive approach to schema evolution by specifying the translation between the old and new schemata. However, it does not address the issue of backwards compatibility raised in Section 3.3.

Much of the current work in schema manipulation has ignored difficult problems such as the ability to determine correspondences between constructs in the different schemata. In particular, the solutions used to address this problem in practice usually require either a trivial equivalence test, or a great deal of real world knowledge. Structural differences between the schemata pose additional practical problems. For example, an entity in one schema may be represented as an attribute or a relationship in another schema, or may be represented by several distinct constructs. Semantic differences between schemata, such as differing types or units of the same attribute, are another aspect that must be considered when manipulating schemata. Many schema integration algorithms have ignored these problems completely, assuming the user has resolved all conflicts before the algorithm is invoked. Few approaches have attempted to address all of these problems simultaneously.

This work addresses some of the unresolved issues in this field. In particular, a heuristic algorithm for determining correspondences between constructs is presented. One of the major advantages of this algorithm is that it has the potential to resolve complex structural conflicts automatically. In addition, the algorithm does not require detailed domain knowledge or user input, although it will use this information if it is provided. Obviously, the more user knowledge provided, the better the coercion will be, but a reasonable initial coercion can often be created without any user input.

Another significant, practical contribution of this work is the creation of SCoP: a consistent, comprehensive tool for schema coercion. Batini [15] observed that all schema manipulation problems can be described in terms of four steps:

1) preintegration: the transformation of schemata into the same data model and collection of additional semantic information

1) schema comparison: determination of interschematic correspondences and conflicts

1) schema conformation: resolution of conflicts detected in the previous step

1) schema merging and restructuring: superimposition of schemata using previously determined correspondences and restructure the result as desired to achieve completeness, minimality and understandability.

In addition to these steps, there is another step that is of practical interest: 5) data transfer. Step 5 is not addressed in most work since it is considered to be theoretically uninteresting. It is, however, of great practical importance since it is the step that actually removes much of the burden from the user. Most current integration tools focus on either step 3 or 4 assuming the others have already been performed. The tool developed to demonstrate the feasibility of the concepts presented in this work addresses steps 1,2,3, and 5 with step 4 being unnecessary for schema coercion.

The next chapter describes three data models and presents the conflicts that may arise manipulating schema represented in these models. Chapter 3 presents work done by other researchers in this area. This work was motivated by the needs of the Human Genome Project, that are described in Chapter 4. The algorithms used to achieve the required functionality are outlined in Chapter 5. Chapter 6 demonstrates using SCoP to transfer data between two databases. Chapter 7 describes the simple test cases used to demonstrate the feasibility of the algorithms presented in Chapter 5, as well as the challenge problems this work was designed to address. Chapter 8 summarizes this work and provides directions for future research.

CHAPTER 2

BACKGROUND

This chapter will provide the background required to understand the previously published work in schema manipulation described in Chapter 3. Schemata must be represented in the same data model before they can be manipulated. Unfortunately, since a schema's native data model is determined by the database management system employed, a translation to a uniform data model is usually a required part of the preintegration step. In order to prevent information from being lost, the uniform data model must be at least as expressive as each of the native data models. Section 2.1 describes the three most common data models, the relational, entity-relationship and object-oriented models, in order from least to most expressive.

Once the schemata of interest are represented in the same data model more complicated problems arise. Section 2.2 provides a detailed description of the types of conflicts that may occur between schemata, and outlines the existing solutions to these problems. As shown, most of these solutions are inadequate and improving them is the focus of current research. These problems are presented in the framework of the schema integration problem, since that is the area where they occur most. The ER terminology used in this section is discussed in Section 2.1.2 and was chosen because the ER model is the most common data model at this time.

2.1 Data Models

2.1.1 The Relational Data Model

In the relational data model everything is represented as a table. A table is set of rows, or *tuples*, which correspond to a single instantiation of the concept represented by the table. Each row consists of a fixed number of columns which are identified by a table unique identifier, and contain values from a well specified domain. Table entries have primitive types and cannot directly reference other tables.

Tables may be combined by performing a *join* operation. In a natural join, equal values stored in columns having the same name in both joining tables are used to associate tuples from the different tables. The result of a natural join is a table whose columns are the union of the columns of the original tables, and whose rows consist of concatenated pairs of tuples, one from each table, having equal values for the shared column. The shared columns are represented only once in the result, since the attribute values are the same in both tables. A simple join can be seen in Figure 1, where the result of joining two simple tables, (a) and (b), is shown in (c).

Using column names to perform joins may result in the semantics of the joined table not being well defined. This problem arises because the meaning of a column name

Student

Id	Name
12	Jeff
52	Heidi
67	Terry

Grades

Id	Grade	Class
12	B	561
52	A+	561
52	A	546

Result

Id	Name	Grade	Class
12	Jeff	B	561
52	Heidi	A+	561
67	Terry	A	546

Figure 1 Simple Join Example

is defined relative to its encompassing table; its meaning does not necessarily extend beyond that scope. Consider the result of joining the **Person** and the **Wine** tables defined in Figure 2. The **age** column in the **Person** table represents the current age of the individual, in years. The **age** column in the **Wine** table represents the average number of years the wine from a particular grape should sit before being opened. The **age** column shown in the **Result** table is meaningless because there is no correspondence between the **age** column in the **Person** table and the **age** column in the **Wine** table. Unfortunately, the relational data model does not provide any way to prevent these types of natural joins from occurring.

2.1.2 The Entity Relationship Data Model

The entity-relationship (ER) model was first described by Chen in [30]. This model is the most successful attempt to create a data model that could encompass the three major data models of that time; the relational, network, and hierarchical data models. These data models were popular because they were implemented by existing database management systems. Unfortunately, translating directly between these data models is a difficult process. However, because transforming between these

Person

Name	Age
Terry	25
Heidi	26
Jim	34

Wines

Grape	Age
Chard	10
Zin	5
Cab	25

Result

Name	Grape	Age
Terry	Cab	25

Figure 2 Meaningless Join Example

implementation based data models and the conceptually based ER model is well defined, the ER model became a popular intermediate representation. Another major advantage of this model is that due to its well-defined semantics all interactions, including existence dependencies, between concepts represented in a particular schema can be easily recognized.

There are two types of constructs in the ER model: *entities* and *relationships*. Entities correspond to "things" in the real world; a collection of similar things is an *entity set*. For example, a collection of the names, addresses, social security numbers, employee numbers, and age of the current employees of a company form the **Employee** entity set. If an entity set is dependent upon another entity set for its existence and its ability to distinguish members within that set, it is called a *weak entity set*. For example, an employee database may also include the first name and age of all legal dependents of every employee in weak entity set **Dependents**. Because the name and age characteristics are not sufficient to distinguish one dependent from another, a member of the **Dependents** entity set requires an **Employee** entity to be associated with it. Therefore, deleting an **Employee** entity requires all related **Dependent** entities to be removed. An instance of a *strong* entity does not require the existence of any other construct. Entities have *attributes* that allow differentiation between different elements of the same entity set. Each attribute represents a mapping from an entity set to a value set thereby associating a value with a particular entity. Attributes usually have semantically significant names, assigned by the schema designer. Unfortunately, the semantics associated with these names cannot be represented in the ER model, and this information is lost.

Relationships model associations between entity sets or other relationships; the latter behave as entity sets in this regard. Relationships may contain attributes associated with a particular instance of the association. Each of the entity sets participating in a relationship performs a specific *role*. For example, the relationship **Marriage** may have two roles; **Husband** and **Wife**. Whereas the general case allows an arbitrary number of entity sets to participate in a relationship in practice, relationships are usually restricted to relate exactly two entity sets. This does not restrict the semantic capabilities of the model, and greatly simplifies analysis of the resulting schema.

A graphical language has been designed to represent the constructs in the ER data model. A box is used to represent entity sets, a double box represents a weak entity, an ellipse represents an attribute, and a diamond represents a relationship. Arcs are used to connect attributes and relationships to entity sets. Names are used to differentiate the different constructs. Numbers or variables may be placed next to the ends of a relationship to represent the cardinality of the relationship with respect to the entity being connected at the numbered end. For example, in Figure 3 (b) marriage is represented as a one to one relationship between people. This implies that a person may be married to at most one other person. If a variable, such as n, is used instead of a number, the function relating the entity sets may be multi-valued. It is not always clear how a particular real world concept should be represented in the ER model. For example, the concept of marriage can either be modeled as a relationship between two people, as in Figure 3(a) and (b), as an entity in its own right as in Figure 4(a), or as an attribute of a person as in Figure 4(b). It is the choice of the schema designer to determine the representation best suiting user needs.

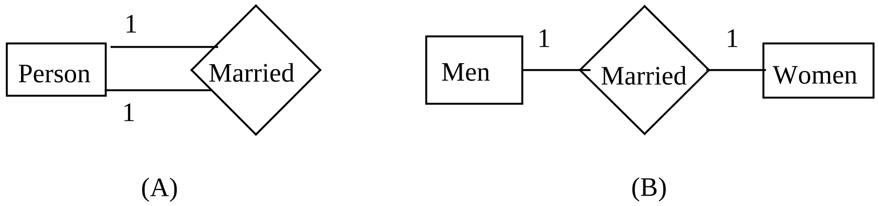

Figure 3 Marriage Between Two People

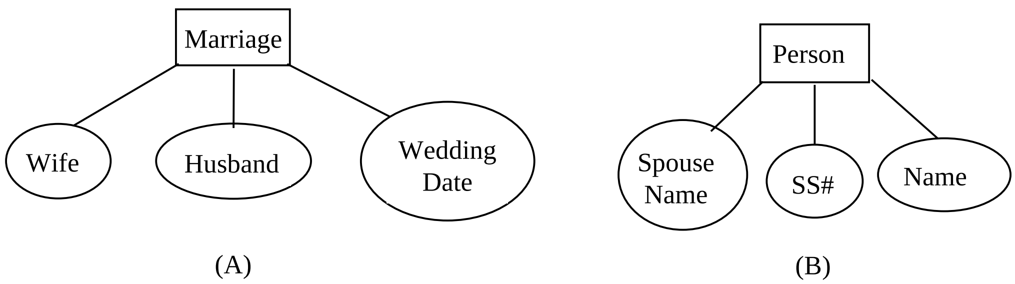

Figure 4 Other Marriage Representations

The concepts of inheritance, generalization and specialization are too important to omit from current data models. However, the ER model is unable to adequately represent these concepts because they do not correspond to the semantics of either entities or relationships. The traditional solution identifies a distinguished relationship, *isa*, to encapsulate the semantics of these concepts. The use of a relationship to represent generalization information complicates the semantics of the model significantly because relationships no longer share a semantic foundation. Whether a relationship represents a traditional or inheritance relationship between two entity sets depends entirely on the name of the relation.

Several extensions to the ER model have been proposed in an attempt to address this problem. One of the most common is the Entity-Category-Relationship (ECR) model proposed by Elmasri in [44]. This data model is fundamentally the same as the ER model, but introduces a new construct, the category, to represent a specialization or a generalization and replace the *isa* relationship used in the traditional ER model. It clarifies the semantics of the model by removing the dual role relationships were required to fill. The category construct is represented as a hexagonal box in the graphical description of a schema. Figure 5 shows an example of a simple schema represented first in an ER format then in an ECR format. Since the semantics of generalization and specialization in the ECR model are better defined than those of the ER model, it is being used more in current database work.

2.1.3 Object-Oriented Model

There is only one construct in the object-oriented (OO) data model, the *object*. An object obtains its structure and behavior from its *class*, which acts as a template for the object. The structure of a class is represented by a set of *attributes*. Depending on the model , the type of an attribute may vary dramatically from primitive values to objects, collections of objects, or any defined first class value. *Methods* are used to define the behavior of an object. Each object represents an instance of a class and models a real world entity. Objects have a unique identity that differentiates instances of the same class

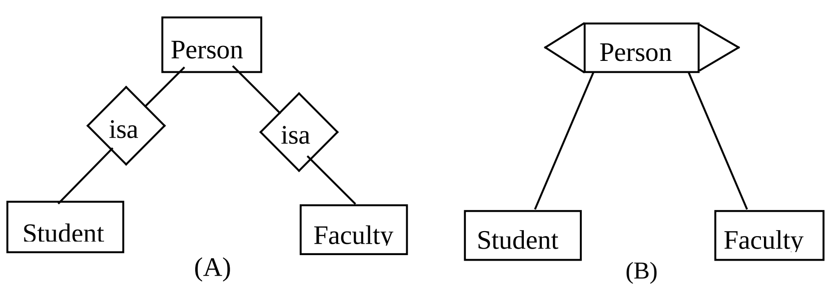

Figure 5 Simple ECR Diagram

with the same attributes. A class may inherit attributes and methods from another class or, in many cases, multiple classes. The subclass can override definitions from its parent class with local definitions, that are passed on to its subclasses.

There are several areas in which the semantics of the OO data model are not well defined. For example, multiple inheritance is not required of a data model for it to be OO. In addition, assuming a model does support multiple inheritance, the inheritance conflict resolution strategy is not consistently defined. Different models often handle this resolution in different ways, making transformation between the models difficult. Attributes are also not well defined, as some data models permit them to be arbitrarily complex, whereas others require them to be either a primitive or a class reference. This lack of consensus is due, in part, to the fact that the OO data model is the newest data model currently in popular use and is a reflection of the diversity of current object-oriented programming languages. Current work [10] is attempting to address this problem, however the standard will require some time to mature before becoming common-place in commercial applications.

2.2 Conflicts

The different problems that arise when manipulating multiple schemata have been documented in several papers including [15] [24] [25] [65]. Naming conflicts arise from the difficulty in identifying similar concepts in different schemata because the database designers may have made different choices in naming the concept. Once the equivalent concepts have been identified, structural conflicts resulting from differences in representation can be addressed. Type and semantic problems may arise because a schema is designed to model concepts only within a specific domain, and the domains of the schemata being manipulated may not be compatible. The following subsections describe each of these problems, and the known solutions to them.

2.2.1 Naming Conflicts

There are two types of naming conflicts: synonyms and homonyms. Synonyms occur when different schemata use different names to represent the same concept. For example, one database may have a entity called **Employee**, whereas another uses the entity **Worker** to represent the same concept. Homonyms occur when different databases use the same name to represent different concepts. For example, an auto racing club may have an entity **Driver** that represents the different people who are qualified to drive cars in different races, whereas a golf club may have an entity **Driver** that represents the different brands of drivers available in the Pro Shop. Naming conflicts arise because the name is the only meta-level information available in most data models, and therefore is the only information used to determine whether two database concepts represent the same real world concept. There are three approaches to solving the naming problem: user interaction, semantic enrichment and expert systems.

User interaction requires the user to explicitly specify which entities are the same. If two entities are not declared to be the same, they are considered different, even if their names are the same. In some cases such as [27], name equivalence is assumed and only constructs with the same name are merged. Name equivalence requires the user to resolve all naming conflicts before the schemata are presented for integration. Obviously, the user interaction approach puts a large burden on the integrators. It requires them to know about every concept represented in the various schemata, and create associations between related concepts. However, it does remove this burden from the programmer and is, as such, a popular option.

Many people, including [29] [61] [62] [83] [110] [116] [126], have suggested ways of enhancing the database schema to contain enough information to resolve naming conflicts. By ensuring the information needed to identify similar concepts is provided with the database schema, the integration can be performed without additional user input. There are three drawbacks to this approach. First, the database administrators must be willing and able to define the semantics of all the databases under their control, and modify these semantics as the databases' evolution requires. The integration process will not produce a meaningful result if this information is not correct or does not exist. Second, a consensus has not been reached on what information is required to perform schema integration, or how this information should be represented. Several alternative specifications have been proposed, but none is sufficient for general integration. Eventually, a set of characteristics capable of describing the information required for most integrations will be identified. Until then, the meta-information associated with a databases will not be understood by different database management systems. Finally, even if the databases record the same information, the real world knowledge must be described using the same vocabulary in all the databases in order for the sharing to be meaningful. This requires close interaction between the various database administrators. Unfortunately, this is not always practical. These limitations seriously restrict the usefulness of this approach, at least while using current technology.

The final approach to this problem consists of creating an expert system to propose likely correspondences between the schemata, and request verification of these correspondences from the user. The expert system can contain domain specific knowledge, as in [40], or can contain more general real world knowledge as in [32] and [25]. Dilts created an expert system that used knowledge of computer integrated manufacturing (CIM) to resolve naming conflicts between different databases within that domain. The problem with this system is that it lacks generality. If the integration application were to be used in any other domain, the expert system would have to be replaced with a new system that had knowledge about the new domain. Obviously, this limits the usefulness of the tool to the domain(s) that it knows about. Collet and Bright took the second approach. Their systems depend on large knowledge bases that contain a tremendous amount of general knowledge. Collet's system [32] is based on the Cyc database, whereas Bright's [25] is based on a specialized implementation of Roget's thesaurus. Both systems have the potential to do well when they are presented with general schemata since they are able to identify similar concepts easily. However, since the knowledge used in both systems is very general, they do not perform well when integrating schema from a field where the terminology is highly specialized and inconsistent. A general problem with the expert system approach is that it does not address the homonym problem: there is no way to establish that entities that initially appear to be similar are actually different.

As can be seen from the approaches above, there is currently no universal solution to the problems associated with naming conflicts. The automatic solutions work well for specific cases, but are not able to handle all of the diverse problems that occur, whereas the manual case puts an undue burden on the user.

2.2.2 Structural Conflicts

Structural conflicts are the result of designers representing the same concept in different ways. There are simple and complex structural conflicts. Simple structural conflicts consist of a one to one mapping from a construct in one schema to a different type of construct in the other schema. For example, integrating schemata containing the different representations of the concept of marriage shown in Figures 4 (a) and 3 (b) would constitute a simple structural conflict. Once all equivalent concepts have been identified, resolving simple structural conflicts is easy. The rules for this resolution, as described in [31], are as follows:

- If two equivalent entities have different attributes, the resulting entity's attributes are the union of the given attributes.
- An attribute can be converted into an equivalent entity, and the implicit association between an attribute and its corresponding entity can then be made into a relationship.
- If two entities are mergeable, but not equivalent, create a new entity whose attributes are the intersection of the attributes of the initial entities, and make each of the initial entities a specialization of the new entity.
- Constructs that are not equivalent cannot be merged. However, constructs that are known to overlap may be combined to form a new construct that contains only the overlapping elements; however the new construct is not represented

as a generalization of the original entities. For example, if there was an entity **Students** in one schema, and an entity **Employees** in another schema, the integrated schema may have an entity representing **Working-students**, as well as entities **Students** and **Employees**.

Following these rules, the integration of the schemata shown in Figure 4 (b) and Figure 3 (a) is easy. The attribute **Spouse** can be converted into the equivalent entity **Person**, and the implicit relationship between the spouse and the person corresponds to the explicit relationship **Married**. Thus, the result of this integration is the schema represented in Figure 3 (a).

Complex conflicts occur when a concept in one schema is represented by several distinct concepts in another schema. For example, consider the schema described in Figure 6 (a) and (b). The **Takes** relation and the **Semester** attribute in the first schema are combined and expanded into several different relations in the second schema. There are well known techniques for the resolution of the common complex structural conflicts. In particular, the expansion of a construct into several equivalent constructs can be performed in the following ways:

- A relationship can be converted into an equivalent relationship-entity-relationship structure by creating a new entity and replacing the original relationship with two relationships that link the new entity to the entities participating in the original relationship. For example, the **Married** relationship in Figure 3 (b) can replaced the entity **Marriage** and relationships **Husband**, linking **Marriage** to **Men**, and **Wife**, linking **Marriage** to **Women**.

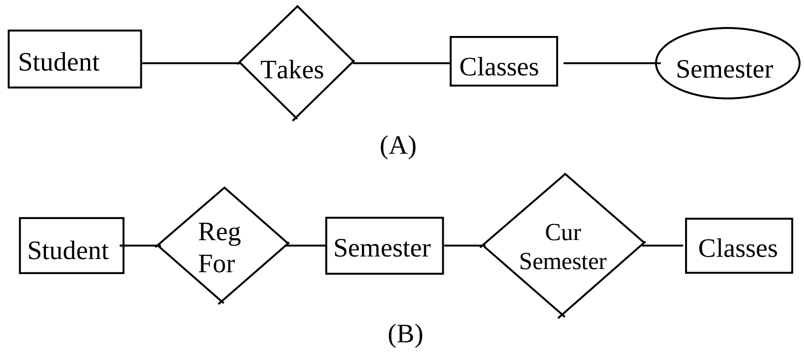

Figure 6 Complex Structural Conflict Example

- An entity can be converted into an equivalent entity-relationship-entity structure. For example, the representation of marriage shown in Figure 4 (a) can be decomposed into the representation shown in Figure 3 (b), with the **Wedding Date** attribute being associated with the relationship **Married**.

Spaccapietra describes a partial solution to the problem of how to integrate additional conflicts in [118] [119]. This solution requires that all equivalencies are known before the integration process begins. The equivalencies are used to manipulate the constructs used in the different schema into the least general representation that will accommodate both representations without loss of information. This solution addresses several of the complicated problems in the area of structural integration successfully, and allows for complicated integrations to take place automatically.

Unfortunately, while this is the best solution to date, it still makes a number of unrealistic assumptions. In particular, it assumes that specialization and generalization are not used in the schema being integrated. This assumption is likely to be violated in most cases, significantly restricting the usefulness of this algorithm. Other assumptions require that references between objects be bidirectional, and that the cardinalities of relationships are precisely defined. While these assumptions are not as unrealistic as the first one, they will still be violated in many existing databases further limiting the usefulness of this solution.

2.2.3 Type Conflicts

Type conflicts occur when the same concept is represented by different data types in different schemata. The difference in types must be the only difference between the concept representations. If other differences, such as different units or domains, exist then the difference is a semantic conflict rather than a type conflict.

Consider different representations of a social security number. It could be represented as an integer or as an alphanumeric string that may or may not contain dashes. If the schemata being integrated have chosen different representations for the shared concept of social security number, a type conflict has occurred. The ability to translate between an integer and a string is trivial; the integer can simply be coerced. The inverse translation is also trivial, provided the string is known to contain only numeric values. However, a slightly more complicated translation is required to provide a translation between an integer and a string of numbers separated by dashes.

Coercions from one data type to a more general data type, such as those outlined in [11], can be automatically performed by the integration program without loss of information. Other coercions can be performed by the integration program, but the results are not well defined. For example, a coercion from a floating point number to an integer could automatically be performed, but this may result in a loss of precision. In addition, the semantics of the coercion could adversely affect the integrity of the result.

For example, if the coercion were to truncate instead of round the data, the result of the coercion could differ significantly. In general, a coercion that may result in loss of information should only be done with the knowledge of the user.

2.2.4 Semantic Conflicts

Semantic conflicts occur because schema designers have different perspectives on the information they are trying to model, and these perspectives may not be compatible. There are two common types of semantic conflicts; unit conflicts and domain conflicts.

Unit conflicts occur when different schemata represent the same concept in different units. This type of problem is common in many different fields. In manufacturing applications, the length of an object may be modeled in inches in one schema, and in centimeters in another. In financial applications, one schema may represent the price of a stock in US Dollars, whereas another uses Japanese Yen. Unit conflicts can often be resolved with a function that converts from one unit to another. However, this function may not always be invertible. For example, consider a grade attribute represented as a percent score in one schema and as a letter grade in another schema. There is a one to one mapping from the percentage grade to the letter score, but the inverse mapping is not one to one since a letter grade will map to a range of percentages. The choice of which representation to use in the final schema must be made by the user due to the tradeoffs involved in selecting one representation over another. The inability to obtain the inverse of the conversion function may pose problems for some applications.

Domain conflicts occur when semantically similar concepts are represented in different domains. For example, the concept represented by a **student** entity in one schema may correspond to graduate students only, whereas another schema may allow both undergraduate and graduate students to be represented in its **student** entity. The **student** in the first schema may correspond to the possibly unidentified concept of **graduate student** in the second schema. This conflict is a result of the initial schema's domain of **student** being restricted to only graduate students, whereas the second schema expanded its domain of **student** to include other, nongraduate students. Whereas the concepts represented by the student constructs are closely related, they are not equivalent, and should not be merged. In this case, the correct result would be to have the initial **student** concept be a specialization of the second concept, with an appropriate renaming.

Detecting when the schemata domains do not match, and what correspondence different domains have to each other is not an easy task. Currently, the only automated solution to the semantic conflict problem is semantic enrichment of the schema, as was proposed as a solution to the problem of structural conflicts. In order for domain conflicts to be resolved automatically, information about the domain, and methods to convert between domains, must be stored with the object. This information can then be used to translate between the different domains in response to user requests. Unfortunately, many of the conflicts are too subtle for most of these methods to catch, and user interaction is required to resolve them. In addition, user interaction is required to provide the conversion routines used to resolve unit conflicts, since these are beyond the scope of the program to provide.

2.2.5 Unsolved Problems

As the previous sections show, there are several unsolved problems in the area of schema manipulation. Current programs are unable to identify similar concepts without some level of user interaction, despite a tremendous amount of work in this area. Expert systems have the potential to perform well, but a single expert system cannot be expected to handle the multitude of specialized areas that databases are used for. Spaccapietra has taken a significant step towards solving the problem of structural conflict, but more work is required to make the solution applicable to the majority of interesting, existing, databases. Finally, reliable detection of semantic conflicts is impossible without a tremendous amount of user input. Until these problems are resolved, the goal of a fully automatic schema manipulation tool is unattainable. However, the creation of reliable manipulation tools that meet the needs of most users will be possible if solutions can be found to even some of these problems.

CHAPTER 3

PREVIOUS WORK

Most of the work done in the field of schema manipulation has focused exclusively on either the practical or the theoretical problems. Previous work in the area of schema manipulation has concentrated exclusively on either the practical or theoretical aspects of the problem, ignoring the other aspect completely. The theoretical work has focused on resolving a particular type of conflict inherent in schema manipulation. However, in such cases, to obtain acceptable results, the schemata typically must be presented in a specific, often obscure, format leaving all other types of conflicts for manual resolution. The practical work endeavors to provide a tool that will aid the user in resolving conflicts, but does not attempt to automatically address any of the conflicts that arise. Very few works [80] provide a comprehensive approach that attempts to address all of the issues in a coordinated manner.

This chapter provides a detailed description of the three most common forms of schema manipulation, and describes the unique problems arising in each of them. Various approaches to solving these problems are discussed. When appropriate, the tools that have been developed are discussed and compared.

3.1 Schema Translation

Schema translation occurs primarily in two situations. The first is when integrating two or more autonomous schemata into a single consistent schema. In this case, the translation is a preprocessing step that is required to be invertible since queries made on the integrated schema need to be translated to the original schemata for processing. The second situation is when a conceptual schema is translated into an implementational representation; for example when converting from an ER design into a relational database system. This does not require the translation to be invertible since a mapping from the resulting schema to the original schema is not required. In both of these cases, the goal of the translation is to ensure that the semantics[1] of the resulting schema are as close as possible to those of the original.

Unfortunately, a semantically lossless translation is not always possible since some data models are able to represent concepts that cannot be accurately represented by others. In general this happens when translating from a semantically rich model to a less expressive model. For example, the relational data model is not capable of expressing the complete semantics of an inheritance hierarchy described in an object-oriented data model, even though it is possible to approximate these semantics. The relational model simply does not have the capability to express all of the implicit information. Therefore, information may be lost when converting from an object-oriented model to a relational

---

[1]The semantics of a schema refers to all of the interactions between various concepts in the schema. This includes, but is not limited to, the schema's dependencies and constraints.

model. If the target data model has at least the same level of expressibility as the original data model, a lossless translation can generally be made.

Most conceptual schema designs use a semantically rich data model such as the ER model. Unfortunately, since most commercial databases use the relational data model to represent the database, the translation from conceptual design to implementational representation may result in a loss of semantics. Whether this loss is significant or not depends upon the application, and the constraints placed upon the users of the database. External sources of constraints, such as application program interfaces, may be used to enforce design semantics the implementation model cannot. Fortunately, object-oriented database management systems provide a semantically rich data model as their implementation model. This enables lossless translation from a conceptual model to the implementation.

Put [97] recommends the use of an extended entity-relationship (EER) model to serve as the target model for preprocessing translations. This recommendation was made, in part, because the EER model is capable of modeling all the concepts that can be represented by other data models. Thus, translations from other data models to the EER model will be lossless. In addition, there are well known algorithms to convert a schema represented in the EER data model to the implementation based data models. Whereas a translation from an arbitrary schema represented in the EER model to a less expressive data model may result in semantic loss, a translation from the implementation data model to the EER model will have a lossless inverse translation. This discrepancy occurs because the translation from the original model to the EER model restricts the corresponding EER schema to contain only those EER constructs that have correspondences in the implementation model. Therefore, inexpressible semantics are prevented from entering the resulting EER schema, and a lossless translation into the implementation model may be generated.

3.2 Schema Integration

Schema integration is traditionally separated into view integration and database integration. Database integration is a superset of view integration that requires addressing additional conflicts. For example, semantic conflicts rarely arise in view integration since the data are stored in the same representation, but are a constant problem in database integration. Since any solution to the database integration problem also addresses the view integration problem, only database integration will be considered further.

Schema integration is the process of combining several distinct schemata into a single, unified schema that represents all of the information available from the original schemata. Schema integration algorithms usually integrate only two schemata at a time for simplicity. This does not restrict the generality of the algorithm since the resulting schema can be used as an input to another integration process. Whereas different integration orders may produce structurally different schemata, they will be conceptually equivalent. Assuming all conflicts have been resolved, integration can be regarded as a superimposition of the schemata onto each other, with similar concepts being unified. In addition to the conflicts described in the previous chapter, the following problems arise in schema integration: semantic preservation, query processing, and instance integration.

Semantic preservation requires that the integrated schema maintain the semantic information inherent in the original schemata. The integrated schema should not introduce relationships, dependencies, or constraints not present in the original schemata, since this may compromise the original semantics. However, the integration of several schemata may produce an association between entities and relationships not visible within any individual schema. These associations represent semantics implicit within the collection of schemata. After the integration is complete, the user may introduce additional interschemata relationships, dependencies and constraints, since these would presumably provide additional, nonconflicting semantic information about the integrated schema.

In order to form queries on the integrated schema, the set of local schemata associated with each global construct must be available. This is usually done by associating relevant information with the global construct during integration. Global queries are sent to a global query manager that is responsible for identifying the local databases which may participate in the query based on references to global constructs. Once the participating databases have been identified, the query manager decomposes the original query into a set of queries which are sent to the local databases. Reformulating this query requires identifying the structures of interest in the local database, resolving semantic differences between the schemata, and translating the global query into the appropriate data manipulation language. For example, a constant used in the a global query may have to be converted to a different value for a set of local databases. Once the local databases return the subquery results, the global query manager again resolves structural differences, type and semantic conflicts before combining the results and returning the answer.

Instance integration deals with a practical problem: when data contained within different databases are merged, how do you identify which *instances* of a concept are equivalent? At first glance, this seems trivial; equivalent instances have the same key and attribute values. However, there is a significant problem with this approach. Different schemata do not necessarily have the same attributes associated with a concept and may use partially, or totally, different keys. For example, a customer database may use **phone-number** as a key to its **Client** concept, and an employee database may use **employee-number** as the key to its **Employee** concept. An integrated schema may have a single, generic concept, **Person**, that includes all clients and employees. It is not clear how to recognize that a particular client is also an employee. Identifying identical instances is not possible in the general case.

Partial solutions to the instance integration problem are addressed in several papers. Larson [69] requires the user to specify correspondences between attributes and uses corresponding attributes to determine the integration strategy. Lim [73] requires the user to enter real world information about the items being modeled, and uses this information to determine if the instances represent the same real world object. Unfortunately, these solutions require a significant amount of user interaction. Wang [128] uses a large knowledge base containing real world information about objects to heuristically determine if the instances are the same. This approach requires less user interaction than the other approaches if the knowledge base knows about the concepts being integrated, but still requires significant input if the object has not been previously encountered.

Another potential problem with data integration is an attribute that is supposed to have the same value in different databases in fact having different values. This may occur because the attribute has a subjective or variable value, such as a hotel rating, or because one of the databases has obsolete information, such as an old address. In the solution proposed by Lim [73], the confidence in each version of the data is obtained from the user. When a conflict is identified, the data with the highest confidence are used. Unfortunately, in addition to being probabilistic in nature, this solution is limited in scope, and will not work when the confidence in the information is unknown or varies depending upon unrecorded criteria. For example, hotel ratings within a single database may be very reliable if evaluated by one person, but unreliable if evaluated by anyone else. A single confidence value is unable to incorporate this meta-information. The problem is aggravated further if the reviewer's identity is not stored in the database, but must be identified through indirect methods such as writing style.

Significant work has been done in the area of schema integration. Batini provided a survey of several schema integration programs and problems in [15]. Biskup provided a formal description of the integration process, including desired semantics, in [19]. Several others [14] [33] [60] [112] [115] have refined the problem and created simple tools. Much of this work has focused only on the practical problems, leaving the theoretical problems largely unsolved. As a result, it is usually the user's responsibility to resolve all conflicts before the integration begins. Failure to do this will result in the integration producing incorrect results. For example, it is often required that the user describe all relationships between the constructs in the schemata being merged. Five types of relationships may be used: equal, contained in, contains, overlapping, and disjoint but mergeable. The first three types of relationships are self explanatory. The overlapping relationship is used to identify when two construct represent similar concepts, but an particular instance may be represented in both constructs. For example, a person may be both a student and an employee. The disjoint but mergeable relationship is used to create a generalization of two constructs, when it is not possible for any instance to be represented in both constructs. For example, undergraduate-students and graduate-students are a disjoint set that may be specializations of a student class, even though a person cannot be both a graduate and an undergraduate student at the same time. If a relationship between two constructs is not specified, they will not be integrated. Forcing the user to resolve all conflicts manually is an unnecessary burden.

Some of the tools developed recently have started to address this shortcoming. Both Bouzeghoub [22] and Gotthard [52] describe systems that perform simple manipulations in an attempt to overcome structural conflicts in the schemata being integrated. Gotthard's tool also reduces the burden on the user by assuming name equivalence on attributes, and a heuristic match for entities based on the percentage of similarly named attributes, to help resolve name conflicts.

Buneman and Kosky also assume name equivalence in [26] [27] [66] [67]. However, the interesting contribution of this work is their approach to solving attribute type conflicts. In this solution, corresponding attributes having different types in the original schemata will appear in the integrated schema as an attribute whose type is the generalization of the original types. As a result, the integrated schema contains all of the information contained in the original set of attributes within a single attribute. Previous approaches required the existence of multiple attributes to represent this data. This approach works well when the type conflict is the only conflict that exists between the attributes. If other conflicts are present, however, the resulting attribute may not have a well-defined semantic meaning. For example, if one database represents prices using an integer value in Canadian cents, and another database represents prices as a fixed point number in US dollars, the semantics of a generalization of these attributes are not clear. A practical problem also arises with this approach since relational databases will not allow attributes to be represented by generalized types.

3.3 Schema Evolution

Schema evolution refers to the modification of an existing database schema, usually in response to new requirements. Whereas the existing schema was presumably designed to meet current requirements, as the database is used shortfalls of the design and new requirements may be identified. This may be the result of an incomplete initial description of the requirements, new application needs, or an evolving understanding of the concepts represented in the database. Whereas modifications required because of the first reason are facilitated by a flexible design, there is little the database designer can do to minimize the impact of changes required by the last two reasons. Since these reasons reflect the usefulness of the database, evolution is considered an important step in its life cycle.

There are three ways to approach this problem: ignore it, completely redesign the database, or modify the existing schema. Obviously the first approach, while the easiest, is not satisfactory. In fields such as genetics the understanding of the data evolves at a rapid rate. Ignoring this changing understanding will cause the database to become obsolete, and unusable within a very short period of time. The second approach may be necessary when the new understanding of the data is very different from the original; however it requires a major investment of time and energy. In most cases, redesigning the schema from scratch, and converting the existing information to the new schema, is a daunting project. The final approach is the most common and most desirable, since an affordable amount of effort is expended, yet user requirements are satisfied. This is the approach discussed throughout the remainder of this section.

The major problem in evolving a schema is maintaining compatibility with existing applications. An existing database usually has many application programs that rely on the current database schema to execute correctly. Depending on how the schema is modified, these programs may produce incorrect results or generate run-time or compile-time errors. There are two conflicting opinions on the impact schema evolution should have on these programs.

The first opinion is that existing applications should work without any modifications. This approach, commonly known as *versioning*, can be further refined as to whether affected applications work on all the data, or only on data in the database prior to the schema modification. The inability of existing programs to access new data has obvious restrictions and is not considered further. In order for existing applications to access all the information in a database, conversion functions must be provided between the new and old schemata. These functions allow the database management system to automatically convert the data to the required schema as described in [84] [99]. In [74] [124], views are used to reorganize and restrict the information presented, whereas the physical schema remains constant. This approach has the advantage of not requiring database restructuring after every schema change. However, the schema modifications are limited since the new view cannot contain more information than is stored in the database schema. In versioning the user is required to define a transformation from the old schema to the new schema, to enable new applications, and queries to access all of the information in the database.

The other opinion is that existing information should be converted to the new schema and affected applications should be rewritten. The justification for this approach is that the old schema is outdated and existing programs will not produce meaningful results with respect to the new interpretation of the data. Unfortunately, there is little support for identifying the applications that are likely to be affected by a particular schema modification. Some systems, such as [12] [53], require the conversion from the old schema to the new take place at once. This approach has the benefit that queries respond quickly since the database management system is not required to perform transformations between versions. It also simplifies the enforcement of semantic constraints since the entire database is represented by a single schema at all times. The major disadvantage of this approach is the amount of time required to perform the transformation on a large database. This is particularly troublesome because the database is often unavailable while the transformation is occurring. Other systems [21] [90] [109] allow dynamic, lazy transformation. The data are converted from the old representation to the new upon its first access, then stored in the new format. This approach does not require the database to be isolated during the transformation, since it is spread over many queries. Another advantage is that unused data are not converted, hence archived information will remain in the original format. There are two disadvantages to this approach. First, queries on old data will take longer than normal because of the transformations that must be performed. Second, it may be impossible to maintain semantic consistency because required information may not be stored in the most current representation while the transformation is occurring.

In order to automate the evolution process as much as possible, many systems restrict the transformations permitted. The semantic implications of performing any of a set of well-defined operations on a database schema can be embodied in a program. The program is then able to perform these transformations automatically. Typically, the following operations are provided: add a construct; delete a construct; change the type of a construct; modify a construct; and change the relationship between constructs, usually by modifying the *isa* hierarchy. For example, transforming an attribute of an entity into a separate entity with several attributes is easily automated. A new entity is created with the desired attributes, instances of the entity are created and the attributes are filled in, a relationship is formed with the old entity, and the attribute is deleted from the old entity. The values for instance attributes may be obtained from the old entity, default values, or a user defined calculation. Evolving a schema is explicitly decomposed into a sequence of these well-defined operations, then the restructuring is performed by a program stepping through these transformation and modifying the database accordingly.

However, by restricting the transformations performed to a small set of all possible operations, some restructurings are impossible to obtain. For example, information spread across several constructs cannot be arbitrarily rearranged, and context sensitive changes to existing data, such as changing the value of a column based on another value, cannot be made. An alternative solution, discussed in [71], allows for arbitrary restructuring of a schema by replacing the limited set of transformation operators with a simplified programming language. This language allows a variety of transformations, including the complete restructuring of the database in the limit. This increased flexibility makes this solution much more practical than previous solutions.

CHAPTER 4

MOTIVATION

As shown in Chapter 3, there is significant work occurring in the field of schema manipulation, most of which is focused on the areas of schema integration and schema evolution. Schema coercion has been ignored in current computer science research, because it is considered a subset of the integration problem. However, in disciplines such as genetics there is a desperate need for a solution to the schema coercion problem. This chapter describes the requirements of the genetics community, which are the primary motivation for this work. Section 4.1 provides an overview of the Human Genome Project (HGP), focusing on the problems with the current data organization. Section 4.2 discusses the Genome Topographer project that is a prominent attempt to address these problems. The database environment and research requirements of the Utah Center for Human Genome Research are presented in Section 4.3.

4.1 The Human Genome Project

The ultimate goal of human genetics research is to manipulate genetic structures to overcome undesired predispositions, such as a predisposition to breast cancer. The Human Genome Project [47] is an ambitious first step in achieving this goal. It is an attempt to completely sequence, map and annotate the human genome. This goal is composed of three distinct projects, that are currently proceeding in parallel. First, the complete sequence of nucleotides making up the human genome must be identified. Second, the portion of these sequences corresponding to coding regions, or genes, must be identified. Third, the association of a specific function or functions to every gene must be made. Once this step has been completed, the researchers can focus on other, more challenging problems, such as how to modify the genes to prevent or mitigate certain diseases. In addition to the researchers concerned only with the human genome, many geneticists are interested in the close correspondences between human genetic coding regions and similar structures found in other creatures, such as mice or bacteria. These correspondences, known as homologies, allow researchers to determine the effects of a gene by a wider range of controlled experiments than is appropriate for human subjects.

The HGP is a huge project, consisting of thousands of researchers, spread across hundreds of labs. In order to distribute the information generated at all of these labs, some of the larger labs host community databases. Each community database is responsible for maintaining a subset of the genetic information available for a set of model organisms. For example, the Genbank database contains sequence information for several organisms including mammals, mice and bacteria. Researchers from all over the world are encouraged to use these facilities, and to contribute their findings to the database curators so the information can be kept up to date. In order to encourage submissions, many journals now require the underlying data be reported to the relevant database administration before an article may be published. In order to maintain the confidentiality of unpublished submissions, the data are not released until it appears in a published article. This is an extremely important consideration since prematurely releasing the data could have serious repercussions for the researchers who submitted it.

Since most of the contributed information is obtained through indirect observation and statistical analysis, there may be contradictory pieces of information on any subject contained within the database at any time. This problem is compounded by the natural variation in individuals caused by mutations. For example, there may be several opinions as to the correct location of a particular gene, or the correct sequence for a particular region. The current approach is to enter all of the data, with attributions, into the database and allow individual researchers to determine which information to use.

Community databases rely heavily on their database management software to maintain the integrity of the data while providing service to multiple users concurrently. These requirements are too stringent to be met by independently developed systems consequently most community databases use commercial relational or object-oriented database management systems. The semantics of the database are usually represented, directly or indirectly, in external documentation to prevent ambiguous terminology from causing undue confusion. Due to the dynamic nature of the field, the database schemata are forced to evolve along with researchers' rapidly changing understanding of the genome. However, this evolution is relatively slow in the case of the community databases because of the number of users affected by such a change. Unfortunately, by reducing the frequency of schema modifications, the usefulness of the database is also reduced, since the current representation quickly becomes obsolete.

Due to the complexity of the database schema, and the need to ensure the data are as consistent as possible, most community databases do not allow direct interaction with the DBMS. Instead, new submissions and modifications are directed to a curator, who is responsible for ensuring they are correctly entered into the database and appropriately attributed. This prevents accidental and malicious manipulation of the data by researchers. Requests for information contained within the database are usually handled through programs or Web interfaces. Some programs, such as BLAST, execute sophisticated retrieval algorithms far beyond the capability of basic SQL interfaces. In some cases special purpose queries that cannot be handled by the normal interface are required. To accommodate these requests a flat file representation of the database is usually provided. This duplicate representation allows arbitrary manipulation of the data without affecting the integrity of the community database. The format of this representation varies from a set of ASCII table dumps (GDB) to a binary ASN.1 file (Genbank).

Despite the critical role played by the community databases. the importance of the smaller laboratory databases must not be underestimated. However, due to the huge differences in functionality found in these databases, integrating them with the larger databases can be difficult. Some are represented as flat files with minimal structure and no semantic constraints. Others use object-oriented database management systems, and have well defined schemata and semantics. Some labs have large informatics groups to maintain the database and associated applications, others require the geneticists do it themselves. A common requirement of laboratory databases is the ability to rapidly evolve the database schema as the researcher's view of the information changes. Unfortunately, these changes often require transformations beyond the simple reclassification strategies accommodated by most database evolution facilities.

In addition to local databases, geneticists require access to several of the community databases. Currently, this access is achieved through the execution of several distinct programs -- one for each database of interest. Since these interface programs are supplied and maintained by the community database administrators, new implementations, corrections, and enhancements must be obtained from them. Unfortunately, these programs are not customized for individual labs. If the data are to be manipulated further, or combined with data from other sources, the lab is required to coerce the data into a useful format, usually by importing it into the local database. There is no way to query all of the community databases simultaneously, nor is this likely to become practical in the near future.

To complicate matters further, there is no standard terminology within the genetics community. Even within a small community of researchers, such as the human sequencing community, there are subtle differences in terminology. The differences in terminology between different communities is formidable. This problem manifests as a plethora of naming conflicts when integrating databases. Unfortunately, these naming conflicts cannot be resolved by most existing integration programs, because the distinctions are too subtle: resolving these conflicts requires an expert familiar with all participating databases.

4.2 Genome Topographer

Once the human genome has been sequenced, the challenge becomes identifying the genes and their functions within the sequence, and determining interactions between coding regions. In order to formulate interesting queries on the data, it must be presented in a consistent format with well-defined semantics. This is the problem Tom Marr is attempting to address with the Genome Topographer (GT) project. The idea behind GT is that there is a limited amount of information of interest to the genetics community as a whole: most of the information stored in individual lab databases is not required to answer the important questions facing researchers at large. The concepts contained within GT, have been carefully selected and precisely defined to ensure they are of general interest. As a result, the human genome information representable by GT has been restricted to where it can be contained within a single database server. This permits a single, uniform interface to access all of the interesting information. The alternative to restricting the representable data, creating a federation of related databases, is a formidable task. The GT database currently includes the information stored in multiple community databases and larger laboratory databases. Unfortunately, it is currently unclear whether the GT representation is too restrictive to remain useful over the long term, or whether the amount of *interesting* information will eventually be too large to be represented within a single DBMS.

Information is imported into the GT database using hand constructed parsers, which read the desired data from the original database and import it to the GT database. All conflicts between the lab and GT databases are resolved by the parser. If either of the schemata change, the parser must be modified to reflect this change. A new parser must be written for every database whose information is to be imported into GT. The amount of effort expended in making a parser efficient is determined by the size of the database and the number of times the parser is expected to be used. Since the parser may be used to keep the GT database up to date, the number of times it is expected to be used is dependent upon the importance of the original database, and how often its schema and data are expected to change.

The importation of information from other databases allows a single GT query to search all of the available, collected information. However, there is still the problem of converting the foreign databases into the GT format, since hand coding a parser for every database is expensive. This problem could be solved by a schema coercion program. The foreign database schema could be coerced into the GT format semiautomatically. This process would allow the user to resolve the subtle conflicts manually, while the obvious translations are performed automatically. The automatic generation of a translator from the local schema to the GT schema would provide a way for the information to be transferred between databases with minimal additional effort. Changes in either schemata could be modeled within the coercion program, providing a simpler interface than manual code modification.

4.3 The Utah Center for Human Genome Research

The primary goal of the Utah Center for Human Genome Research (UCHGR) is to develop high throughput sequencing technology. To test this technology, the center is attempting to sequence and annotate the entire Pyrococcus genome, approximately 2 million nucleotides, within a 3-year period. Pyrococcus is a hyperthermophilic archea -- a bacterium that resides in volcanic ocean vents. It is of great interest because of its ability to produce stable proteins at temperatures exceeding one hundred degrees. After the Pyrococcus genome has been sequenced, UCHGR hopes to utilize its technology to sequence human chromosome 17 which is rich in suspected disease related genes.

The new sequencing technology demands that the UCHGR informatics group solve the practical problem of accurately representing a rapidly evolving view of the data.

Rather than constantly revising a traditional relational schema, a new data model was developed [104]. This model identified five basic concepts of laboratory management: objects, relationships, processes, environments, and protocols. Instances of these concepts comprise objects normally representing concrete entities within a genetics laboratory: wells, sequences, microtitre dishes etc. The basic concepts were then represented within a Sybase RDBMS, along with meta-information about the primitive objects. This model greatly simplified the evolution process by allowing the Sybase data manipulation language to manipulate the conceptual schema using ordinary transactions on the meta-data. This flexibility has been crucial to the informatics group's ability to quickly adapt to the geneticists' evolving view of the data.

As the sequencing phase of the project nears completion, UCHGR needs to compare their Pyrococcus data with similar data from other organisms. In order to do this, they need to convert data from the community databases holding this information into the UCHGR format. Complicating the coercion process are the dramatic differences between the UCHGR Sybase schema and the schemata of the community databases, given the former's innovative data model. It has been estimated that creating a program to convert an interesting subset of the data from a single community database to the UCHGR database format would take one person month. Modifications to import a different data set are estimated to require one person week.

While the effort required to define and maintain a single coercion between databases is not overly onerous, the creation and maintenance of a large number of coercions is extremely burdensome. Unfortunately, this is the situation at UCHGR. Data transfers are performed on a regular basis to ensure UCHGR database remains current.

As a result, these coercions must be modified every time either the community or UCHGR database schema changes. In addition, as the UCHGR project progresses, new information is required from, and additional information is distributed to, the community databases requiring the definition of new coercions. Creation of a program that can define and modify these coercions without significant user interaction has been the major motivation of this work.

CHAPTER 5

CONCEPTUAL DESIGN

The theoretical concepts this work is based on have been used to develop the Schema Coercion Program (SCoP). This chapter describes these concepts in detail. Whereas this description uses the SCoP implementation to outline how problems were addressed, it is important to differentiate between the fundamental concepts and the implementation details. Where applicable, the needs of UCHGR have influenced design decisions. The next section presents the terminology used throughout this chapter. Section 5.2 defines the problems addressed by this work. The remaining sections present the approaches taken to overcome these problems.

5.1 Terminology

Schema coercion involves mapping from one database, the **source** database, onto another, the **reference** database. Usually, this requires translating the schemata from their native data models into a **uniform data model**, then manipulating the new schemata. Schema coercion can be viewed at three levels of detail: as a single mapping between two schemata, as a set of mappings between constructs, or as a sequence of mappings onto reference attributes. To minimize confusion about which level is discussed, different terms are used. A **coercion** refers to the mapping from the source database to the reference database as a single function. A coercion consists of several

conversions that define the mapping between two constructs of any type, one from each schemata. A conversion is represented as a set of **transformations** that define the mapping from the source database to the reference construct's attributes. A **translation program** is an executable instance of a coercion. It is responsible for transferring data between the source and reference databases.

5.2 Problem Statement

Schema coercion is the process of mapping between concepts in the source database and corresponding concepts in the reference database. Source concepts which do not have correspondences in the reference database do not participate in the coercion. In order to perform a coercion, both the source and reference schemata must be known in advance, and are considered to be fixed. Schema coercion is a sequence of four distinct steps. First, the schemata of interest are located and transformed into a uniform conceptual data model. Second, participating source constructs are identified. This may require resolving naming and structural conflicts. Third, semantic conflicts are resolved. Fourth, a translation program is generated to transfer data from the source database to the reference database. In general, automatically generating a coercion between arbitrary databases is an unsolvable problem due to the amount of unrepresented semantic information required to resolve conflicts.

An interesting and useful solution should successfully address each of the steps involved in schema coercion. First, the schemata involved in the coercion should be automatically transformed into the desired uniform data model. Forcing manual entry of complex schemata is an unnecessary burden, and is likely to introduce errors. Second, conversions should be automatically created. By identifying obvious correspondences between the schemata a reasonable matching heuristic can create most of the desired conversions between typical schemata. Since heuristics are not able to identify all correspondences, the ability to manually create conversions must also be provided. Third, creation of arbitrarily complex transformations should be possible. Limitations on transformation complexity impede productivity, and may result in the desired coercion being inexpressible. Ideally, the power of a fully functional programming language should be available. Fourth, a mechanism to perform the data translation between the databases should be provided. Whereas identifying and specifying the required conversions is an important step, a practical system should also generate a program to execute the coercion. Failure to do so places a significant and unnecessary burden on the user.

In addition to the problems associated with generating individual coercions, a practical solution should reduce the interaction required to perform a series of coercions. Two approaches may be taken to reduce repetitive interaction: logging and annotating. Logging requires identifying the conversions created for a particular coercion, including manual modifications. After that coercion is complete, the information can be used to create identical conversions in another coercion. This is useful when several similar schemata are to be coerced to the same reference, or a prior coercion must be redefined due to evolution of the source or reference database. Annotating consists of associating meta-information with a database. Whereas significant user interaction is required to create an annotation, coercing sufficiently annotated databases does not require any interaction. If a database is involved in several coercions, annotating may significantly reduce the amount of interaction required to obtain the desired conversions.

This work provides a partial solution to the schema coercion problem. In particular, it demonstrates the feasibility of semiautomatic coercion between heterogeneous database systems. This constitutes a significant contribution for both theoretical and practical reasons. Most of the theoretical problems in schema coercion have corresponding problems in schema integration. Solutions to these problems are important because of the need to unify distributed databases into a single conceptual framework. In addition, schema coercion is a pervasive problem that has not been adequately addressed. In domains such as genetics there is a tremendous practical need for a comprehensive tool that reduces the amount of user interaction required to move data between databases. This work addresses both the theoretical and practical needs.

5.3 Database Interaction

The extended ER model described in Section 2.1.2 was chosen as SCoP's uniform data model for three reasons. First, it is a semantically rich data model. This implies transformations from other data models can be lossless. Second, it is a well-studied model with precisely defined semantics. Algorithms for mapping between this model and others are widely available. Third, the graphical interface allows the schema and associated semantic information to be presented in an understandable format.

A useable tool must be able to interact with database management systems in two different ways. First, it must be able to read the database schemata's native data models and convert them to the uniform model before being presented to the user. Automatic translation is easier, faster and less error prone than manually specifying the schemata. Second, it must be able to create a translation program to perform the desired manipulations. This requires producing code in the database's data manipulation language. Unfortunately, interacting with a database requires significant meta-information, much of which is implicit. For example, the data manipulation language is specific to the database being manipulated. Since each database system represents its meta-information differently, interacting with a new database system requires defining a new interface. Currently, SCoP recognizes four database systems: the Sybase RDBMS, the UCHGR database, ASN.1 binary files, and flat files. The next subsection describes these databases and the motivation for selecting them. The following subsection describes how the different schemata are read and converted into the ER model. Finally, generating code to manipulate the data is discussed.

5.3.1 Recognized Database Systems

Sybase is a commercial relational database management system with a client/server interface. It was selected because it is representative of the relational data model, and is available at the university. The server maintains data integrity, addresses concurrency and serializability issues, and processes client queries. The client provides the user interface and is responsible for establishing a connection, possibly remote, to the server. Because the client/server protocol is platform independent connections may be created between heterogeneous machines. For example, a PC client running Windows NT may access a server on an HP workstation running UNIX. One Sybase server is capable of supporting several user databases. The *master* database is a special server database containing meta-information about other databases. This information includes the database name, database administrator, and user permissions for each database. Individual databases have a collection of system tables that hold meta-information about the other tables within a database (*sysobjects*), the columns in each table (*syscolumns*), and the keys for each table (*syskeys*).

The UCHGR system is based on the model described in Section 4.3, and is used to represent the local genetics database. It is implemented as a single Sybase database divided into a set of conceptual domains by naming conventions. In addition to a separate domain for each genetic view of the data, there is an administrative domain that contains meta-information about the constructs in the database. The *id* table is responsible, via stored procedures, for ensuring every construct in the database is allocated a globally unique identifier similar to an OID. The *type* table contains meta-information about all objects, relationships, and processes in the database. This includes the construct's name, type, and domain, as well as the Sybase table name where attributes of the construct are stored. Note that it does not identify the type of constructs connected by a relationship. The *dependents* table is used to associate complex attributes, such as sets, with a particular object. Meta-information about instances is stored in three tables within each domain. The *objects* table associates a type with each object id. Each instance of a relationship is associated with the appropriate relationship type by the *instances* table. The constructs connected by these instances are stored in the *roles* table in a highly normalized format. This representation allows the model to express complex relationships, such as ordered sets, in the same format as traditional relationships.

ASN.1 is a format interchange language defined in [121] [122]. This format was initially designed for data transfer, not data storage. However, it has become popular in the scientific community because of its ability to represent complex data structures. NCBI distributes its genetics database, Genbank, as a collection of several ASN.1 files. The definition file contains a collection of classes that define the structure of the data. This file is similar to a large C header file, and is the only source of meta-information available for the database. The complete class specifications for Genbank are provided in Appendix A, whereas Figure 7 presents three simple ASN.1 classes. The other files are data files consisting of a collection of instances from a known subset of the classes defined in the definition file. In Genbank, each data file consists of a single construct, usually of type cases Bioseq-set. Whereas the definition file is stored in ASCII text, the data files are represented in a binary format to conserve space.

Flat files are a useful representation since most databases systems can dump their data in table format. A flat file database is defined as a set of ASCII files, each representing a single concept. The data in the file are structured in a table format. Fields are delimited by a fixed character string, such as a comma, and tuples are delimited by

```
Cit-art ::= SEQUENCE {
    title Title OPTIONAL ,
    authors Auth-list OPTIONAL ,
    from CHOICE {
      journal Cit-jour ,
      Pub-set := SEQUENCE{
      book Cit-book ,
      name VisibleString,
      proc Cit-proc } }
    articles SET OF cit-art}
```

```
Cit-jour ::= SEQUENCE {
      title Title ,
      imp Imprint }
```

Figure 7 Example ASN.1 Classes

carriage returns. There is no meta-information associated with a flat file database.

5.3.2 Schema Transformation

Automatically presenting database schemata in the ER model can be divided into two logical steps. First, the meta-information must be read from the database, then it must be converted into an ER representation. In practice, these steps are intertwined by obtaining the meta-information for a single database construct and converting it into a corresponding ER construct before reading the next database construct. In order to facilitate code generation, each construct is associated with its database location. For example, an entity created from a Sybase database stores the Sybase table name used to generate it. In many cases, the required meta-information is not available; consequently, heuristics are used to generate a reasonable presentation of the schemata.

The UCHGR schemata are the easiest to translate into the ER model due to the detailed meta-information specifications. First, objects retrieved from the *type* table are mapped to strong entities. For each object, all its complex attributes are retrieved from the *dependents* table and translated to weak entities. The constructs connected by a relationship are not explicitly stored in the database, and must be retrieved by inspecting the instance data. For each relationship in the *type* table, the last instance of that relationship is retrieved from the *instance* table. The constructs connected by this instance are retrieved from the *roles* table. The types of these constructs correspond to the ER constructs connected by the relationship. Relationships connecting more than two constructs are considered to be, possibly ordered, sets. They are translated into a new entity corresponding to a set of constructs and a relationship between the set and the base entity. Relationships having no instances are ignored. The primary key for each entity and relationship is its unique identifier, foreign keys are not defined. Protocols, processes and environments are not of interest since they represent work flow concepts. If they were involved in a coercion they would map to entities.

The algorithm to map between the relational model and the ER model is well documented, and required only minor adaptations to work with the Sybase representation. User defined tables for the database can be identified from the *sysobjects* table. Columns associated with each table are retrieved, along with their location, name, type, length, and status from the *syscolumns* table. Primary and foreign keys are obtained from the *syskeys* table. The table names associated with foreign keys are also retrieved from the *syskeys* table. Once the information for a table has been read from the DBMS, it is easily mapped to a corresponding ER representation. If a table has no keys at all it maps directly to a strong entity. If it has a primary key that is independent of all foreign keys, all the nonforeign key attributes correspond to a strong entity. Weak entities are represented by tables having a single foreign key, and either no primary key or a primary key that includes the foreign key. If a foreign key exists in a table that has an independent primary key, it represents a relationship between the entity sharing the table and the entity corresponding to the table connected by the foreign key. Relationships are also formed by tables having two foreign keys and no independent primary key. In this case only, the relationship may have attributes beyond those required to link the related entities. If a table has more than two foreign keys all nonforeign key values represent a single entity, and the foreign keys represent a set of relationships associated with that entity.

The ASN.1 system was the most challenging to translate to the ER model due to the complexity of its attributes. Since translating from the ASN.1 model to the ER model was not the primary motivation of this work, a simple translation was chosen. A special parser was built to read the definition file and return the collection of classes it defines. The base classes for each database, and the associated data files, were explicitly identified. These classes are mapped to strong entities. Complex attributes of a class map to weak entities. Since ASN.1 classes can be recursively defined, a limit is placed on the number of attributes to expand. Due to the potential complexity of the resulting image, another limit controls the depth of weak entities displayed on the screen. These limits are required because the default matching algorithm will not expand attributes beyond the former limit in an attempt to find a match. In this simplistic translation of ASN.1 classes, relationships are not created. This mapping encouraged the creation of a simple interface to identify specific attributes, as described in the next subsection. An alternative mapping which generated a more descriptive ER representation was also considered. This mapping differentiated between optional, mandatory, and collection attributes by creating relationships with appropriate cardinalities: optional attributes are 0:1; mandatory attributes are 1:1; collection attributes are 1:n. Choice attributes would be represented using a 1:1 relationship between a generalizations of the choice attributes and the enclosing entity. This option was not pursued for two reasons. First, the additional complexity of the resulting ER representation is beyond the needs of this program. Second, the creation of an interface to access data based on the resulting ER representation is significantly more involved than for the simple mapping. However, this translation could be implemented in the future if needed.

Flat file databases do not provide meta-information so the schema cannot be converted into the ER model. However, to reduce the amount of work required to utilize flat files, two file formats for defining meta-information have been created. The first, and more general, is a flat file description (ffd) file that describes the layout of a database in a relational representation. The format for this type is:

delimiter
[*fileName*, *numCols*
[*columnName*, *alias*, *key*, *type*, *size*]^{numCols}]*

Where *fileName* is a character string representing the name of the table, and its location in the current directory. *columnName* is a character string used to identify the column, and can be overridden by the *alias* string. The *key* field may be **Y**, **F** or null representing a primary key, a foreign key or not a key value respectively. *type* is any of the standard primitive types, such as integer, string, float etc. *delimiter*s separate the fields in a tuple; carriage returns separate the tuples in a table. Null values are permitted for the alias and key fields. The *size* field may be omitted for *types*, such as integer, that have a default size. For example, a database containing a single table describing a person may appear as:

```
,
person, 3
name,,,string,50
ss#,id,Y,string,9
age,,,Integer,
sex,,,string,1
```

The second file format is used by GDB to distribute its genetics database. This format relies heavily on extensive type definitions and naming conventions. A meta-information file in this format consists of a collection of table definitions in the following format:

```
==========================================================
TABLE: tableName
Column Name                  Type                   Length
---------------------------  ---------------------- ----------
[ colName                    gdbType                length ]*
```

Key values are determined by *gdbType* which, in addition to the expected type information, also specifies whether the attribute represents a key or not. Due to a strict naming convention, foreign keys are distinguished from primary keys by a *columnName* value that does not start with a prefix of *tableName*. Foreign keys are associated with the table having that column as a primary key. In both the ffd and gdb formats, strong entities, weak entities, and relationships are created using the relational to ER mapping described by the Sybase translation. If the user does not want to create meta-information files for a flat file database, the graphical interface may be used to create the desired schema.

Adding a new database system to the collection of databases whose schemata can be automatically translated to the ER model requires creating a new subclass of the database class. The new class must define appropriate methods for retrieving meta-information from the database, recording the location of the data, and mapping constructs to the correct ER representation. However, by subclassing, appropriately significant functionality can be reused. For example, adding another relation database would only require modifying the meta-information retrieval; translating the data to an ER representation would be the same as for the Sybase RDBMS.

5.3.3 Data Manipulation

Generating a translation program requires the ability to read data from the source database, and insert data into the reference database. This requires an intimate knowledge of each database's data manipulation language (DML). This subsection describes the interface to each database system, and outlines the algorithms used for generating the appropriate DML code.

A comprehensive Sybase interface was provided as part of the Smalltalk development environment. An example of this interface is shown in Figure 8. The interface defines a database specification (step 1) and can establish a server connection (step 2). A session object (step 3) represents a sequence of database query operations. Arbitrary SQL commands are composed as strings, and passed to the session for parsing (step 4). Parsing the query is necessary because the interface allows variables to be embedded in the SQL command. These variables are identified by the parser, and are bound (step 5) prior to the query being executed (step 6). Late binding is extremely useful in several cases, for example when inserting several tuples into a single table. The results of the operation are obtained by repeatedly requesting an answer from the session (step 7). The value returned will either be #noMoreAnswers, or will be an answer stream

```
connect :=SybaseDetailedConnection new.              "Step 1"
connect environment:    'sprecher_db1';
        username:       'critchlo';
        password:       'foo'.
connect connect.                                     "Step 2"
sess := connect getSession.                          "Step 3"
sess prepare: 'SELECT  *  FROM gao_Gene where (gaoId > ?)'. "Step 4"
bind := OrderedCollection new.  bind add: 1802.
sess bindInput: bind.                                "Step 5"
sess execute.                                        "Step 6"
ans := sess answer.                                  "Step 7"
Stream endOfStreamSignal handle: [:sig| sig ] do:[
[true] whileTrue: [|temp|
      temp := ans next.                              "Step 8"
      Transcript show: temp  printString; cr ; cr. ].].
^nil
```

Figure 8 Sybase Interface Example

of unknown length. If a stream is returned, the individual tuples are retrieved by requesting the next value from the stream (step 8). Since the session is unable to determine when the end of the stream is reached, an endOfStreamSignal is sent when the data are exhausted. Values returned by Sybase are automatically converted into equivalent Smalltalk instances by the interface.

Manipulating data using the Sybase interface is trivial. Retrieving data just requires constructing the appropriate *select* statement. Since each entity and relationship contains its associated Sybase table name, and the attribute names reflect the column names, this is a simple string concatenation. Updating the database is only slightly more difficult. Entity instances are added to the database by creating an *insert* statement using the attribute positions to ensure correct ordering. If a relationship does not share its table with an entity, it is inserted in the same way. Otherwise, the appropriate tuple must be identified before the foreign key attribute is modified. This requires constructing an *update* statement with a where clause identifying the tuple to be updated.

The UCHGR interface is the same as the Sybase interface. However, additional queries are required to perform all data manipulations except retrieving entities. Retrieving a relationship is complicated by its unique storage format. First the type of the relationship is retrieved from the *type* table. Then, the instances of that type are retrieved from the *instances* table. Finally, the constructs participating in each instance are retrieved from the *roles* table. Inserting data is complicated because meta-information must also be inserted. Inserting an entity requires an additional insertion to update the *objects* table to reflect a new instance of the appropriate type. Several additional insertions are required to create a new instance of a relationship. First, the *instances* table must be updated to include the new instance. Then the instance is decomposed and entries are added to the roles table. Finally, additional attributes are inserted into the associated Sybase table, if appropriate.

A naive interface was developed to interact with flat file databases. An SQL DML was implemented because of the relational format of the data, and the desire to maintain similar user interfaces across databases. Currently, all data required to perform the query are read into memory before it is executed. This allows the VM system to perform memory management through paging, whereas still providing reasonable performance. This approach works because the individual files are small. If queries are to be performed on a collection of large files, a better approach would be to create and use an index. Determining which attributes to index could be achieved by an intelligent parsing of the query. Memory management techniques such as prefetching and caching could also be used. Query and join optimizations, which were not implemented, would dramatically improve performance. Because tables reside in memory for the duration of the connection, transaction semantics are not implemented. Instead, table modifications are cached until the session is closed, at which time the table is written to a temporary file. After the temporary file is successfully created, it is copied over the original. Because of the SQL interface, the code used to manipulate flat file databases and Sybase databases is identical.

An SQL interface was also created for manipulating ASN.1 databases. Due to the complexity of the binary ASN.1 format a parser was not built from scratch. Instead the ASN.1 data interaction is performed in two steps. First the data is converted into C structures by an existing program, then the desired attributes are imported into the

Smalltalk environment. The NCBI Toolkit [89] is used to convert between the ASN.1 data and C structures. The toolkit uses an explicit mapping between ASN.1 classes and a collection of loosely corresponding C structures. Unfortunately, this mapping is not consistent. For example, *choice* attributes in ASN.1 classes were mapped to six different representations in the corresponding C structures. This makes the second step, converting between the C structures and the corresponding ER representation, extremely difficult because the ER representation is based on the ASN.1 class definitions. Using the ASN.1 classes, and a collection of special rules, the C structures are traversed until the desired attribute is located. Then a mapping function is used to convert the C data into corresponding Smalltalk values. For primitive data types, this mapping is a predefined conversion function. Mappings for complex types such as sets were also defined. In order to present a homogeneous view of the data, all mappings return a set containing the attribute's values.

Once the ASN.1 data are mapped into the Smalltalk environment it can be manipulated by the interface. Unfortunately, due to the complexity of the ASN.1 classes, the interface must address two special considerations. First, a format representing the traversal of complex attributes is required to allow a specific attribute to be identified. The format used is similar to the dot notation used to reference C structures. For example, using the class definitions in Figure 7, the specification Cit-art.from.proc.title would reference the title of the proceedings in which an article appeared.

Second, returning the expected results of queries is complicated by complex attributes such as sets. For example, consider the query ‘select a.name, a.articles.title, a.article.from.proc.title from Pub-set a’, using the class definitions in Figure 7 and the data file in Figure 9 (a). The expected results of this query are shown in Figure 9 (b). To obtain these results, the query processor must understand that a set of articles needs to be mapped onto the same name. To ensure the correct mapping is performed, the concept of nesting levels was introduced. The nesting level of an attribute corresponds to how many attribute references are required to identify it. The query manager identifies the nesting level of all attributes involved in a query. For each level, the value sets associated with all attributes at that level are iterated over concurrently. Each iteration produces a single result set for that level. This result set is associated with each result set from the next deepest nesting. The collection of results for nesting level zero are returned to the user.

```
Database = {
   pub-set:{
      name:  OOPSLA
      articles: {{title: On Automatic Class Insertion with Overloading
                  authors: H.Dick, C. Dony, M. Huchard, T. Libourel
                   from:   proc :
                              {title:        OOPSLA 96
                              publisher:     ACM}
                  }
                 {title: A Metaobject Protocol for C++
                  authors: Shigeru Chiba
                  from:    proc:
                              {title:        OOPSLA 95
                               publisher:    ACM}  }}}}
                                  (a)

   {(OOPSLA, On Automatic Class Insertion with Overloading, OOPSLA 96)
     (OOPSLA, A Metaobject Protocol for C++, OOPSLA 95) }

                                  (b)
```

Figure 9 A Simple ASN.1 Database and Query

Since a.name is nested at level 1, and a.articles.title is at level 2, the same name is mapped to each of the titles. Because the value set for a.article.from.proc.title contains one element for each level 2 set, only one proceedings is associated with each article.

Currently the ASN.1 interface does not support modifications. This functionality has not been implemented for three reasons. First, UCHGR does not require this functionality. Significant work is involved in converting between the ER representation and the C structures used by the toolkit. Since the functionality is not expected to be used, this effort was not invested. Second, the semantics of modifying weak entities are not well defined. Due to the complexity of the data, and the lack of unique identifiers, correctly identifying the parent of a weak entity is complicated.   The obvious solution of only allowing modifications to strong entities would require development of a complex data specification format. Third, a modification to an existing data element would require the entire file to be rewritten in order to preserve the ordering of higher level constructs. This functionality can be added to the interface at a later time, if it is required.

Creating code to query an ASN.1 database using this interface is simple. Each strong identity corresponds to a data file. Weak entities are identified by the path linking them to their strong entity. This traversal is automatically created for entities of interest. Since complex attributes are not defined within the ER model, if a desired attribute is not a primitive type, it is converted to a representative value. This value is the representative value of the specified attribute's first attribute. In certain cases the representative value may be recursively defined, so an unspecified value is returned if a cycle is detected.

An appropriate interface must exist before code can be generated for a new database system. This interface defines the DML used, and directly manipulates the database. Once the DML interface is specified, methods to generate retrieval and insertion code from ER representations must be added to the translation generator. Depending on the database system's DML, some of the existing code may be reusable. For example, another relational database could use the existing Sybase code generation methods.

5.4 Correspondence Identification

Identifying corresponding constructs is the most important challenge that needs to be addressed when automating a schema coercion program. The more relevant correspondences automatically identified, the less user interaction required. However, if undesired correspondences are identified, more interaction may be required to delete them than to create the desired conversions manually.

SCoP compares each reference entity and relationship to each source entity and relationship to determine the relative confidence the two constructs represent the same concept. The matching heuristic attempts to identify two types of correspondences: basic and complex. Basic correspondences are identified using only information directly related to the constructs, such as their names and attributes, whereas complicated correspondences require external information, such as previously identified correspondences, to be identified. Since complex correspondences use existing conversions to identify new correspondences, it is possible the matching algorithm may require multiple iterations to finish identifying correspondences. However, since the confidence in a complex correspondence is always less than the confidence in the associated conversions, as shown in Section 5.4.2, eventually a fixed point in which no additional correspondences are identified will always be reached.

This section describes how conversions are automatically created. First, the basic matching algorithms are described. Then, the heuristics used to identify complicated correspondences and resolve complex structural conflicts are presented. The ease in which these conflicts are resolvable is an interesting, and surprising, feature of the matching algorithm.

5.4.1 Basic Correspondences

The base case for identifying all correspondences is determining the confidence that an attribute corresponds to another construct. This confidence is represented by one of three values: **goodMatch** 0.7; **fairMatch** 0.5; **poorMatch** 0.3. These values represent relative confidences in the various conditions, and were arbitrarily chosen. In addition, a fourth value, **nameWeight**, is specified by the user, and represents the weight to give to the construct names in the comparison. Typically, **nameWeight** varies from 0.4 to 0.8; if it is 1.0 the matching algorithm will emulate the name equivalence algorithm described in Chapter 2. The possibility that names may differ slightly is taken into account by using the Smalltalk string comparison method that returns an integer value between 0.00 and 1.00 representing the similarity between two strings. The correspondence confidence is multiplied by the similarity value to obtain a *similarity adjusted* confidence. While additional confidences may eventually be added to further refine the algorithm, the success of the heuristic in identifying correspondences with only three confidence values demonstrates the flexibility of this approach. The various options considered when comparing two attributes and their associated confidence are, in order from the most to least likely:

- The attributes have the same name, data type, data size, key value, minimum value and maximum value -- the confidence is **goodMatch**.
- The attribute definitions match on the name and key features -- the confidence is **fairMatch**.
- The attribute definitions match on all features except the name -- the confidence is **fairMatch**.
- The attributes are both keys, and do not necessarily share any other features -- the confidence is **poorMatch**.
- The attributes have the same name, without any other corresponding features -- the confidence is **nameWeight**.
- Otherwise the confidence is 0.0.

The confidence an attribute and an entity or relationship correspond to each other is extremely slim. If the names are similar, the confidence is **poorMatch** multiplied by the name similarity and the **nameWeight**. If the names are different, the confidence is 0.0.

Determining if entities and relationships are similar is a direct extension this algorithm. Figure 10 outlines the process used to determine the confidence two constructs represent corresponding concepts. First (1), each attribute from the reference construct is compared against all the attributes from the source construct to identify the

```
   matches: other
   | confidence nameSimilarity attributeConfidence |
   attributeConfidence := 0.
   attributes do: [:curAttr | |best|
(1)   self findBestMatchFor: curAttr in: other attributes.
(2)   attributeConfidence := attributeConfidence +
                     (best confidence / self numUsedColumns).].
   confidence := MatchValue new.
(3)nameSimilarity := self name spellAgainst: other name.
(4)(nameSimilarity > self minNameSimilarity)
(5)   ifTrue: [ (attributeConfidence >=  self minAttributeConfidence)
(6)            ifTrue: [confidence value: self; confidence:
                           (attributeConfidence + self nameWeight).]
(7)            ifFalse:[confidence value: self;
                           confidence:  self nameWeight].]
(8)   ifFalse:[confidence value: self;
                           confidence: attributeConfidence.].
   ^confidence
```

Figure 10 Determining Construct Correspondence Confidences

highest confidence correspondence, according to the heuristic described in the preceding paragraph. When a relationship is compared, the attributes in the connected entities corresponding to the relationship’s foreign key attributes are also compared since they may provide better matches than the foreign key attributes. These confidences are used to determine the average attribute correspondence confidence (2). Then, the similarity between the construct names is calculated (3). If this similarity is above a minimum threshold (4), the constructs names are considered to match. If the names do not match (8), the confidence the constructs correspond is just the **attributeConfidence**. Otherwise, the correspondence confidence at least **nameWeight** (7). If, in addition, **attributeConfidence** is above a threshold value (5), it is added to **nameWeight** (6), reflecting the additional confidence in the correspondence resulting from the attributes corresponding. Construct correspondence confidences above a threshold, **minConfidence**, are represented as conversions.

5.4.2 Complex Correspondences

Matching functions may be liberal in the identification of correspondences, because those with a low confidence are ignored. Therefore, the basic matching functions presented in the previous section have been enhanced with information about alternative construct representations. For example, consider Figure 11 where a relationship in the source database, *ordered*, corresponds to a relationship-entity-relationship structure, *orders-invoice-line*, in the reference database. This alternative representation is incorporated into the matching functions by adding the following information. One relationship corresponds to another if the basic matching function returns a correspondence, or the relationship's entities have correspondences and there is a path between the corresponding entities that includes the other relationship. The confidence in this correspondence is the average of the confidences of the entity mappings, adjusted by the length of the path. A relationship corresponds to an entity if either the basic matching function returns a correspondence, or if the relationship corresponds to two of the

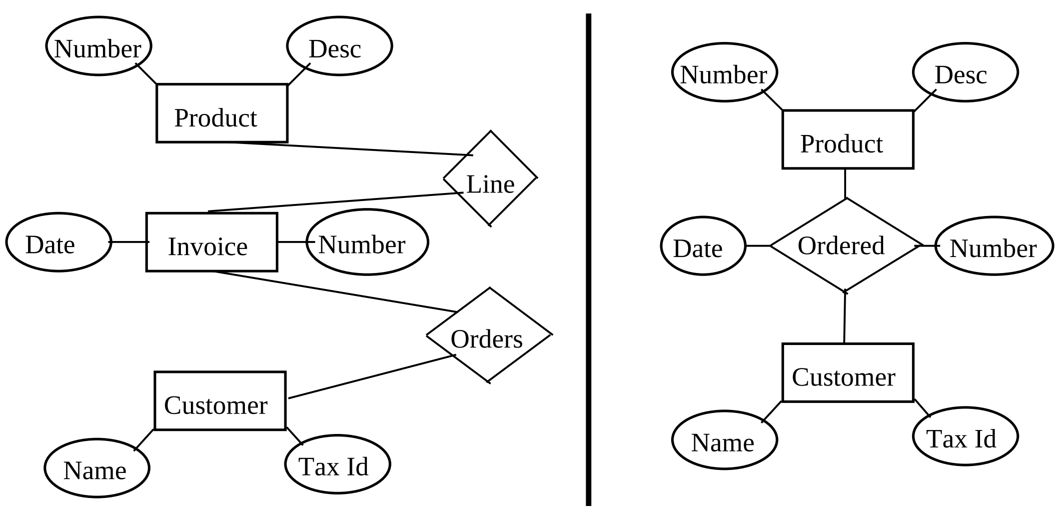

Figure 11 Complex Structural Conflict

relationships the entity participates in. The confidence of the resulting correspondence is the average of the confidences between the relationship correspondences, reduced by a value representing the complexity of the decomposition. If the value of **minConfidence** is high, some of these correspondences may not be identified. If the value is very low, correspondences will also be identified between relationships and relationship-entity-relationship-entity-relationship structures. However, setting the minimum confidence that low is not desirable because these expansions are rarely correct, and a large number of unwanted correspondences will be created.

Applying these enhanced matching functions to Figure 11 illustrates how structural conflicts are resolved. First, the reference entities are matched, and correspondences between the *product* and *customer* entities are identified by the basic functions. The correspondence between *invoice* and *ordered* is not created because the basic function does not recognize any similarity, and the relationship correspondences have not been identified. Then the reference relationships are matched. Since *orders* is on a path between the entities corresponding to the entities connected by *ordered*, *product* and *customers*, a correspondence is created. A similar correspondence is identified for *line*. The confidence in these correspondences is significantly lower than the average of the entity correspondences, due to the path length of three. These are the only correspondences identified the first time the matching algorithm is applied. However, on the next iteration, the correspondence between *invoice* and *ordered* is identified because the relationship correspondences have been defined. The confidence in this correspondence is lower than in the average of the relationship correspondences.

Whereas this matching algorithm performs well for most coercions, in some cases the heuristics are not appropriate. In these cases, alternative matching functions, such as pure name equivalence, may be defined to achieve the desired results. In the rare occasions when no matching algorithm will produce the desired results, the conversions must be manually created.

5.5 Transformations

Once a correspondence is identified, a conversion and its associated transformations are created to represent it. The basic attribute matching function described in the previous section is used to identify potential correspondences for each reference construct attribute. Each source attribute is mapped to at most one reference attribute. By associating a source attribute with only its best corresponding attribute, one attribute does not dominate the conversion. If a correspondence is identified, the attribute's value is used as the default transformation. If an appropriate correspondence cannot be found, a null transformation is used. If a key attribute is assigned a null transformation, a warning message is associated with the conversion.

The collection of attributes eligible for comparison with the reference attributes depends on the type of the source construct. If the source construct is an attribute, only it is eligible. If it is an entity, all its attributes are used. If the source construct is a relationship, however, all of its attributes and the attributes of the constructs it connects are available for comparison. Using this expanded attribute set greatly increases the likelihood of automatically recognizing the desired transformations.

Default transformations are mappings between corresponding attributes. If the attributes have the same data type, the transformation is just an assignment. However, if the data types are different, a type cast must be performed. Default transformations perform safe type casts whenever possible. For example, an integer will be cast into a floating point number. If the source type cannot be safely cast to the reference type, an assignment is used as the default transformation and a warning message identifying the unsafe cast is associated with the conversion.

Once the initial transformation is created, it is checked for two special conditions. First, default transformation that generate an **endOfStreamSignal** are identified. This signal is used by the translation methods to identify the end of data streams, so default transformations that raise this signal are enclosed in a handler. For example, since casting a *Timestamp* value to a *String* raises this signal, this cast is wrapped in a handler that catches the signal, and proceeds with the cast. Second, if the reference attribute is a key, and it corresponds to an attribute that is not a key, the default transformation is wrapped in code that prevents duplicates from being inserted. A warning is raised to ensure the transformation is verified before the translation program is generated.

Whereas the defaults are sufficient for many transformations, they may be manually overridden with arbitrary Smalltalk code. This allows the transformation to interact with both the source and reference databases, as well as perform complicated computations. Variables representing commonly used values are provided to facilitate the creation of complex transformations. **sourceDB** and **referenceDB** represent the appropriate databases, and connections to these databases are obtained through the #**getConnection** method. **reference** is the connection to the reference database used by the conversion. If the reference database is a UCHGR database, a unique identifier can be generated by applying the #**nextId** method[2] to it. The **userDefined** variable is assigned a user specified value at the start of the conversion. This variable is used to store data between transformations. The **library** variable is assigned an instance of the *ConversionLibrary* class. This class defines the **typeConversions**[3] methods. Finally, the **extraReference** variable contains conversion specific information used by the translation. For example, when converting to an UCHGR entity, it contains the id of the entity's type. This variable should be referenced only by experienced users. Complex transformations are not checked for type safety because they are assumed to be error free.

5.6 Logs

To reduce the interaction required to create several similar coercions, a primitive logging facility has been implemented. Since logging requires additional overhead, this facility is normally deactivated. However, when active, all conversion and transformation manipulations are recorded in a temporary log, that can be explicitly saved to a file. Replaying this file later, possibly in a different coercion, recreates the conversions represented by the log. In this way, similar conversions may be created in different coercions with minimal additional interaction.

---

[2] Currently, his method does not query the database to obtain the correct identifier. Instead, it increments a local counter. This is useful when working with smaller instances of the database. However, if the production database is to be manipulated, the method needs to be reimplemented to access the correct Sybase stored procedure instead.

[3] See Section 5.7 Annotating.

The logging facility provides the ability to undo conversions, in last created, first removed order. At any point, new conversions may be created and are inserted into the log's current position. Inserting new conversions does not affect other conversions in the log. Undone conversions are recreated by rolling the log forward. Removing a conversion from the log requires undo the conversion, then explicitly deleting.

There are two strategies that govern what happens when the log attempts to recreate a conversion. In either case, if corresponding constructs cannot be located in the current coercion, the conversion will not be created. This will normally occur only when the log is replayed in a different coercion. Assuming corresponding constructs can be identified, the first strategy will create the exact same coercion, including the confidence and all transformations. The second strategy evaluates the confidence of the conversion with respect to the current coercion, and determines whether it would be created or not. If the new confidence is lower than the current value of **minConfidence**, the user is prompted to confirm the conversion's creation. Whether the conversion is created or not, it remains in the log and may be recreated later. Consider the correspondence between *invoice* and *ordered* identified in Section 5.4.2 . Since this correspondence is dependent on the existence of other correspondences, if they are not present, it is unclear whether the conversion should be created. Because this algorithm is based on the dependencies between conversions, and interacts with the current coercion settings, the results of replaying a log using this strategy are unpredictable.

5.7 Annotations

Annotating the participating databases is the most successful approach to reducing the interaction required to create the desired coercion. Annotation files allow significant additional meta-information to be associated with the constructs represented in a database. This information is used to better identify correspondences and create transformations. If the annotations are sufficiently expressive, creating the desired coercion between two annotated databases does not require any interaction.

Figure 12 defines the annotation file format. It is essentially a sequence of *names*, that map to constructs, and their associated *annotation features*. Most features are applicable to either attributes, or entities and relationships. Only the **alias** feature is defined for all constructs. In addition, the following features are only used if the annotated database is the reference database: **defaultConversion**, **mandatoryConversion**, **defaultCoercions**, **typeConversions**, and **userDefined**. The features associated with attributes are described next, followed by the features associated with entities and relationships.

```
AnnotFile            =    Entry

Entry                =    (name (annotation features) * )

Annotation features  =    key boolean                       |
                          key (attr*)                       |
                          alias string                      |
                          typeInformation (string*)         |
                          defaultConversion (exp)           |
                          manditoryConversion (exp)         |
                          typeConversion (type cast)*       |
                          defaultCoercions (source conf)*   |
                          userDefined (exp)
```

Figure 12 Annotation File Format

Consider the attribute annotation defined in Figure 13. The **alias** annotation provides an alternative name that will be used by the matching function. The **key** value of *false* specifies the attribute is not to be regarded as a key, even if the database representation implies otherwise. The **typeInformation** provides a detailed type specification for the attribute, that is also used during matching. The additional types are presented in a successively more restrictive order. The **typeConversions** are used in conjunction with the **typeInformation** when a transformation is created. If an attribute of the specified type is mapped to the annotated attribute, the associated casting method is used by the default transformation. If an attribute of an unspecified type is mapped to this attribute, the arbitrary Smalltalk code specified by the **defaultConversion** is used. The value of *SRC* is used to represent the associated attribute. The **mandatoryConversion** feature is identical to the **defaultConversion**, except that the specification is always used as the default transformation, and the *SRC* representation is not available.

Now, consider the entity annotation defined in Figure 14. The **alias** feature defines an alternative name used during matching. The strings associated with the **key** feature represent the attributes that are to be considered keys of this entity, even if the database representation contains conflicting information. **defaultCoercions** outline conversions that should be created if the named construct exists in the source database. It

```
(cash
      (key false)
      (typeInformation (currency US-dollars))
      (alias money)
      (typeConversions (Canadian-dollars CanToUs) (Yen YenToUS))
      (defaultConversion ((($SRC$ * 100) truncate) / 100) ) )
```

Figure 13 Attribute Annotation

```
(bank (alias ‘Large Conglomerates’)
      (key (‘Bank name’ ‘Branch location’ ))
      (defaultCoercions ((‘Credit Union’ 0.75)))
      (userDefined (foreignExchange getJapaneseRate)))
```

Figure 14 Entity Annotation

also provides the confidences associated with these conversions. Finally, the **userDefined** feature can be assigned arbitrary Smalltalk code. By default, this code is assigned to the userDefined variable for every conversion involving this entity.

To provide generality in associating ER constructs with annotations, three naming conventions are defined. These conventions represent a naming hierarchy and are presented in order from the most to least specific. First, if the *name* is a dot notation specified attribute name, the annotations are only applied to that particular attribute. Second, if the *name* is a generic string the annotations are propagated to all constructs of that name, regardless of type. This may result in several attributes, entities, and relationships being assigned similar annotations. Finally, there are two special names, **defaultEntity** and **defaultRelationship**, used to define **defaultCoercion** or **userDefined** features for every entity or relationship respectively. The features associated with a particular construct are the most specific defined in its naming hierarchy.

5.8 Translation Generation

The final step in schema coercion is the generation of a translation program to transfer data from the source database to the reference database. However, before a translation program can be created, the coercion must be completely specified. This requires all conversions and their associated transformations to be appropriately defined, and all errors and warnings to be corrected. In SCoP, the translation program is represented as a single Smalltalk class, *Translator*. Generating the translation code is done in two steps. First, the initialization routine is created, then each conversion defines two methods to perform the data transfer specified by its transformations..

The initialization routine defines the instance variables and calls the conversion methods. Generating code to assign the variables is simple. The **library** variable is always assigned an instance of the *ConversionLibrary*. The source and reference databases, and their connections, are defined using information from the coercion. Before the calls to the conversion methods are specified, the conversions are sorted. The sort orders the conversions based on data dependencies. This ensures that the data required by a conversion are written before the conversion is called. For example, before an instance of a relationship can be inserted into a database, the entities it connects must already be there. Roughly, the sort ordering corresponds to all strong entity conversions, then weak entity conversions, and finally relationship conversions. Once the conversions are sorted, the code calling the appropriate methods is added. The name of the method associated with a conversion is a combination of the conversion's source and reference construct names. Since only one conversion can exist between two constructs, it is extremely unlikely two conversions will have the same method name.

After the initialization method is defined, each conversion generates two methods that perform its transformations. The first method constructs the queries required to interact with the databases. Two queries are required for each conversion: the first selects the desired data from the source database, the second inserts the transformed data into the reference database. The DML code for these queries is generated by the algorithms presented in Section 5.3.2. Once these queries have been generated, creating the method is simple. First, the **userDefined** variable is assigned the appropriate value. Then the queries are prepared, and the selection query is executed. A loop is created to repeatedly call the second method, passing it the selection answer stream. Finally, the database connections are closed.

The second method performs the transformations on the data and inserts it into the reference database. The next tuple is read from the answer stream, and is mapped to a set of local variables. The transformation for each reference attribute is assigned to a local variable representing the attribute. Since the transformations are represented as Smalltalk code, this assignment is straightforward. Finally, the reference variables are bound to the insert operation, the operation is executed, and the result returned.

The translation program is written to a file that redefines the *Translator* class. However, before it can be imported into the Smalltalk environment, the existing class definition must be deleted. To facilitate the data transfer process, every time a translation program is generated it is used to redefine the *Translator* class. Once the program is imported, it may be inspected and modified as normal Smalltalk code. The program is executed by creating an instance of the class, that automatically invokes the initialization method.

CHAPTER 6

IMPLEMENTATION GUIDE

This chapter describes the SCoP user interface in detail. Section 6.1 outlines how to create an instance of a coercion and associate the desired databases. Section 6.2 discusses manipulating conversions and transformations. Section 6.3 describes the other capabilities SCoP provides, such as logging. Finally, Section 6.4 presents the translation facility and demonstrates data transfer. To facilitate this discussion, a single example will be used throughout the chapter. This example demonstrates the creation of a coercion between two databases containing customer and invoice data.

6.1 Getting Started

SCoP is implemented in the VisualWorks Smalltalk environment[93] [94] with the *CoercionCreation* class as its primary interface. This class maintains and displays all database and conversion information. Every coercion is represented as an instance of this class. Prior to an ASN.1 database being loaded, the *ComplexType* **maxLevel**, **maxRecursiveLevel** and **displayLevel** variables should be set. These variables define the maximum depth to which complex structures are expanded and displayed. Associating the source and reference schemata with a coercion is accomplished using one of three approaches.

The first approach utilizes a simple graphical interface. An instance of the *CoercionCreation* class is opened and the **create** option is selected from either the **sourceDB** or **referenceDB** menus, whichever is appropriate. Information about the database is requested using the display shown in Figure 15. The *Database Name* field is used to identify the correct database. In the Sybase and UCHGR databases, this corresponds to the Sybase database name. For ASN.1 databases, valid names correspond to the set of databases for which the base constructs and data files have been identified. Currently, three of these databases, *NCBI*, *Hinf*, and *Mjan*, are defined but others may be added as required. The *Database Manager* options, *Flat*, *Sybase*, *Utah_GC*, and *ASN.1*, identify the four known database systems. The *Location* field identifies the physical location of the database. For Sybase and UCHGR databases, this is the appropriate Sybase server name. For ASN.1 and flat file databases, it is the directory and name of the meta-information file. For flat file databases, the file name also encodes the database format. If the file name ends with a **.gdb** suffix, it is represented in GDB format. Otherwise, it is stored in ffd format. The *Annotations* field is optional, and identifies the

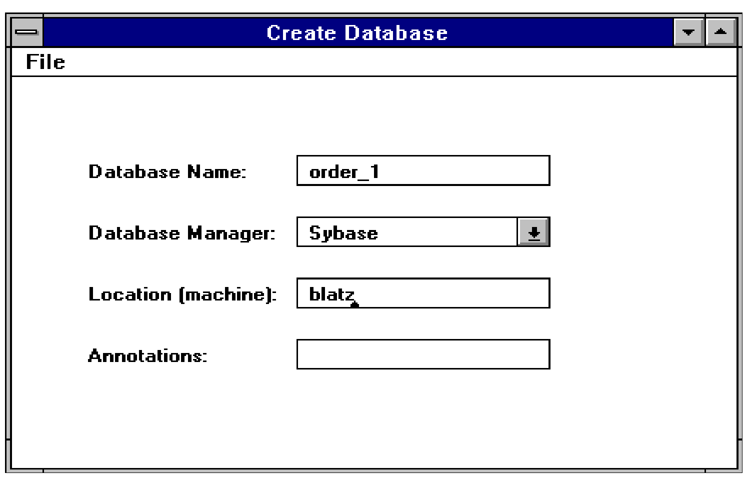

Figure 15 GUI Database Specification

annotation file associated with the database. Once the specification is complete, the schema can be read directly from the database by selecting the **read** option.

The second approach also uses the database specification interface described in the previous paragraph. However, the schema is explicitly defined using the **create** option, instead of automatically read from the database. Manually defining a schema requires every feature of each construct to be specified. Because of the time consuming and error prone nature of this task, this option is only available for database systems that may not provide the required meta-information, i.e., flat file and ASN.1 databases. However, even for these systems the recommended approach is to create a meta-information file and use it to generate the schema

The third approach creates source and reference databases using Smalltalk code, then associates them with an instance of *CoercionCreation*. This approach requires understanding how database specifications create their corresponding schema. It is preferable to the first approach because it is faster and easier to modify. Creating the invoice coercion and associated Sybase databases using Smalltalk code is shown in Figure 16. The creation of a *Database* instance requires the database's identifier to be passed to the #**new:** method. After each instance is created, the appropriate name and location are specified. An annotation file may also be defined using #**annotationFile:**,

```
|coercion src ref|
coercion := CoercionCreation new.
src := Database new:'Sybase'.  ref := Database new: 'Sybase'.
src location:'blatz';name: 'order_2'; readDB.
ref location: 'blatz';  name: 'order_1';  readDB.
coercion reference: ref; source: src; open.
```

Figure 16 Alternative Specification

passing a string representing the file's name and location. After the appropriate information is provided the schemata are read. Finally, the databases are associated with the coercion as either the source or reference, and the display is opened.

Some databases require additional information before a connection can be established. For example, Sybase and UCHGR databases require a user id and password. When a schema is read, this additional information is requested. Both the Sybase and UCHGR databases associate this information with the database instance so additional requests are not necessary. Once the ER representation of a schema is created, it may be saved to a file and later read back using the appropriate options. Since the schema is usually automatically generated, these operations are infrequently used.

Figure 17 shows the invoice coercion and its associated schemata. The source schema is on the right, and the reference schema is on the left. The database names appear above their respective schemata. The standard ER graphical representation is used with two minor modifications. First thick lines are used instead of arrow heads to represent cardinality. For example, the *placedby* relationship in the *order_1* schema is a many-to-one relationship, as shown by the thick line connecting it to customer. Second, weak entities are represented in a box with a thick border, instead of a double box. All strong entities are drawn in a single column on the left. Weak entities appear on the row following their dominant entity, but not necessarily in the same column. Relationships are located to the right of the entities, halfway between the entities being connected. Whether attributes are displayed or not is controlled by the database's **Display attributes** option. Colors distinguish different types of lines from each other. Black lines connect attributes to their entity. Purple lines associate relationships with their constructs.

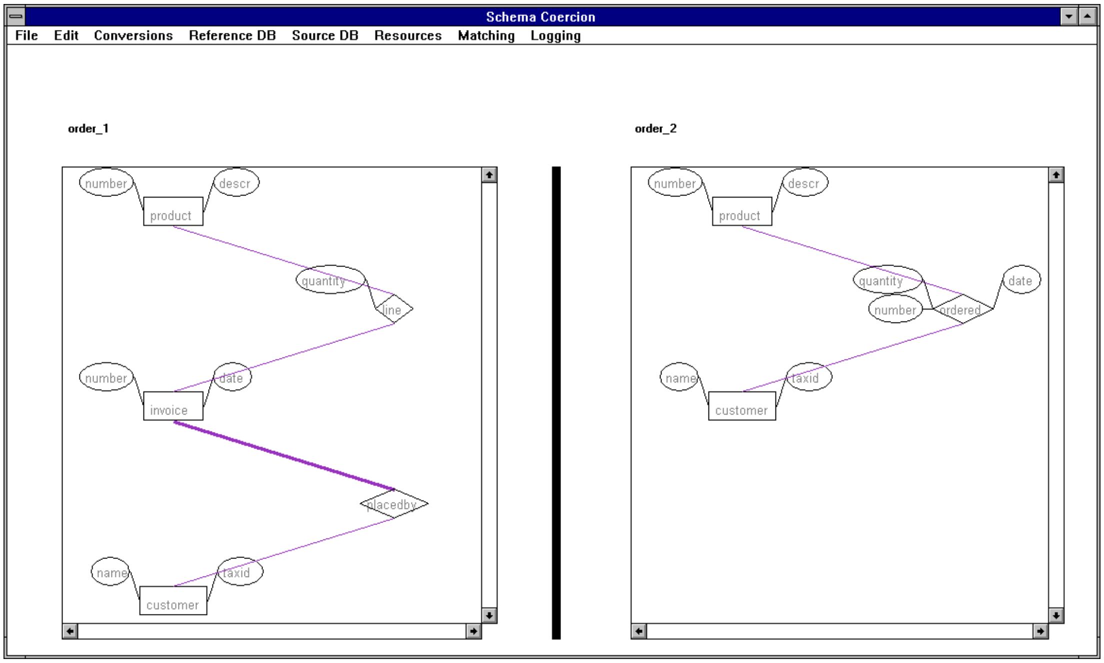

Figure 17 Coercion with Databases

6.2 Conversions and Transformations

Once a coercion and its associated schemata are displayed, conversions may be manually specified, or automatically generated. To create a conversion between two constructs, simply select them. Manually specified conversions are assumed correct, and assigned a confidence of 1.0. Before generating default conversions, the variables used by the matching functions may be adjusted using the **Matching** menu. In addition to the **minConfidence** and **nameWeight** variables described in the previous chapter, the number of iterations for each match request may be set. Setting this value using the **Fixed Point** option will result in the matching algorithm being repeatedly applied until no new conversions are created. This iteration will eventually terminate because complex conversions have a lower confidence than their supporting conversions. Therefore, eventually all unidentified correspondences will have confidences less than the **minConfidence** threshold; at this point, no new conversions are created. If alternative matching functions are defined, the desired one may also be selected from this menu. Once the variables are set, default conversions are created using the **Conversions** menu. If an undesired conversion is created, it is deleted by selecting it and choosing the **Delete selection** option from the **Edit** menu.

Conversions are represented by colored lines between their source and reference constructs. If the line is red, an error or warning was generated when the transformations were created. Otherwise, the line is a shade of blue. Very confident conversions, such as a user defined conversion, appear pure blue, whereas low confidence conversions are shaded yellow. For the invoice coercion, a **nameWeight** of 0.3, and a **minConfidence** of 0.5 were specified. The default conversions generated by the matching algorithm are shown in Figure 18. The *orderedToinvoice* conversion is generated on the algorithm's second iteration.

When a conversion is created, default transformations are automatically generated. To view the information associated with a conversion, double-click on the appropriate line. Figure 19and Figure 20 show the default transformations for the *orderedToinvoice* conversion. The conversion display is split into three areas. The first presents the information required to generate the desired selection and insertion queries. The second shows the current value of the **userDefined** variable, and allows modifications. The third defines the transformations associated with the reference construct's attributes.

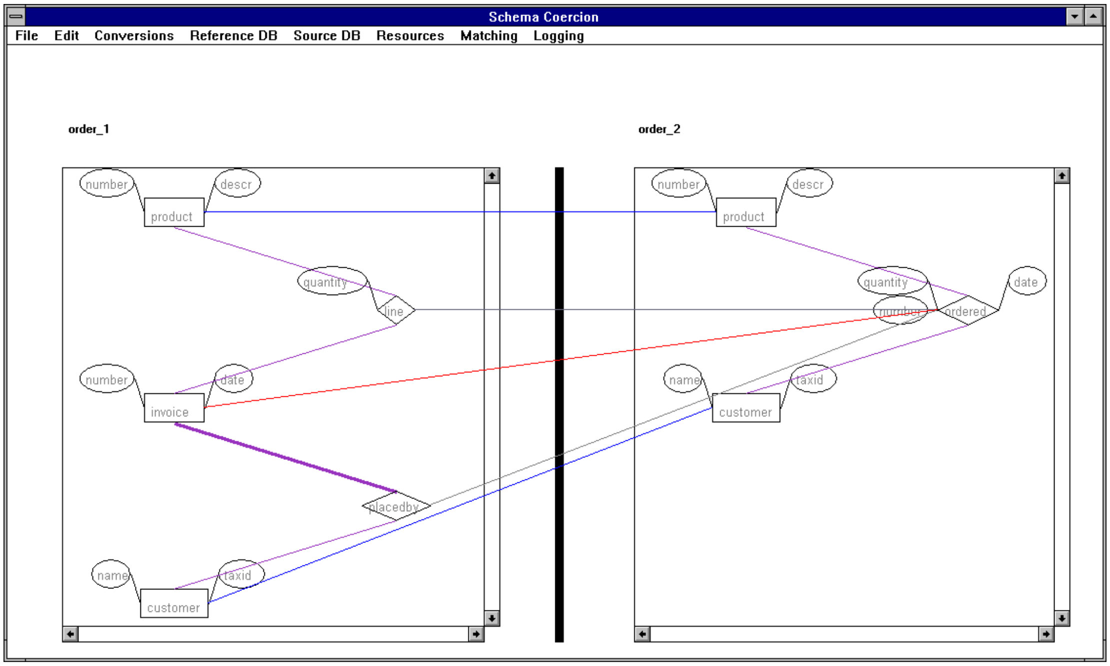

Figure 18 Default Conversions

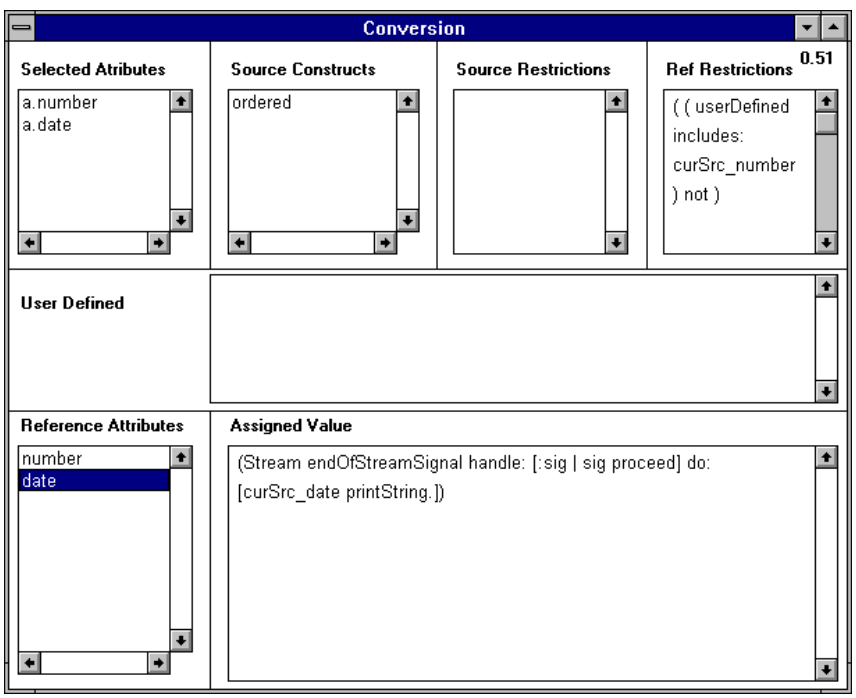

Figure 19 *orderedToinvoice* **date** Transformations

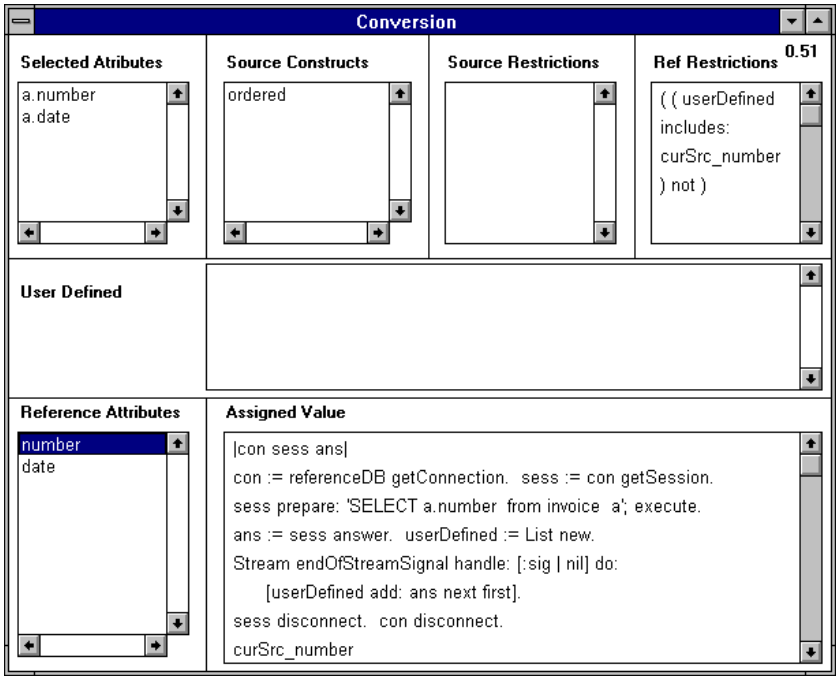

Figure 20 *orderedToinvoice* **number** Transformation

To construct the queries, four pieces of information are required: the attributes retrieved by the selection, the constructs involved in the selection, and restrictions imposed on the source and reference data. The *Source Constructs* list displays the constructs involved in the selection query. Additional constructs are associated with the conversion using the **insert** option from the **Operate** menu. A single character *alias* is assigned to each source construct in the list. The aliases are allocated sequentially for each construct in the conversion, starting with **a**. The *Selected Attributes* list identifies the source attributes involved in the conversion. Additional attributes are retrieved by either directly inserting them, or selecting them using the **browse** option. Browsing is especially useful when attempting to add either an attribute with a long name or a nested ASN.1 attribute. The names of all *Selected Attributes* must be preceded by their construct's alias, even if there is only one construct.

A local variable is associated with the current value of each of the *Selected Attributes*. The default variable name is obtained by affixing a *curSrc_* prefix to the attribute's name. Preventing duplicate variables from being defined when several attributes have the same name requires adding suffixes to their associated variables. The suffix consists of an underscore and a number representing the location of the attribute relative to the other selected attributes with the same name. For example, if the **product**'s **number** attribute was also selected in the *orderedToinvoice* conversion, its value would be associated with **curSrc_number_2**, whereas the initial *number* attribute is always associated with **curSrc_number**.

The *Source Restriction* prevents data from participating in the conversion. In the invoice coercion, this may be used to restrict which customers, products or orders are transferred. The *Reference Restriction* prevents undesired insertions from occurring, and ensures updates occur correctly by identifying the appropriate tuple to modify. In the *order_1* database, the *placedby* relationship is represented as an attribute in the *invoice* table. Updating an instance of this relationship requires correctly identifying the tuple representing the associated *invoice*. This is accomplished using the *Reference Restriction*.

Restrictions return a boolean value using a syntax similar to the standard *SQL where clause* with two minor enhancements. First, the **userDefined** and local variables may be accessed. They are automatically recognized and replaced with their current values. Second, the **#include:** method is defined. These enhancements allow the restrictions to be specified in a straightforward way. For example, the default reference restriction in Figure 19 references the current value of the *number* attribute. In addition, this syntax allows restrictions to be appended to the appropriate query, and executed by the database system.

The third section defines a transformation for each reference attribute. The attributes are presented on the left, and the transformation associated with the selected attribute is displayed on the right. The transformations are defined using arbitrary Smalltalk code that may access the variables described in Section 5.5 as well as the local variables. The value assigned to the reference attribute is the value of its transformation. Because the assignments are not performed within individual methods, this value should not be returned using ^, since that will result in the method prematurely exiting.

The end of the source data stream is identified by an **endOfStreamSignal** raised by the database interface when the next tuple is requested. Therefore, any other operation that may raise this signal should be wrapped in a handler to prevent the conversion from prematurely terminating. For example, since converting a *Timestamp* value to a *String* will generate this signal, the default transformation shown in Figure 19 is wrapped in a handler that proceeds when the signal is raised. This catches the signal before it reaches the external handler, and continues evaluating the expression.

Consider the default transformation shown in Figure 20. A warning is associated with the *orderedToinvoice* conversion because **invoice**'s key, **number**, is associated with a nonkey source attribute, **number**. The warning ensures the default transformation is verified before the translation program is generated. If the default transformation was a simple assignment, duplicate values could be inserted into the reference database, thereby violating the integrity of the database and possibly generating a run time error. Instead, the default transformation defines code that imports the defined key values, and modifies the reference restriction to ensure duplicates are not inserted. This additional code does not change the value of the transformation that is, as expected, the value of **curSrc_number**.

Whereas there are alternatives to the default transformation, the obvious approach of initializing **userDefined** to an empty list, and using the **number** transformation to add the current **number** to the list is incorrect. This is because the **userDefined** variable will be updated by the transformation before the restriction is checked. Therefore, no new **invoice** will pass the test, and nothing will be inserted. If **curSrc_number** was inserted into the list after the restriction was checked, this approach would work.

Unfortunately, two minor modifications are required before the desired translation program can be generated. First, the *orderedToline* conversion required the **ordered number** attribute to be selected and associated with the **line**'s **invoice** attribute. Second, the *orderedToinvoice* conversion required the warning be removed from the conversion after verifying the default transformations are correct. All errors and warnings associated with a conversion are displayed in a separate screen, shown in Figure 21, when the conversion is displayed. These errors are explicitly removed using the **Operate** menu.

6.3 Other Features

The base functionality required to define coercions and create translation programs is not sufficient to make a useful tool. To encourage productive interaction and development, several support features are implemented in SCoP. To allow coercions to be incrementally defined, the ability to save and restore them is provided by the **File** menu. Identifying all existing conversions is a critical step in the coercion process.

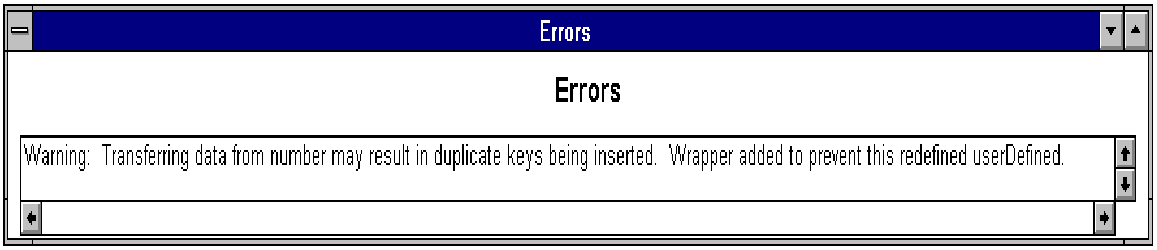

Figure 21 Error Messages

Unfortunately, enumerating this set by displaying individual conversions requires a significant effort because a conversion is only drawn when both its constructs are visible. To alleviate this problem, a list of all existing conversions is available from the **Conversion** menu.

Because UCHGR submits information to community databases, as well as imports information from them, coercions that are the logical inverse of other coercions are required. To aid in the creation of these coercions, a function that creates a primitive inverse coercion based on an original is provided. This function reverses the source and reference databases, and creates conversions between constructs related in the original coercion. Because transformations are not invertible, instead of using information from the original conversion to generate the new transformations, they are generated normally. This function is available from the **Resources** menu.

The logging functions described in Section 5.6 are available from the **Logging** menu. The **Begin Logging** option activates the log and begins recording all conversion and transformation manipulations. The **End Logging** option deactivates logging. The **Replay options** specify the strategy used to recreate conversions. The **Undo** and **Redo** options allow traversal of the log file by deleting and recreating conversions. The **Delete**

next step option removes the last undone conversion from the log. The **Save log file** writes the entire log, including undone conversions, to disk. Saved logs may be loaded and replayed using the **Replay** option.

6.4 Generating the Translation

Once the desired conversions have been created and all errors and warning are corrected, the translation program can be generated. This is accomplished by selecting the **Translation Generation** option from the **Resources** menu. The file associated with the program is prompted for, and the new *Translator* definition is written to it. The existing *Translator* class is deleted, if it exists, and the new definition is imported. The class is successfully redefined if the transformations are syntactically correct. Otherwise, the offending code is displayed in a separated window, and the class definition is left incomplete. After the transformation is corrected, the translation program may be regenerated. Once the *Translator* is defined, it may be examined and modified like any other Smalltalk class.

The invoice translation driver method, *initialize*, is shown in Figure 22. Located in the *initialize-release* category, this method is responsible for initializing the database connections, and calling all of the conversion methods. First, the global variables are initialized to known values. Next, the source and reference databases are created using information from the coercion, and their associated connections are established. Then the methods associated with each conversion are called in order. Finally, the connections and global variables are released.

```
initialize

userDefined := nil.
library := ConversionLibrary new.
sourceDB:= Database new: ’Sybase’.
referenceDB := Database new: ’Sybase’ .

sourceDB machine: ’blatz’; name: ’order_2’;annotationFile: nil.
source := sourceDB getConnection.
referenceDB machine: ’blatz’;name: ’order_1’;annotationFile: nil.
reference := referenceDB getConnection.

self fromcustomerTocustomer. self fromproductToproduct.
self fromorderedToinvoice. self fromorderedToline.
self fromorderedToplacedby.

reference disconnect. source disconnect. library release.
```

Figure 22 Initialize Method

The *conversion* category contains all the conversion methods, including the two methods associated with the *orderedToinvoice* conversion that are shown in Figure 23. The first method initializes the **userDefined** variable, and obtains the database sessions. The selection query is prepared and executed. The answer is retrieved, and checked to ensure at least one value is returned. If it is, the insertion query is prepared and the second method is called until the answer stream is exhausted, as identified by an **endOfStreamSignal**.

The second method reads the next tuple from the answer stream, and assigns the source values to the appropriate local variables. Then, the transformations are performed in order, with their values assigned to a set of reference variables. Since this query performs an **insert**, not an **update**, the reference restriction guards the execution of the

```
fromorderedToinvoice
|sourceSession referenceSession sourceResults|
userDefined := [nil]value.sourceSession := source getSession.
sourceSession userDefined: userDefined.
referenceSession := reference getSession.
referenceSession userDefined: userDefined.
sourceSession prepare:'SELECT a.number,a.date FROM ordered a '.
sourceSession execute.
sourceResults := sourceSession answer.
sourceResults isString
      ifFalse:[referenceSession prepare:
                  'INSERT INTO invoice VALUES ( ?, ?, ?)'.
               Stream endOfStreamSignal handle: [:sig| nil] do:
                   [[true] whileTrue: [
                        self fromorderedToinvoice: sourceResults
                            reference: referenceSession. ]. ].].
sourceSession disconnect. referenceSession disconnect. ^self
```

(a)

```
fromorderedToinvoice: sourceResults reference: referenceSession
|curSourceRow curSrc_number curSrc_date  curRef_insertAttributes
curRef_number curRef_date|
curSourceRow := sourceResults next.
curSrc_number := (curSourceRow at: 1).
curSrc_date := (curSourceRow at: 2).
curRef_number := (true ifTrue: [[|con sess ans|
            con := referenceDB getConnection.
            sess := con getSession.
            sess prepare:'SELECT a.number from invoice a';execute.
            ans := sess answer.  userDefined := List new.
            Stream endOfStreamSignal handle: [:sig | nil] do:
                   [userDefined add: ans next first].
            sess disconnect.  con disconnect.
            curSrc_number] value]).
curRef_date := (true ifTrue:
                  [[(Stream endOfStreamSignal handle:
                    [:sig | sig proceed] do:
                        [curSrc_date printString.])] value]).
curRef_insertAttributes := OrderedCollection new.
curRef_insertAttributes  add: curRef_number;
            add: curRef_date; add: nil.
((( ( userDefined includes: curSrc_number ) not )) value)
      ifTrue: [referenceSession bindInput:
                  curRef_insertAttributes;execute; answer.
            ]. ^true
```

(b)

Figure 23 Conversion Methods

query to ensure inappropriate tuples are not inserted. If the restriction clause evaluates to true, the reference variables are bound to the query, and it is executed. The answer returned by the insertion query is ignored, but must be requested to ensure the next insertion executes properly. If the restriction evaluates to false, the conversion proceeds with the next source tuple. For updates, the restriction is bound to the query's where clause to ensure only the correct tuples are updated.

If the *Translator* code is modified, these modifications are not reflected in the associated definition file. However, the class may be explicitly **filed out** if the changes are to be saved. The data transfer is initiated by executing the command: **Translator new**. After the database specific information is obtained, the data transfer will proceed without user interaction until either an error occurs or the transfer completes. Three common errors account for the vast majority of exceptions generated by translation programs. The first is failure to connect to one of the databases. This will occur when the network connection is down, or if the username or password is incorrect. The second is attempting to violate one of the reference database's integrity constraints. This is usually the result of an incorrect transformation, or failure to correctly specify the reference restriction. The third is a Smalltalk runtime exception raised by an incorrect transformation. For example, attempting to execute a method on an uninitialized variable.

6.5 Functional Enhancements

Because of the vast number of database systems and matching algorithms available, defining interfaces for all of them is impossible. Instead, SCoP is designed to allow incremental functional enhancements. In particular, augmentations are expected in three areas: the known database systems, the matching functions used to identify correspondences, and the annotation specifications.

Even though additional database systems are anticipated, defining a new system requires significant coding. The database identifier must be added to *Database*'s **AllowedDatabase** class variable, and the **new:** method must be modified to recognize this identifier and return the appropriate subclass. New database systems should be represented by a direct subclass of either *Database*, or an abstract subclass of *Database*. Subclasses provide code reuse between similar database systems. For example, if the Oracle database system was added, an abstract *RelationalDatabase* class would be created. This class would encapsulate generalized concepts found in all relational database systems, such as mapping from tables to corresponding ER constructs. The subclasses of this class, *OracleDatabase* and *SybaseDatabase*, reflect the representational differences between the database implementations.

Database subclasses must define at least three methods: **dbms**, **getConnection**, and **readDB**. The **dbms** method is a trivial function that returns the database's identifier. The **readDB** method is responsible for obtaining meta-information about a database and transforming it to an appropriate ER representation. The **getConnection** method establishes a connection to a specific database, usually through an external database interface. The interface is a collection of classes that provides consistent access to external database systems by defining generic methods that query the database and retrieve the corresponding results. Interfaces for commercial database systems, such as Oracle and Sybase, may be purchased directly from Smalltalk vendors. Others, such as

ASN.1 interfaces, must be created as required. The database interface, and associated DML, are used by the *Conversion* class to generate appropriate selection and insertion queries.

Each ASN.1 database requires explicit identification of its data files, and their associated base structures. Therefore, accessing a new ASN.1 database requires creating a new interface. In addition to defining subclasses with the appropriate information, the *ASN1Schema* and *ASN1Connection* classes must be modified to return the appropriate class based on the database name. Fortunately, subclasses need to override only four methods, including the class initialization method that defines the base structures and associated data files.

Whereas the default matching algorithm is extremely versatile, other matching algorithms may be desired. New matching methods should be defined on *Database*, and are selected by setting the coercion's **matchingFunction** through the *CoercionCreation* **fileMenu**. When matching is requested, the reference database's **createDefaultConversionsFrom:** method invokes the specified method, passing the source database as the sole argument. Depending on the algorithm, additional methods may be defined on the *Entity*, *Relationship* and *Attribute* classes.

The annotation file format presented in Section 5.7 provides meta-information required by the current matching functions. If the existing algorithm is enhanced, or new algorithms are added, additional meta-information may be required; examples of the information that may be desired are suggested in [110]. To aid in this enhancement, most interaction between the annotations and their related constructs is enclosed in *Annotation* and *AnnotationFile*. Defining a new annotation requires creating a parser to accept the new information and modifying the **convertFrom:to:using:** and **annotate:** methods appropriately.

CHAPTER 7

VALIDATION

Two phases of testing have been conducted to validate the concepts presented in Chapters 5 and 6. First, several small test sets demonstrated SCoP's ability to resolve conflicts and automatically generate correct transformations. Section 7.1 presents some of these tests and their results ordered from least to most complicated. After these tests were satisfactorily completed, the principal challenge problem was addressed. This problem was chosen based on UCHGR's need to import data from the Genbank community database. Section 7.2 describes this coercion in detail, and discusses the reusability of the translation program with respect to transferring a second, similar data set. Section 7.3 discusses the expected scalability of this approach, and Section 7.4 outlines how this approach handles the evolution of either the source or reference schemata.

7.1 Basic Tests

The first test set corresponds to mapping a database onto itself. The ability to correctly identify this coercion forms the basis for recognizing other, more complicated, coercions. The databases used in this test, shown in Figure 24, are identically defined except for the social security number (**ss**) attribute, that is represented as a character array in one database (**book_1**), and an integer in the other (**book_2**). This test established the

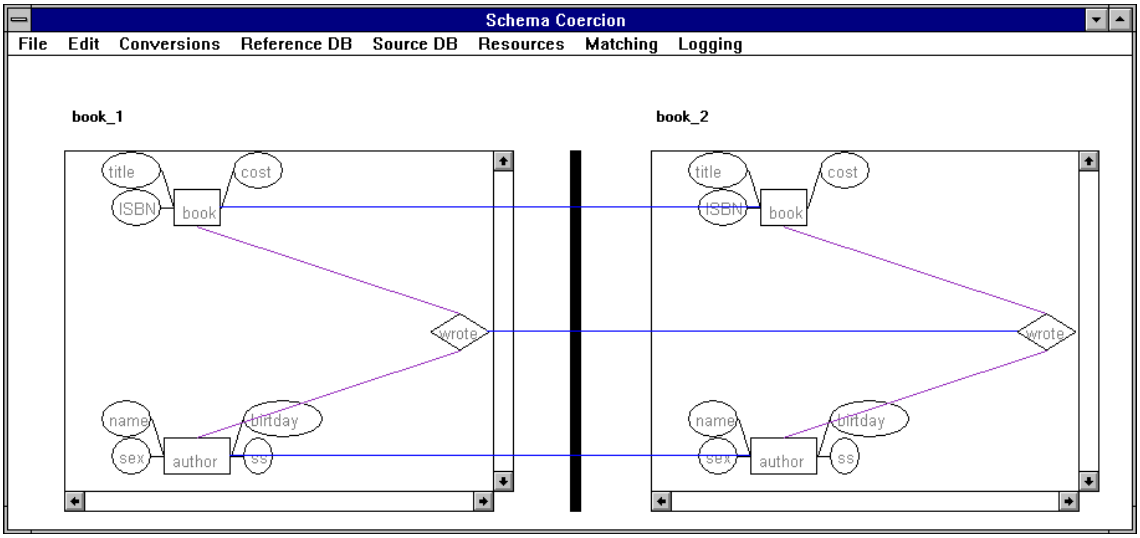

Figure 24 Mapping A Database Onto Itself

capability of the algorithm to identify correspondences between identical constructs and create basic transformations. Using different data types for the **ss** attributes demonstrated the ability to perform type casts, and associate casting errors with the conversion. The annotation **typeInformation** feature was tested by refining the **book_1 ss** attribute with information inexpressible by the database system:

(author.ss (typeInformation (string integer)))

that specifies that the attribute, although represented as a string, is implicitly restricted to contain only integer values. This information is used to reconsider the safety of the type cast.

The second test set modifies one of the databases, as shown in Figure 25, to introduce a simple structural conflict. The same information is represented in both databases, however the **wrote** relationship is implicit in the source database's dominant / subordinate relationship between the **book** and **author** entity sets. The correspondence between the subordinate entity set, **author,** and the **wrote** relationship is correctly

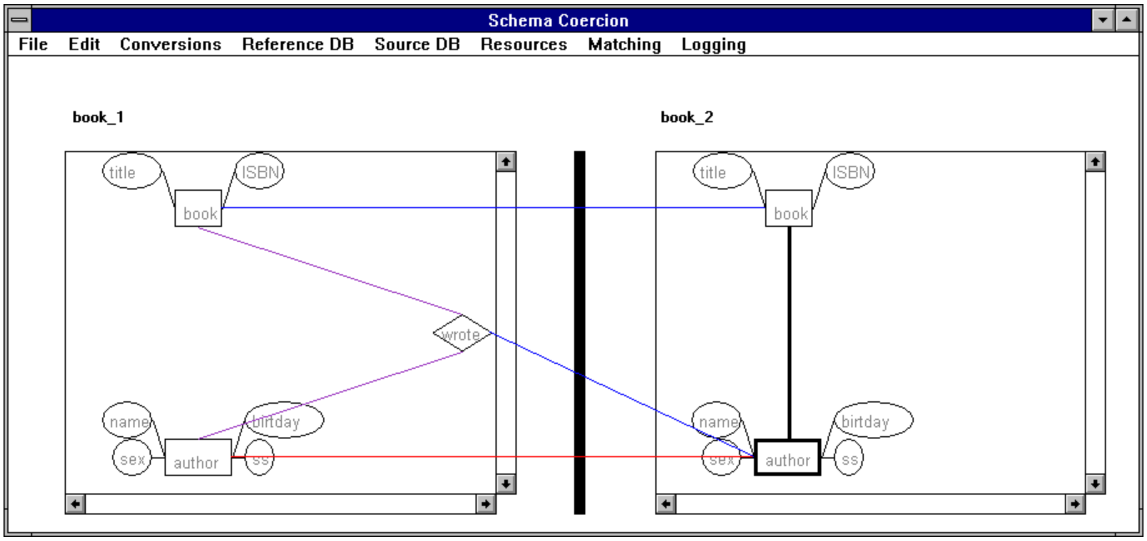

Figure 25 First Structural Conflict

identified, as is the correspondence between the **book** entity sets. Appropriate transformations are created for the associated conversions. However, because the **book_2 author**'s primary key is different from the **book_1 author**'s, a warning is associated with the **authorToauthor** conversion, and the default **ss** transformation attempts to prevent duplicate key insertion. Choosing the **author** entity set as dominant, and the **book** entity set as subordinate, changes the default conversions predictably: the **wrote** relationship and the duplicate key warning are still associated with the subordinate entity, in this case **book** instead of **author**.

The third test set consist of a collection of similar coercions. Three representations of marriage, presented in Chapter 2, are coerced into a fourth representation. Figure 26 shows the default conversions for the first coercion. The **person** entity sets correspond to each other, and the **marriage** relationship corresponds to both the **married** relationship and the **marriage** entity set. Two modifications to the

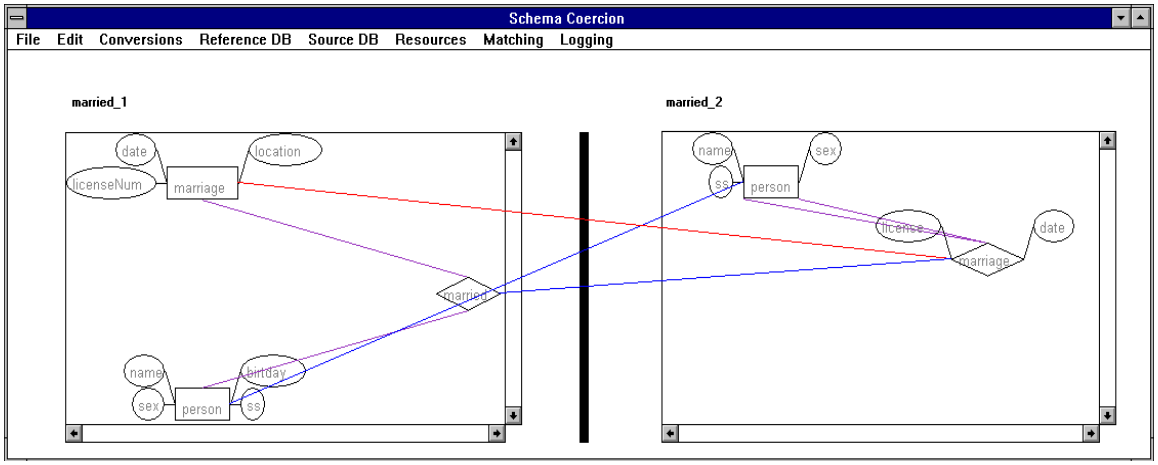

Figure 26 Marriage Between Two Persons

default conversions are required. First, the duplicate key warning must be removed from the **marriageTomarriage** conversion. Second, the **marriageTomarried** conversion must be modified to create two instances of **married** for each instance of **marriage**. This is required because the **marriage** relationship implicitly associates two **persons** with the concept of *marriage*, whereas the **married** relationship explicitly associates one. Because all source schemata in this test set relate *marriage* to two people, this problem must be addressed by every coercion. There are several ways to define an appropriate transformation, the simplest of which is shown in Figure 27. Both the **husband** and **wife** attributes are selected, and the **person** transformation returns a list containing these values. The Sybase interface associates nonlist values with each list element before executing the query. Therefore, the **licenseNum** will be associated with both **persons**, and desired insertions will be performed.

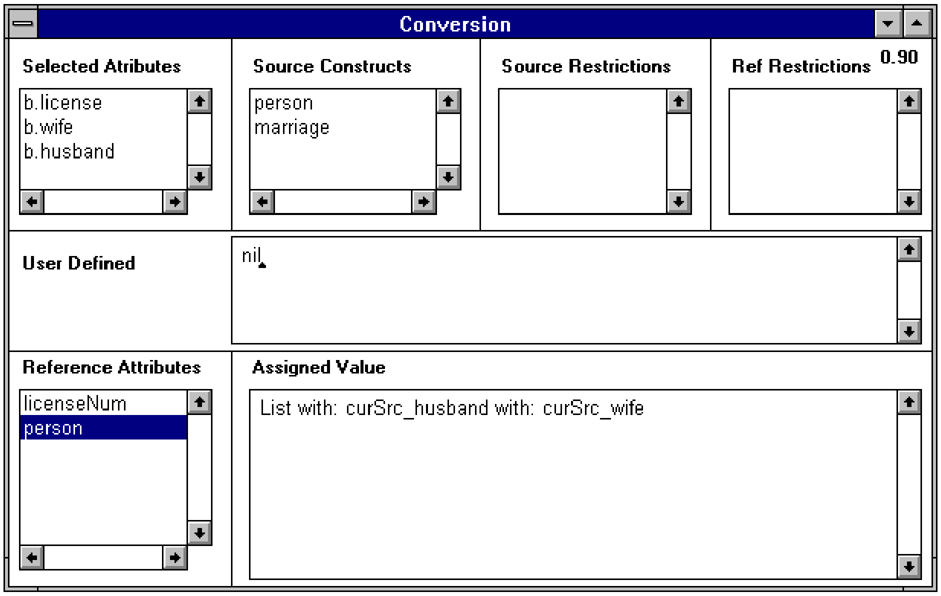

Figure 27 marriageTomarried Conversion

The second coercion, shown in Figure 28, requires creating a conversion between the **husband** and **wife** attributes and the **person** entity set, in addition to inserting two instances of **married** for each instance of **marriage**. User defined transformations must be assigned for these conversions because attribute correspondences cannot be identified. Whereas the transformations for **name** and **birthday** may return default or null values,

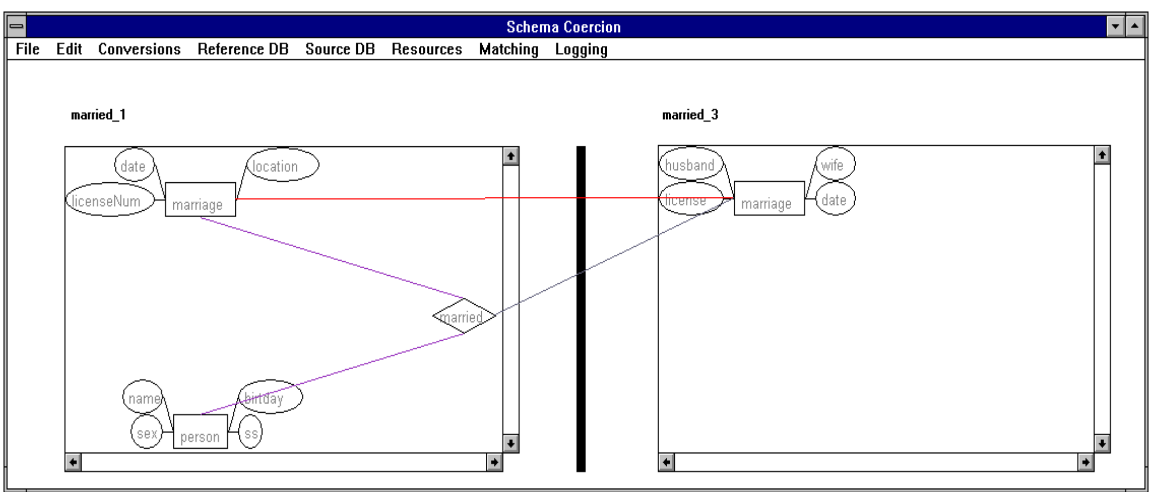

Figure 28 Marriage as an Entity

the correct transformations for **ss** and **sex** can be defined: the **ss** transformation returns the source attribute, and the **sex** transformation returns **M** for the **husbandToPerson** conversion, and **F** for the **wifeToperson** conversion.

Figure 29 presents the final coercion in this test set. In addition to mapping each instance of **marriage** into two instances of **married,** two minor modifications are required. First, the duplicate key warning must be removed from the **marriageTomarriage** conversion. Second, the **sex** transformations in the **womenToperson** and **menToperson** conversions must be modified to return the appropriate values.

The final test set consists of the complex structural conflict example presented in the preceding chapter. Because of the thorough evaluation presented there, additional discussion is not necessary. Whereas most coercions require some modification to achieve the desired transformations, these tests demonstrate the validity of this approach in two ways. First, the majority of the desired correspondences and the associated

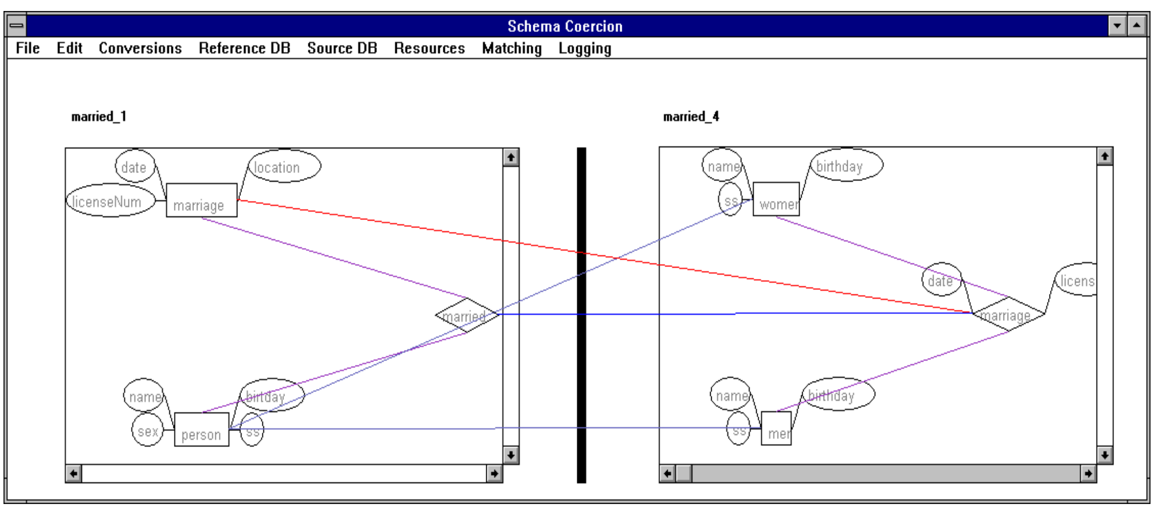

Figure 29 Marriage between Men and Women

transformations are identified. This demonstrates the ability to recognize similar concepts despite radically different representations. The incorrect transformations result from either complex structural mappings or insufficient source information. Correctly identifying these mappings requires a thorough understanding of the concepts being coerced, so it is unlikely even a specialized coercion program could automatically recognize them. Second, the complicated transformations involved in these coercions, such as restrictions preventing duplicate insertions, could be specified. Unfortunately, most other programs restrict the transformations that may be defined. As a result, some complex coercions are inexpressible. The ability of this program to recognize complex correspondences and easily express complicated transformations is a significant advance.

7.2 Challenge Problem

The challenge problem consists of transferring information about the Haemophilus Influenzae bacteria from Genbank to a local UCHGR database; a representative coercion from the genetics domain. Since UCHGR requires a local representation of this data, in addition to demonstrating SCoP’s ability to perform complex coercions between substantial databases, this problem addresses an issue of practical importance. The Haemophilus genome consist of a single chromosome approximately 1.8 Mbp in length, containing over 7,000 genes. The chromosome is decomposed into 163 overlapping sequences of varying lengths by Genbank.

The generic Genbank ASN.1 distribution format is described in Section 5.3. The Haemophilus data are contained in two data files. The first file contains a single entry that identifies Haemophilus’ chromosome and associated sequences within its **Seq-**

entry.seq.inst.ext.seg attribute. This attribute specifies the sequence's Genbank identifier, start location, and end location within the chromosome. The second file contains a single entry representing all Genbank bacterial sequence and gene information within its **Bioseq-set.seq-set** attribute. Individual sequence data, including the sequence's Genbank identifier, name, and length, is represented by the **set.seq-set.seq** attribute. Genes are associated with their enclosing sequence through the sequence's **inst.seq-data** and **annot.data.ftable** attributes.

For the purpose of this coercion, the local database may be viewed as four entity sets and three relationships: **organism**, **chromosome**, **sequence**, **gene**, **orgXchr**, **chrXseq**, **seqXgene**. Because the local database and the Genbank databases have extremely different representations, the matching algorithm is unable to identify any correspondences, even basic attribute correspondences. The desired coercion consists of 7 conversions and 39 transformations, that were manually defined. The transformation complexity varied from reasonably complicated to trivial: 15 transformations require Smalltalk programming, 12 directly correspond to source constructs, 9 evaluate to null, and 3 to constant, non-null values. The large number of complex transformations is typical for coercions to UCHGR databases because a utah_id must be generated for each new entity and relationship, and relationships must query the reference database to retrieve the utah_id of the constructs they connect.

Approximately one day was required to correctly identify and specify the conversions and transformations involved in this coercion. The translation program generated from this coercion successfully imported the Haemophilus data into a local database. Unfortunately, the program requires approximately 60 hours, on a Pentium 120 running NT with 8 Mb of main memory, to complete the transfer. Four factors contribute to this significant execution time. First the source database consists of over 225 Mb of binary ASN.1 data in an unindexed format. Simply reading the data, converting it to C structures, and immediately releasing these structures requires over an hour. Second, insufficient main memory was provided by the test machine. The minimal memory and the large data set results in continuous paging; at least 32 Mb of main memory is required, and 64Mb is preferable, for acceptable performance when manipulating this amount of data. Third, UCHGR database is located on a remote machine used by other individuals. Ideally, this overhead could be reduced by performing the coercion on a dedicated machine that also contained the Sybase server. Fourth, the use of Smalltalk instead of a compiled language such as C++ introduced additional run-time overhead. It is difficult to estimate the effect of this overhead; however it is not considered to be significant when compared to the other factors. A similar, but slightly smaller, coercion executed on a Pentium 120 with 32Mb main memory, running NT and the Sybase server completed in approximately 20 hours.

Because of the significant representational differences between the databases, SCoP's inability to identify correspondences was expected, if mildly disappointing. However, the ability to express this coercion, and the complex transformations associated with it, is viewed as an alternative validation of this approach. In addition, it is estimated that approximately one week would be required for a programmer familiar with both databases to create a translation program similar to the one generated in one day using SCoP. This substantiates the belief that SCoP's intuitive interface masks significant computational complexity from view, allowing concentration on the conversions.

Unfortunately, significant interaction is required to manually specify this coercion. However, the existence of several similar transformations implies annotating the reference database may significantly reduce the interaction required, and further demonstrate the feasibility of annotating databases. Whereas it is possible to specify a coercion by explicitly annotating every transformation, this does not reduce the overall interaction required. Therefore, these annotations should be used sparingly. The composition of the annotation file created for this coercion, presented in Appendix B, is summarized in Figure 30. Of the 119 lines in the file, only 78 are annotation definitions; the remaining 41 are comments.

These limited annotations significantly improve correspondence identification. Because the reference constructs are renamed appropriately, identifying the corresponding source constructs is trivial. However, a small amount of interaction is still required to correctly define the associated conversions. Two conversions require an additional source construct to be selected, and all source restrictions require modification to ensure that only the Haemophilus data is imported into the local database.

```
22 alias annotations
      7   prevent any matches
      15 match appropriate constructs
7 defaultCoercion annotations
      2 for default constructs
      2 prevent any conversions from being identified
      3 provide better mappings to source attributes
5 manditoryConversions were created
      3 provide mappings to a constant value
      2 provide a common mapping
1 typeInformation
3 userDefined annotations
      2 for default constructs
      1 over-rode the default
```

Figure 30 Annotation File Composition

In addition, 24 of the 39 transformations (62%) are correctly defined. Four factors contribute to the generation of incorrect transformations. First, six transformations require querying the reference database to obtain the utah_id associated with an object. Second, five transformations return a simple assignment involving the correct source construct, instead of the desired computation involving the construct. Third, two transformations identify corresponding attributes from the wrong source construct. Fourth, two transformations are associated with constructs when they should be assigned constant values.

The ability to correctly specify 62% of transformations with only 38 annotations, when there is no other evidence of correspondence, demonstrates the expressive power of these annotations. It is unlikely that additional annotations would reduce the overall interaction required to obtain the desired coercion, since it is as much effort to explicitly define a transformation in the annotation file as through the SCoP interface. However, completely specifying the current coercion using only annotations allows these annotations to be used as a starting point for defining similar coercions in the future. These results also suggest the set of expressible meta-information is sufficient for most coercions.

After the Haemophilus data were transferred, another coercion involving data for the Methanococcus bacteria, also represented in Genbank, was requested by UCHGR researchers. Because of the similarity between the databases, the existing coercion could expedite the new coercion definition in three ways. First, the existing annotation file could recreate the conversions and transformations. The interaction required to generate the new coercion would be the same as for the initial coercion: the more detailed the annotations, the less interaction required. Second, the translation program code could be directly modified to retrieve the Methanococcus data instead of the Haemophilus data. This would require minimal additional interaction, and could easily be performed outside the Smalltalk environment. Third, if the initial coercion had been logged, replaying the log using the Methanococcus database would create the correct conversions without any additional interaction.

The second approach was used in this case because the annotation file for Haemophilus had not yet been created, and the Haemophilus coercion definition was not logged. It took approximately two hours to modify the original translation program to perform the second coercion, despite several representational differences between Haemophilus and Methanococcus in the Genbank database. This reduction in the required user interaction is significant, since several additional transfers from Genbank to the local database are anticipated.

7.3 Scalability

If an approach successfully addresses large scale problems, it becomes usable to a larger segment of the population, and therefore increases its overall value. Within the schema coercion problem there are at least three different variables whose scale may affect performance: the number of entities and relationships represented by a database, the amount of data stored in a database, and the number of coercions defined. Increasing the number of entities and relationships complicates the graphical representation of the schema, and increases the number of comparisons required to identify correspondences with other databases. The amount of data stored in the source database affects the data transfer time and network traffic. Defining multiple coercions will increase the overall interaction required, although the incremental cost of each coercion may be reduced. The remainder of this section outlines the effect of each of these variables on the approach described in the previous chapters.

Increasing the number of entities and relationships will have a significant impact on the usability of this approach for two reasons. First, the simple schema display will not create a comprehensible diagram with a large number of constructs: the entities are randomly displayed, and the lines connecting relationships cover other graphic objects and may blur together. To display a large schema in an understandable way, the display algorithm must be rewritten to minimize a wide variety of constraints including: the distance between entities connected by a relationship, and the number of graphic objects crossed by lines. This solution has not been pursued because it would require significant programming effort and some commercial applications already demonstrate similar capabilities; however, it could be added if required. Second, the matching algorithm uses an exhaustive search to identify potential correspondences between the databases. As the number of constructs represented by the participating databases increases, the time required to identify correspondences will increase appropriately. Therefore, a schema with a sufficiently large number of constructs will substantially reduce the effectiveness of the matching algorithm. Unfortunately, since there is no way to determine a *priori* the likelihood two constructs correspond, neither an order nor an index can be imposed on the constructs. Therefore, exhaustive search is the only appropriate algorithm for this domain. However, the matching algorithm could be modified to recognize early in the comparison when an acceptable result will not be generated, and could reduce the complexity of the comparison appropriately.

The size of the source database affects the time required to perform the data transfer, but does not affect the user's interaction with the interface. The effect of increasing the size of the source database varies depending on the organization and DBMS of the database. For example, if a table uses indices appropriately, increasing the size of the table will not dramatically affect the time required to retrieve data from the source database. However, if the query involves only unindexed attributes or the DBMS does not support indices, for example flat file or ASN.1 databases, increasing the size of the table may have a dramatic affect. The size of the reference database will have a minimal (O(log n)) impact on the time required to perform a data transfer, although transformations that query the reference database will be affected by size changes in the same way.

Obviously, as the number of defined coercions increases, the overall user interaction required will also increase. However, if the coercions are from source databases with similar schemata onto a single target database, this interaction can be significantly reduced by using either the logging or annotation features defined by this work. This scenario is common, for example in the genetics problem described in the previous chapter.

The approach outlined in this work is not scaleable to databases with large numbers of entities and relationships. Whereas these schemata are often viewed as the result of bad design, they are common in domains such as genetics where there are large numbers of similar, but distinct, concepts which are closely related in a variety of ways.

This failing is a result of the attempt to identify all correspondences between databases, is common to all approaches and cannot be easily overcome. Fortunately this filing does not overly restrict the applicability of this approach because it is able to reduce the interaction required to define coercions between similar databases, and to handle large amounts of data. These are the more common problems in the domains of interest. For example, there were approximately 50 entities and relationships represented by the Utah and Genbank databases used in the previous section, however the Genbank bacteria database contains over 225 Mb of binary data. In addition, the expectation of defining additional coercions between the Genbank and the local database highlights the importance of reducing the incremental cost of defining additional, similar coercions.

7.4 Schema Evolution

If a single data transfer from the source database to the reference database retrieves all desired data, future evolution of the source or reference schemata will not require the coercion to be updated. However, if multiple data transfers are required, modifications to the source and reference schemata must be reflected by the coercion. Multiple data transfers are used when the source database is frequently updated with new information. In this case, a series of incremental data transfers, that incorporate the data entered since the last transfer into the reference database, are required to ensure the reference database remains current.

This work provides three methods to reduce the interaction required to incorporate schema modifications onto an existing coercion. First, if the schema modifications are minor, the corresponding ER representation may be directly modified to reflect them. For example, if a column is added to a table in a relational database, the corresponding attribute can be directly added to the appropriate entity or relationship. The participating conversions can then be explicitly updated. If the schema modifications are complicated, manually modifying the ER representation is a time consuming and error prone task, and another method should be used. The second method, defining a new coercion based on the original coercion's log, is capable of handling more complex modifications. Replaying the log identifies appropriate conversions between corresponding constructs. The default transformations associated with a conversion reflect the new schemata, whereas transformations that had been explicitly modified in the original coercion will be appropriately redefined in the new coercion. This method is appropriate if the matching algorithm is able to recreate the majority of the original conversions. However, if the schema modifications are extensive, this may not be the case and the third method should be used. This method uses the existing annotation file as a basis for defining a new annotation file that reflects the new schemata. The new file is then used to automatically generate a new coercion. Redefining the coercion in this way may require significant interaction, but is still the best option when extensive modifications prevent the desired correspondences from being identified. Methods two and three may be combined to further reduce the required interaction.

The ability to handle schema modifications is critical to any approach's long term usefulness. The logging and annotation features defined by this work also allow existing coercions to reflect schema modifications with little effort.

CHAPTER 8

FUTURE WORK AND CONCLUSIONS

8.1 Future Work

This project's development is expected to continue in three major areas: database functionality, matching capability, and annotation specification. Expanding database functionality involves not only improving existing interfaces, but also creating new ones. Three enhancements should be made to the current ASN.1 interface. First, intelligent parsing and dynamic index creation could significantly improve query performance. Second, the ability to insert new objects into an existing database should be provided. Third, a more complex transformation from ASN.1 classes to corresponding ER constructs should be defined: for example, relationships could be created for optional attributes. In addition to these enhancements, as the need arises to interact with additional database management systems, such as ObjectStore, Oracle and GemStone, new interfaces will be defined and incorporated into the existing framework. By expanding the variety of databases that may participate in a coercion, the usefulness of the tool is increased.

Enhancing the matching capability involves a combination of refining the existing algorithm and defining new ones. The existing algorithm could be refined to resolve additional structural conflicts and complex naming conflicts. Using a thesaurus in conjunction with the string similarity function would provide better correspondence identification than currently possible, because homonym conflicts could be recognized in addition to the synonym conflicts already detected. Using multiple thesauruses simultaneously would further improve correspondence identification, since specialized thesauruses provide better matches for terminology rich domains, such as genetics, whereas general thesauruses perform well for generic domains. Alternative algorithms ranging from those easily defined, such as name equivalence, to those requiring substantial additional research, such as an expert system for a specific domain, could also be implemented. Because of the difficulty defining a single matching algorithm suitable for all coercions, significant research remains in this area.

The set of expressible annotations is expected to evolve for two reasons. First, some conversion information, such as source and reference restrictions, cannot be specified by the existing annotations. Defining annotations for this information would permit complete specification of coercions using just the annotation file. Second, as the matching capability is enhanced, additional meta-information will be required. For example, new annotations identifying the application domain of a database, or specifying which thesaurus to use when attempting to identify correspondences involving a particular construct, may be desired.

8.2 Conclusions

Whereas domains such as genetics are desperate for a solution to the schema coercion problem, traditional research has ignored this problem in favor of schema integration and schema evolution. Traditional tools semiautomatically generate an integrated schema based on an initial collection of manually specified schemata. Source constructs and their corresponding integrated representation are related by a limited set of transformations. Transferring data between the initial schemata and the resulting schema is usually not addressed; instead, this translation must be implemented without assistance.

This work differs from others in that it provides a comprehensive tool directly addressing the schema coercion problem. This tool is capable of automatically transforming schemata from several data models into corresponding ER representations. It semiautomatically identifies correspondences between these schemata, and allows complex transformations between corresponding constructs to be defined. Based on these transformations, a translation program to automatically transfer data may be generated.

The flexibility and usefulness of this integrated approach is enhanced by the logging and annotation features that have been incorporated. These features reduce the interaction required to define similar coercions, and the interaction required to incorporate source and reference database schema modifications after an initial coercion has been defined. This approach been validated by defining multiple coercions and transferring data between two diverse genetics databases.

In addition to creating a useful tool, this work addresses the significant theoretical problems associated with schema and data conflicts in two ways. First, complex structural conflicts are resolved by liberally identifying potential correspondences, and using low confidence values to eliminate the unlikely candidates. Second, annotations allow meta-information to be associated with a database, including information that may not be expressible within the database’s management system. This information significantly improves correspondence identification and transformation creation, reducing the interaction required to define the desired coercion.

APPENDIX A

GENBANK ASN.1 CLASS DEFINITIONS

```
Link-set ::= SEQUENCE {
   num INTEGER ,                                      -- num links to this doc type
   uids SEQUENCE OF INTEGER OPTIONAL ,                -- the links
   weights SEQUENCE OF INTEGER OPTIONAL }             -- the weights

PubMedId ::= INTEGER                               -- Id from the PubMed database

Cit-art ::= SEQUENCE {                             -- article in journal or book
   title Title OPTIONAL ,                          -- title of paper (ANSI requires)
   authors Auth-list OPTIONAL ,                    -- authors (ANSI requires)
   from CHOICE {                                   -- journal or book
      journal Cit-jour ,book Cit-book , proc Cit-proc } }

Cit-jour ::= SEQUENCE {                            -- Journal citation
   title Title ,                                   -- title of journal
   imp Imprint }

Cit-book ::= SEQUENCE {                            -- Book citation
   title Title ,                                   -- Title of book
   coll Title OPTIONAL ,                           -- part of a collection
   authors Auth-list,                              -- authors
   imp Imprint }

Cit-proc ::= SEQUENCE {                            -- Meeting proceedings
   book Cit-book ,                                 -- citation to meeting
   meet Meeting }                                  -- time and location of meeting

Cit-pat ::= SEQUENCE {                             -- patent citation
   title VisibleString ,
   authors Auth-list,                              -- author/inventor
   country VisibleString ,                         -- Patent Document Country
   doc-type VisibleString ,                         -- Patent Document Type
   number VisibleString ,                           -- Patent Document Number
   date-issue Date ,                               -- Patent Issue/Pub Date
   class SEQUENCE OF VisibleString OPTIONAL ,         -- Patent Doc Class Code
   app-number VisibleString OPTIONAL ,             -- Patent Doc Appl Number
   app-date Date OPTIONAL ,                        -- Patent Appl File Date
   applicants Auth-list OPTIONAL ,                 -- Applicants
   assignees Auth-list OPTIONAL ,                  -- Assignees
   priority SEQUENCE OF Patent-priority OPTIONAL ,    -- Priorities
   abstract VisibleString OPTIONAL }               -- abstract of patent

Patent-priority ::= SEQUENCE {
   country VisibleString ,                         -- Patent country code
   number VisibleString ,                          -- number assigned in that country
```

```
   date Date }                                           -- date of application

Id-pat ::= SEQUENCE {                                    -- just to identify a patent
   country VisibleString ,                               -- Patent Document Country
   id CHOICE {number VisibleString , app-number VisibleString } ,
   doc-type VisibleString OPTIONAL }                     -- Patent Doc Type

Cit-let ::= SEQUENCE {                                   -- letter, thesis, or manuscript
   cit Cit-book ,                                        -- same fields as a book
   man-id VisibleString OPTIONAL ,                       -- Manuscript identifier
   type ENUMERATED {
      manuscript (1) , letter (2) , thesis (3) } OPTIONAL }

Cit-sub ::= SEQUENCE {                                   -- citation for a direct submission
   authors Auth-list ,                               -- not necessarily authors
   imp Imprint OPTIONAL ,                            -- this only used to get date - will go
   medium ENUMERATED {                                   -- medium of submission
      paper   (1), tape    (2), floppy  (3), email   (4), other   (255) } OPTIONAL ,
   date Date OPTIONAL ,                                  -- replaces imp, will become required
   descr VisibleString OPTIONAL }                        -- changes for public view

Cit-gen ::= SEQUENCE {                                   -- NOT from ANSI, this is a catchall
   cit VisibleString OPTIONAL ,                          -- anything, not parsable
   authors Auth-list OPTIONAL ,
   muid INTEGER OPTIONAL ,                               -- medline uid
   journal Title OPTIONAL ,
   volume VisibleString OPTIONAL ,
   issue VisibleString OPTIONAL ,
   pages VisibleString OPTIONAL ,
   date Date OPTIONAL ,
   serial-number INTEGER OPTIONAL ,                      -- for GenBank style references
   title VisibleString OPTIONAL ,                        -- cit="unpublished",title="title"
         pmid PubMedId OPTIONAL }          -- PubMed Id

Auth-list ::= SEQUENCE {
      names CHOICE {
         std SEQUENCE OF Author ,                        -- full citations
         ml SEQUENCE OF VisibleString ,                  -- MEDLINE, semistructured
         str SEQUENCE OF VisibleString } ,               -- free for all
      affil Affil OPTIONAL }                             -- author affiliation

Author ::= SEQUENCE {
   name Person-id ,                                      -- Author, Primary or Secondary
   level ENUMERATED {
      primary (1), secondary (2) } OPTIONAL ,
```

```
    role ENUMERATED {                                -- Author Role Indicator
        compiler (1), editor (2), patent-assignee (3), translator (4) } OPTIONAL ,
    affil Affil OPTIONAL ,
    is-corr BOOLEAN OPTIONAL }                       -- TRUE if corressponding author

Affil ::= CHOICE {
    str VisibleString ,                              -- unparsed string
    std SEQUENCE {                                   -- std representation
    affil VisibleString OPTIONAL ,                   -- Author Affiliation, Name
    div VisibleString OPTIONAL ,                      -- Author Affiliation, Division
    city VisibleString OPTIONAL ,                    -- Author Affiliation, City
    sub VisibleString OPTIONAL ,                     -- Author Affiliation, County Sub
    country VisibleString OPTIONAL ,                 -- Author Affiliation, Country
    street VisibleString OPTIONAL ,                  -- street address, not ANSI
    email VisibleString OPTIONAL ,
    fax VisibleString OPTIONAL ,
    phone VisibleString OPTIONAL ,
    postal-code VisibleString OPTIONAL }}

Title ::= SET OF CHOICE {                            -- Valid for:
    name VisibleString ,                             -- Title, Anal,Coll,Mono    AJB
    tsub VisibleString ,                             -- Title, Subordinate       A B
    trans VisibleString ,                            -- Title, Translated        AJB
    jta VisibleString ,                              -- Title, Abbreviated        J
    iso-jta VisibleString ,                          -- specifically ISO jta      J
    ml-jta VisibleString ,                           -- specifically MEDLINE jta  J
    coden VisibleString ,                            -- a coden                   J
    issn VisibleString ,                             -- ISSN                      J
    abr VisibleString ,                              -- Title, Abbreviated          B
    isbn VisibleString }                             -- ISBN                        B

Imprint ::= SEQUENCE {                               -- Imprint group
    date Date ,                                      -- date of publication
    volume VisibleString OPTIONAL ,
    issue VisibleString OPTIONAL ,
    pages VisibleString OPTIONAL ,
    section VisibleString OPTIONAL ,
    pub Affil OPTIONAL,                              -- publisher, required for book
    cprt Date OPTIONAL,                              -- copyright date, "    "   "
    part-sup VisibleString OPTIONAL ,                -- part/sup of volume
    language VisibleString DEFAULT "ENG" ,           -- put here for simplicity
    prepub ENUMERATED {                              -- for prepublication citaions
        submitted (1), in-press (2), other (255)  } OPTIONAL ,
    part-supi VisibleString OPTIONAL ,               -- part/sup on issue
    retract CitRetract OPTIONAL }                    -- retraction info
```

```
CitRetract ::= SEQUENCE {
   type ENUMERATED {                              -- retraction of an entry
      retracted (1), notice (2), in-error (3), erratum (4) } ,
    exp VisibleString OPTIONAL }                  -- citation and/or explanation

Meeting ::= SEQUENCE {
   number VisibleString ,
   date Date ,
   place Affil OPTIONAL }

FeatDef ::= SEQUENCE {
   typelabel VisibleString ,                      -- short label for type eg "CDS"
   menulabel VisibleString ,                      -- label for a menu
   featdef-key INTEGER ,                          -- unique for this feature definition
   seqfeat-key INTEGER ,                          -- SeqFeat.data.choice from objfeat.h
   entrygroup INTEGER ,                           -- Group for data entry
   displaygroup INTEGER ,                         -- Group for data display
   molgroup FeatMolType                           -- Type of Molecule used for
}

FeatMolType ::= ENUMERATED {aa (1), na (2), both (3) }

FeatDefSet ::= SEQUENCE OF FeatDef

FeatDispGroup ::= SEQUENCE {
         groupkey INTEGER ,
         groupname VisibleString }

FeatDispGroupSet ::= SEQUENCE OF FeatDispGroup

FeatDefGroupSet ::= SEQUENCE {
         groups FeatDispGroupSet ,
         defs FeatDefSet }

Date ::= CHOICE {str VisibleString, std Date-std }

Date-std ::= SEQUENCE {                           -- this is NOT a unix tm struct
   year INTEGER ,                                 -- full year (including 1900)
   month INTEGER OPTIONAL ,                       -- month (1-12)
   day INTEGER OPTIONAL ,                         -- day of month (1-31)
   season VisibleString OPTIONAL }                -- for "spring", "may-june", etc

Dbtag ::= SEQUENCE {
   db VisibleString ,                             -- name of database or system
```

```
   tag Object-id }                                      -- appropriate tag

Object-id ::= CHOICE {id INTEGER, str VisibleString }

Person-id ::= CHOICE {
   dbtag Dbtag, name Name-std, ml VisibleString, str VisibleString }

Name-std ::= SEQUENCE {                    -- Structured names
   last VisibleString ,
   first VisibleString OPTIONAL ,
   middle VisibleString OPTIONAL ,
   full VisibleString OPTIONAL ,           -- full name eg. "J. John Poop, Esq"
   initials VisibleString OPTIONAL,        -- first + middle initials
   suffix VisibleString OPTIONAL ,         -- Jr, Sr, III
   title VisibleString OPTIONAL }          -- Dr., Sister, etc

Int-fuzz ::= CHOICE {
   p-m INTEGER ,                           -- plus or minus fixed amount
   range SEQUENCE {                        -- max to min
      max INTEGER ,
      min INTEGER } ,
   pct INTEGER ,                           -- % plus or minus (x10) 0-1000
   lim ENUMERATED {                        -- some limit value
      unk (0), gt (1), lt (2), tr (3), tl (4), circle (5), other (255) },
   alt SET OF INTEGER }                    -- set of alternatives for the integer

User-object ::= SEQUENCE {
   class VisibleString OPTIONAL ,   -- endeavor which designed this object
   type Object-id ,                        -- type of object within class
   data SEQUENCE OF User-field }           -- the object itself

User-field ::= SEQUENCE {
   label Object-id ,                       -- field label
   num INTEGER OPTIONAL ,                  -- required for strs, ints, reals, oss
   data CHOICE {                           -- field contents
      str VisibleString, int INTEGER, real REAL, bool BOOLEAN, os OCTET STRING,
      object User-object, strs SEQUENCE OF VisibleString,
      ints SEQUENCE OF INTEGER, reals SEQUENCE OF REAL ,
      oss SEQUENCE OF OCTET STRING, fields SEQUENCE OF User-field ,
      objects SEQUENCE OF User-object } }

Medline-entry ::= SEQUENCE {
   uid INTEGER OPTIONAL ,                  -- MEDLINE UID, not always available
   em Date ,                               -- Entry Month
   cit Cit-art ,                           -- article citation
```

```
   abstract VisibleString OPTIONAL ,
   mesh SET OF Medline-mesh OPTIONAL ,
   substance SET OF Medline-rn OPTIONAL ,
   xref SET OF Medline-si OPTIONAL ,
   idnum SET OF VisibleString OPTIONAL ,                -- ID Number
   gene SET OF VisibleString OPTIONAL ,
         pmid PubMedId OPTIONAL ,                       -- may include PubMedId
         pub-type SET OF VisibleString OPTIONAL }       -- may show pub types

Medline-mesh ::= SEQUENCE {
   mp BOOLEAN DEFAULT FALSE ,                 -- TRUE if main point (*)
   term VisibleString ,                       -- the MeSH term
   qual SET OF Medline-qual OPTIONAL )        -- qualifiers

Medline-qual ::= SEQUENCE {
   mp BOOLEAN DEFAULT FALSE ,                 -- TRUE if main point
   subh VisibleString }                       -- the subheading

Medline-rn ::= SEQUENCE {                     -- medline substance records
   type ENUMERATED {                          -- type of record
      nameonly (0), cas (1), ec (2) },
   cit VisibleString OPTIONAL ,               -- CAS or EC number if present
   name VisibleString }                       -- name (always present)

Medline-si ::= SEQUENCE {                     -- medline cross reference records
   type ENUMERATED {                          -- type of xref
      ddbj (1), carbbank (2), embl (3), hdb (4), genbank (5), hgml (6), mim (7),
      msd (8), pdb (9), pir (10), prfseqdb (11), psd (12),swissprot (13) } ,
   cit VisibleString OPTIONAL }               -- the citation/accession number

Ncbi-mime-asn1 ::= CHOICE {entrez Entrez-general}

Entrez-style ::= ENUMERATED {
         docsum (1), genbank (2), genpept (3), fasta (4), asn1 (5), graphic (6),
         alignment (7), globalview (8), report (9), medlars (10), embl (11) ,
         pdb (12), kinemage (13) }

Entrez-general ::= SEQUENCE {
         title VisibleString OPTIONAL,
         data CHOICE {
                  ml  Medline-entry, prot Seq-entry , nuc Seq-entry, genome Seq-entry ,
                  structure Biostruc, strucAnnot Biostruc-annot-set } ,
         style Entrez-style ,
         location VisibleString OPTIONAL }
```

```
PrintTemplate ::= SEQUENCE {
   name TemplateName ,                          -- name for this template
   labelfrom VisibleString OPTIONAL,            -- ASN.1 path to get label from
   format PrintFormat }

TemplateName ::= VisibleString

PrintTemplateSet ::= SEQUENCE OF PrintTemplate

PrintFormat ::= SEQUENCE {
   asn1 VisibleString ,                         -- ASN.1 partial path for this
   label VisibleString OPTIONAL ,          -- printable label
   prefix VisibleString OPTIONAL,
   suffix VisibleString OPTIONAL,
   form PrintForm }

PrintForm ::=   CHOICE {                        -- Forms for various ASN.1 cmpnts
   block PrintFormBlock, boolean PrintFormBoolean, enum PrintFormEnum,
   text PrintFormText, use-template TemplateName, user UserFormat ,
   null NULL }

UserFormat ::= SEQUENCE {
   printfunc VisibleString ,
   defaultfunc VisibleString OPTIONAL }

PrintFormBlock ::= SEQUENCE {                   -- for SEQUENCE, SET
   separator VisibleString OPTIONAL ,
   components SEQUENCE OF PrintFormat }

PrintFormBoolean ::= SEQUENCE {
   true VisibleString OPTIONAL ,
   false VisibleString OPTIONAL }

PrintFormEnum ::= SEQUENCE {values SEQUENCE OF VisibleString OPTIONAL }

PrintFormText ::= SEQUENCE {textfunc VisibleString OPTIONAL }

Pub ::= CHOICE {
   gen Cit-gen, sub Cit-sub, medline Medline-entry, muid INTEGER, article Cit-art ,
   journal Cit-jour, book Cit-book, proc Cit-proc, patent Cit-pat, pat-id Id-pat,
   man Cit-let, equiv Pub-equiv, pmid PubMedId }

Pub-equiv ::= SET OF Pub                        -- equivalent ids for same citation

Pub-set ::= CHOICE {
```

```
   pub SET OF Pub, medline SET OF Medline-entry, article SET OF Cit-art ,
   journal SET OF Cit-jour, book SET OF Cit-book, proc SET OF Cit-proc,
   patent SET OF Cit-pat }

Bioseq ::= SEQUENCE {
   id SET OF Seq-id ,                           -- equivalent identifiers
   descr Seq-descr OPTIONAL ,                   -- descriptors
   inst Seq-inst ,                              -- the sequence data
   annot SET OF Seq-annot OPTIONAL }

Seq-descr ::= SET OF Seqdesc

Seqdesc ::= CHOICE {
   mol-type GIBB-mol, modif SET OF GIBB-mod, method GIBB-method,
   name VisibleString, title VisibleString, org Org-ref, comment VisibleString,
   num Numbering, maploc Dbtag, pir PIR-block, genbank GB-block, pub Pubdesc,
   region VisibleString, user User-object, sp SP-block, dbxref Dbtag, embl EMBL-block,
   create-date Date, update-date Date, prf PRF-block, pdb PDB-block, het Heterogen,
   source BioSource, molinfo MolInfo}

MolInfo ::= SEQUENCE {
   biomol INTEGER {
      unknown (0), genomic (1), pre-RNA (2) , mRNA (3), rRNA (4), tRNA (5),
      snRNA (6), scRNA (7), peptide (8), other-genetic (9), genomic-mRNA (10),
     other (255) } DEFAULT unknown ,
   tech INTEGER {
      unknown (0), standard (1), est (2), sts (3), survey (4), genemap (5), physmap (6),
      derived (7), concept-trans (8), seq-pept (9), both (10), seq-pept-overlap (11),
      seq-pept-homol (12), concept-trans-a (13), other (255) } DEFAULT unknown ,
   techexp VisibleString OPTIONAL ,             -- explanation if tech not enough
   completeness INTEGER {
    unknown (0), complete (1), partial (2), no-left (3), no-right (4),
    no-ends (5), other (255) } DEFAULT unknown }


GIBB-mol ::= ENUMERATED {                       -- type of molecule represented
   unknown (0), genomic (1), pre-mRNA (2), mRNA (3), rRNA (4), tRNA (5),
   snRNA (6), scRNA (7), peptide (8), other-genetic (9), genomic-mRNA (10),
   other (255) }

GIBB-mod ::= ENUMERATED {                       -- GenInfo Backbone modifiers
   dna (0), rna (1), extrachrom (2), plasmid (3), mitochondrial (4), chloroplast (5),
   kinetoplast (6), cyanelle (7), synthetic (8), recombinant (9), partial (10), complete (11),
   mutagen (12), natmut (13), transposon (14), insertion-seq (15), no-left (16),
   no-right (17), macronuclear (18), proviral (19), est (20), sts (21), survey (22),
```

```
    chromoplast (23), genemap (24), restmap (25), physmap (26), other (255) }

GIBB-method ::= ENUMERATED {                    -- sequencing methods
    concept-trans (1), seq-pept (2), both (3), seq-pept-overlap (4), seq-pept-homol (5),
    concept-trans-a (6), other (255) }

Numbering ::= CHOICE {                          -- any display numbering system
    cont Num-cont, enum Num-enum, ref Num-ref, real Num-real }

Num-cont ::= SEQUENCE {                         -- continuous numbering system
    refnum INTEGER DEFAULT 1,                   -- number assigned to first residue
    has-zero BOOLEAN DEFAULT FALSE ,            -- 0 used?
    ascending BOOLEAN DEFAULT TRUE }            -- ascending numbers?

Num-enum ::= SEQUENCE {                         -- any tags to residues
    num INTEGER ,                               -- number of tags to follow
    names SEQUENCE OF VisibleString }           -- the tags

Num-ref ::= SEQUENCE {                          -- by reference to other sequences
    type ENUMERATED {                           -- type of reference
        not-set (0), sources (1), aligns (2) },
    aligns Seq-align OPTIONAL }

Num-real ::= SEQUENCE {                         -- mapping to floating point system
    a REAL ,                                    -- integer system used by Bioseq
    b REAL ,                                    -- position = (a * int_position) + b
    units VisibleString OPTIONAL }

Pubdesc ::= SEQUENCE {                          -- how sequence presented in pub
    pub Pub-equiv ,                             -- the citation(s)
    name VisibleString OPTIONAL ,               -- name used in paper
    fig VisibleString OPTIONAL ,                -- figure in paper
    num Numbering OPTIONAL ,                    -- numbering from paper
    numexc BOOLEAN OPTIONAL ,                   -- numbering problem with paper
    poly-a BOOLEAN OPTIONAL ,                   -- poly A tail indicated in figure?
    maploc VisibleString OPTIONAL ,             -- map location reported in paper
    seq-raw StringStore OPTIONAL ,              -- original sequence from paper
    align-group INTEGER OPTIONAL ,              -- seq aligned with others in paper
    comment VisibleString OPTIONAL,             -- any comment pub in context
        reftype INTEGER {                       -- type of ref in a GenBank record
            seq (0) , sites (1), feats (2) } DEFAULT seq }

Heterogen ::= VisibleString                     -- cofactor, prosthetic group, inibitor

Seq-inst ::= SEQUENCE {                         -- the sequence data itself
```

```
   repr ENUMERATED {                              -- representation class
      not-set (0), virtual (1), raw (2), seg (3), const (4), ref (5), consen (6),
      map (7), delta (8), other (255) } ,
   mol ENUMERATED {                               -- molecule class in living organism
      not-set (0), dna (1), rna (2), aa (3), na (4) ,other (255) } ,
   length INTEGER OPTIONAL ,                      -- length of sequence in residues
   fuzz Int-fuzz OPTIONAL ,                       -- length uncertainty
   topology ENUMERATED {                          -- topology of molecule
      not-set (0), linear (1), circular (2), tandem (3), other (255) } DEFAULT linear ,
   strand ENUMERATED {                            -- strandedness in living organism
      not-set (0), ss (1), ds (2), mixed (3), other (255) } OPTIONAL ,
   seq-data Seq-data OPTIONAL ,                   -- the sequence
   ext Seq-ext OPTIONAL ,                         -- extensions for special types
   hist Seq-hist OPTIONAL }                       -- sequence history

Seq-ext ::= CHOICE {
   seg Seg-ext, ref Ref-ext, map Map-ext, delta Delta-ext }

Seg-ext ::= SEQUENCE OF Seq-loc
Ref-ext ::= Seq-loc
Map-ext ::= SEQUENCE OF Seq-feat
Delta-ext ::= SEQUENCE OF Delta-seq

Delta-seq ::= CHOICE {loc Seq-loc, literal Seq-literal }

Seq-literal ::= SEQUENCE {
   length INTEGER ,                       -- must give a length in residues
   fuzz Int-fuzz OPTIONAL ,               -- could be unsure
   seq-data Seq-data OPTIONAL }           -- may have the data

Seq-hist ::= SEQUENCE {
   assembly SET OF Seq-align OPTIONAL ,-- how was this assembled?
   replaces Seq-hist-rec OPTIONAL ,       -- seq makes these seqs obsolete
   replaced-by Seq-hist-rec OPTIONAL ,    -- these seqs make this one obsolete
   deleted CHOICE {bool BOOLEAN, date Date } OPTIONAL }

Seq-hist-rec ::= SEQUENCE {
   date Date OPTIONAL ,
   ids SET OF Seq-id }

Seq-data ::= CHOICE {                     -- sequence representations
   iupacna IUPACna, iupacaa IUPACaa, ncbi2na NCBI2na, ncbi4na NCBI4na,
   ncbi8na NCBI8na, ncbipna NCBIpna, ncbi8aa NCBI8aa, ncbieaa NCBIeaa,
   ncbipaa NCBIpaa, ncbistdaa NCBIstdaa}
```

```
IUPACna ::= StringStore                -- IUPAC 1 letter codes, no spaces
IUPACaa ::= StringStore                -- IUPAC 1 letter codes, no spaces
NCBI2na ::= OCTET STRING               -- 00=A, 01=C, 10=G, 11=T
NCBI4na ::= OCTET STRING               -- 1 bit each for agct
NCBI8na ::= OCTET STRING               -- for modified nucleic acids
NCBIpna ::= OCTET STRING               -- 5 octets/base, prob for a,c,g,t,n
NCBI8aa ::= OCTET STRING               -- for modified amino acids
NCBIeaa ::= StringStore                -- ASCII extended 1 letter aa codes
NCBIpaa ::= OCTET STRING               -- 25 octets/aa, prob for IUPAC aas in order:
NCBIstdaa ::= OCTET STRING             -- codes 0-25, 1 per byte

Annot-id ::= CHOICE {
  local Object-id, ncbi INTEGER, general Dbtag }

Annot-descr ::= SET OF Annotdesc

Annotdesc ::= CHOICE {
  name VisibleString, title VisibleString, comment VisibleString, pub Pubdesc,
  user User-object, create-date Date, update-date Date, src Seq-id, align Align-def }

Align-def ::= SEQUENCE {
  align-type INTEGER {                      -- class of align Seq-annot
   ref (1), alt (2), blocks (3), other (255) } ,
  ids SET OF Seq-id OPTIONAL }              -- used for the one ref seqid for now

Seq-annot ::= SEQUENCE {
  id SET OF Annot-id OPTIONAL ,
  db INTEGER {                              -- source of annotation
    genbank (1), embl (2), ddbj (3), pir  (4), sp   (5), bbone (6), pdb   (7),
    other (255) } OPTIONAL ,
  name VisibleString OPTIONAL ,             -- source if "other" above
  desc Annot-descr OPTIONAL ,               -- only for stand alone Seq-annots
  data CHOICE {
    ftable SET OF Seq-feat, align SET OF Seq-align, graph SET OF Seq-graph,
    ids SET OF Seq-id, locs SET OF Seq-loc }

Seq-align-set ::= SET OF Seq-align

Seq-align ::= SEQUENCE {
  type ENUMERATED {
    not-set (0), global (1), diags (2), partial (3), disc (4), other (255) } ,
  dim INTEGER OPTIONAL ,                    -- dimensionality
  score SET OF Score OPTIONAL ,             -- for whole alignment
  segs CHOICE {                             -- alignment data
```

```
    dendiag SEQUENCE OF Dense-diag, denseg Dense-seg,
    std SEQUENCE OF Std-seg,packed Packed-seg, disc Seq-align-set } ,
  bounds SET OF Seq-loc OPTIONAL }

Dense-diag ::= SEQUENCE {                       -- for (multiway) diagonals
  dim INTEGER DEFAULT 2 ,                       -- dimensionality
  ids SEQUENCE OF Seq-id ,                      -- sequences in order
  starts SEQUENCE OF INTEGER ,                  -- start OFFSETS in ids order
  len INTEGER ,                                 -- len of aligned segments
  strands SEQUENCE OF Na-strand OPTIONAL ,
  scores SET OF Score OPTIONAL }

Dense-seg ::= SEQUENCE {                        -- for (multiway) global or partial
  dim INTEGER DEFAULT 2 ,                       -- dimensionality
  numseg INTEGER ,                              -- number of segments here
  ids SEQUENCE OF Seq-id ,                      -- sequences in order
  starts SEQUENCE OF INTEGER ,                  -- OFFSETS in ids order within segs
  lens SEQUENCE OF INTEGER ,                    -- lengths in ids order within segs
  strands SEQUENCE OF Na-strand OPTIONAL ,
  scores SEQUENCE OF Score OPTIONAL }           -- score for each seg

Packed-seg ::= SEQUENCE {                       -- for (multiway) global or partial
  dim INTEGER DEFAULT 2 ,                       -- dimensionality
  numseg INTEGER ,                              -- number of segments here
  ids SEQUENCE OF Seq-id ,                      -- sequences in order
  starts SEQUENCE OF INTEGER ,                  -- start OFFSETS in ids order
  present OCTET STRING ,                        -- sequence present or absent in
  lens SEQUENCE OF INTEGER ,                    -- length of each segment
  strands SEQUENCE OF Na-strand OPTIONAL ,
  scores SEQUENCE OF Score OPTIONAL }           -- score for each segment

Std-seg ::= SEQUENCE {
  dim INTEGER DEFAULT 2 ,                       -- dimensionality
  ids SEQUENCE OF Seq-id OPTIONAL ,
  loc SEQUENCE OF Seq-loc ,
  scores SET OF Score OPTIONAL }

Score ::= SEQUENCE {
  id Object-id OPTIONAL ,
  value CHOICE {
    real REAL, int INTEGER  } }

EMBL-dbname ::= CHOICE {
  code ENUMERATED {
    embl(0), genbank(1), ddbj(2), geninfo(3), medline(4), swissprot(5), pir(6), pdb(7),
```

```
        epd(8), ecd(9), tfd(10), flybase(11), prosite(12), enzyme(13), mim(14), ecoseq(15),
        hiv(16), other (255) } ,
    name    VisibleString }

EMBL-xref ::= SEQUENCE {
    dbname EMBL-dbname,
    id SEQUENCE OF Object-id }

EMBL-block ::= SEQUENCE {
    class ENUMERATED {
        not-set(0), standard(1), unannotated(2), other(255) } DEFAULT standard,
    div ENUMERATED {
        fun(0), inv(1), mam(2), org(3), phg(4), pln(5), pri(6), pro(7), rod(8), syn(9),
        una(10), vrl(11), vrt(12), pat(13), est(14), sts(15), other (255) } OPTIONAL,
    creation-date Date,
    update-date Date,
    extra-acc SEQUENCE OF VisibleString OPTIONAL,
    keywords SEQUENCE OF VisibleString OPTIONAL,
    xref SEQUENCE OF EMBL-xref OPTIONAL }


SP-block ::= SEQUENCE {                              -- SWISSPROT specific descriptions
    class ENUMERATED {
        not-set (0), standard (1), prelim (2), other (255) } ,
    extra-acc SET OF VisibleString OPTIONAL ,        -- old SWISSPROT ids
    imeth BOOLEAN DEFAULT FALSE ,                    -- seq known to start with Met
    plasnm SET OF VisibleString OPTIONAL,            -- plasmid names carrying gene
    seqref SET OF Seq-id OPTIONAL,                   -- xref to other sequences
    dbref SET OF Dbtag OPTIONAL ,                    -- xref to nonsequence dbases
    keywords SET OF VisibleString OPTIONAL ,         -- keywords
    created Date OPTIONAL ,                          -- creation date
    sequpd Date OPTIONAL ,                           -- sequence update
    annotupd Date OPTIONAL }                         -- annotation update

PIR-block ::= SEQUENCE {                             -- PIR specific descriptions
    had-punct BOOLEAN OPTIONAL ,                     -- had punctuation in sequence ?
    host VisibleString OPTIONAL ,
    source VisibleString OPTIONAL ,                  -- source line
    summary VisibleString OPTIONAL ,
    genetic VisibleString OPTIONAL ,
    includes VisibleString OPTIONAL ,
    placement VisibleString OPTIONAL ,
    superfamily VisibleString OPTIONAL ,
    keywords SEQUENCE OF VisibleString OPTIONAL ,
    cross-reference VisibleString OPTIONAL ,
```

```
   date VisibleString OPTIONAL ,
   seq-raw VisibleString OPTIONAL ,              -- seq with punctuation
   seqref SET OF Seq-id OPTIONAL }               -- xref to other sequences

GB-block ::= SEQUENCE {                          -- GenBank specific descriptions
   extra-accessions SEQUENCE OF VisibleString OPTIONAL ,
   source VisibleString OPTIONAL ,               -- source line
   keywords SEQUENCE OF VisibleString OPTIONAL ,
   origin VisibleString OPTIONAL,
   date VisibleString OPTIONAL ,                 -- OBSOLETE old form Entry Date
   entry-date Date OPTIONAL ,                    -- replaces date
   div VisibleString OPTIONAL ,                  -- GenBank division
   taxonomy VisibleString OPTIONAL }             -- continuation line of organism

PRF-block ::= SEQUENCE {
    extra-src      PRF-ExtraSrc OPTIONAL,
    keywords       SEQUENCE OF VisibleString OPTIONAL}

PRF-ExtraSrc ::= SEQUENCE {
    host    VisibleString OPTIONAL,
    part    VisibleString OPTIONAL,
    state   VisibleString OPTIONAL,
    strain  VisibleString OPTIONAL,
    taxon   VisibleString OPTIONAL}

PDB-block ::= SEQUENCE {                         -- PDB specific descriptions
   deposition Date ,                             -- deposition date  month,year
   class VisibleString ,
   compound SEQUENCE OF VisibleString ,
   source SEQUENCE OF VisibleString ,
   exp-method VisibleString OPTIONAL ,           -- present if NOT X-ray diffraction
   replace PDB-replace OPTIONAL }                -- replacement history

PDB-replace ::= SEQUENCE {
   date Date ,
   ids SEQUENCE OF VisibleString }               -- entry ids replace by this one

Seq-code-type ::= ENUMERATED {                   -- sequence representations
   iupacna (1), iupacaa (2), ncbi2na (3), ncbi4na (4), ncbi8na (5), ncbipna (6),
   ncbi8aa (7), ncbieaa (8), ncbipaa (9), iupacaa3 (10), ncbistdaa (11) }

Seq-map-table ::= SEQUENCE {                     -- for tables of sequence mappings
   from Seq-code-type ,                          -- code to map from
   to Seq-code-type ,                            -- code to map to
   num INTEGER ,                                 -- number of rows in table
```

```
   start-at INTEGER DEFAULT 0 ,                    -- index offset of first element
   table SEQUENCE OF INTEGER }                     -- table of values, in from-to order

Seq-code-table ::= SEQUENCE {                      -- for names of coded values
   code Seq-code-type ,                            -- name of code
   num INTEGER ,                                   -- number of rows in table
   one-letter BOOLEAN ,                            -- symbol is ALWAYS 1 letter?
   start-at INTEGER DEFAULT 0 ,                    -- index offset of first element
   table SEQUENCE OF
      SEQUENCE {
         symbol VisibleString ,                    -- the printed symbol or letter
         name VisibleString } ,                    -- an explanatory name or string
   comps SEQUENCE OF INTEGER OPTIONAL } -- pointers to complement nuc acid

Seq-code-set ::= SEQUENCE {    -- for distribution
   codes SET OF Seq-code-table OPTIONAL ,
   maps SET OF Seq-map-table OPTIONAL }

Feat-id ::= CHOICE {
   gibb INTEGER ,                                  -- geninfo backbone
   giim Giimport-id ,                              -- geninfo import
   local Object-id ,                               -- for local software use
   general Dbtag }                                 -- for use by various databases

Seq-feat ::= SEQUENCE {
   id Feat-id OPTIONAL ,
   data SeqFeatData ,                              -- the specific data
   partial BOOLEAN OPTIONAL ,                      -- incomplete in some way?
   except BOOLEAN OPTIONAL ,                       -- something funny about this?
   comment VisibleString OPTIONAL ,
   product Seq-loc OPTIONAL ,                      -- product of process
   location Seq-loc ,                              -- feature made from
   qual SEQUENCE OF Gb-qual OPTIONAL ,             -- qualifiers
   title VisibleString OPTIONAL ,                  -- for user defined label
   ext User-object OPTIONAL ,                      -- user defined structure extension
   cit Pub-set OPTIONAL ,                          -- citations for this feature
   exp-ev ENUMERATED {                             -- evidence for existence of feature
      experimental (1), not-experimental (2) } OPTIONAL ,
   xref SET OF SeqFeatXref OPTIONAL ,              -- cite other relevant features
  dbxref SET OF Dbtag OPTIONAL }                   -- support for xref to other databases

SeqFeatData ::= CHOICE {
   gene Gene-ref, org Org-ref, cdregion Cdregion, prot Prot-ref, rna RNA-ref,
   pub Pubdesc, seq Seq-loc, imp Imp-feat, region VisibleString, comment NULL,
   bond ENUMERATED {disulfide (1), thiolester (2), xlink (3), thioether (4) ,
```

```
      other (255) } ,
   site ENUMERATED {active (1), binding (2), cleavage (3), inhibit (4), modified (5),
      glycosylation (6), myristoylation (7), mutagenized (8), metal-binding (9),
      phosphorylation (10), acetylation (11), amidation (12), methylation (13),
      hydroxylation (14), sulfatation (15), oxidative-deamination (16),
      pyrrolidone-carboxylic-acid (17), gamma-carboxyglutamic-acid (18),
      blocked (19), lipid-binding (20), np-binding (21), dna-binding (22),
      signal-peptide (23), transit-peptide (24), transmembrane-region (25), other (255) } ,
   rsite Rsite-ref , user User-object, txinit Txinit, num Numbering,
   psec-str ENUMERATED {helix (1) , sheet (2), turn  (3) },
   non-std-residue VisibleString, het Heterogen, biosrc BioSource }

SeqFeatXref ::= SEQUENCE {                  -- both because can have one or both
   id Feat-id OPTIONAL ,                    -- the feature copied
   data SeqFeatData OPTIONAL }              -- the specific data

Cdregion ::= SEQUENCE {
   orf BOOLEAN OPTIONAL ,                   -- just an ORF ?
   frame ENUMERATED {
      not-set (0), one (1), two (2), three (3) } DEFAULT not-set ,
   conflict BOOLEAN OPTIONAL ,              -- conflict
   gaps INTEGER OPTIONAL ,                  -- number of gaps on conflict/except
   mismatch INTEGER OPTIONAL ,              -- number of mismatches on above
   code Genetic-code OPTIONAL ,             -- genetic code used
   code-break SEQUENCE OF Code-break OPTIONAL ,   -- individual exceptions
   stops INTEGER OPTIONAL }                 -- number of stop codons on above

Genetic-code ::= SET OF CHOICE {
   name VisibleString ,                     -- name of a code
   id INTEGER ,                             -- id in dbase
   ncbieaa VisibleString ,                  -- indexed to IUPAC extended
   ncbi8aa OCTET STRING ,                   -- indexed to NCBI8aa
   ncbistdaa OCTET STRING ,                 -- indexed to NCBIstdaa
   sncbieaa VisibleString ,                 -- start, indexed to IUPAC extended
   sncbi8aa OCTET STRING ,                  -- start, indexed to NCBI8aa
   sncbistdaa OCTET STRING }                -- start, indexed to NCBIstdaa

Code-break ::= SEQUENCE {                   -- specific codon exceptions
   loc Seq-loc ,                            -- location of exception
   aa CHOICE {                              -- the amino acid
      ncbieaa INTEGER, ncbi8aa INTEGER, ncbistdaa INTEGER } }

Genetic-code-table ::= SET OF Genetic-code  -- table of genetic codes

Imp-feat ::= SEQUENCE {
```

```
   key VisibleString ,
   loc VisibleString OPTIONAL ,                -- original location string
   descr VisibleString OPTIONAL }              -- text description

Gb-qual ::= SEQUENCE {
   qual VisibleString ,
   val VisibleString }

Rsite-ref ::= CHOICE {
   str VisibleString ,                         -- may be unparsable
   db  Dbtag }                                 -- ptr to a restriction site database

RNA-ref ::= SEQUENCE {
   type ENUMERATED {                           -- type of RNA feature
      unknown (0), premsg (1), mRNA (2), tRNA (3), rRNA (4), snRNA (5),
      scRNA (6), other (255) } ,
   pseudo BOOLEAN OPTIONAL ,
   ext CHOICE {
      name VisibleString, tRNA Trna-ext } OPTIONAL }

Trna-ext ::= SEQUENCE {                        -- tRNA feature extensions
   aa CHOICE {                                 -- aa this carries
      iupacaa INTEGER, ncbieaa INTEGER, ncbi8aa INTEGER,
      ncbistdaa INTEGER } OPTIONAL ,
   codon SET OF INTEGER OPTIONAL ,             -- codon(s) as in Genetic-code
         anticodon Seq-loc OPTIONAL }          -- location of anticodon

Gene-ref ::= SEQUENCE {
   locus VisibleString OPTIONAL ,              -- Official gene symbol
   allele VisibleString OPTIONAL ,             -- Official allele designation
   desc VisibleString OPTIONAL ,               -- descriptive name
   maploc VisibleString OPTIONAL ,             -- descriptive map location
   pseudo BOOLEAN DEFAULT FALSE ,              -- pseudogene
   db SET OF Dbtag OPTIONAL ,                  -- ids in other dbases
   syn SET OF VisibleString OPTIONAL }         -- synonyms for locus

Org-ref ::= SEQUENCE {
   taxname VisibleString OPTIONAL ,            -- preferred formal name
   common VisibleString OPTIONAL ,             -- common name
   mod SET OF VisibleString OPTIONAL ,         -- unstructured modifiers
   db SET OF Dbtag OPTIONAL ,                  -- ids in taxonomic or culture dbases
   syn SET OF VisibleString OPTIONAL ,         -- synonyms for taxname or common
   orgname OrgName OPTIONAL }
```

```
OrgName ::= SEQUENCE {
    name CHOICE {
        binomial BinomialOrgName, virus VisibleString, hybrid MultiOrgName,
        namedhybrid BinomialOrgName, partial PartialOrgName } OPTIONAL ,
    attrib VisibleString OPTIONAL ,              -- attribution of name
    mod SEQUENCE OF OrgMod OPTIONAL ,
    lineage VisibleString OPTIONAL ,             -- lineage with semicolon separators
    gcode INTEGER OPTIONAL ,                     -- genetic code (see CdRegion)
    mgcode INTEGER OPTIONAL ,                    -- mitochondrial genetic code
    div VisibleString OPTIONAL }                 -- GenBank division code


OrgMod ::= SEQUENCE {
    subtype INTEGER {
        strain (2), substrain (3), type (4), subtype (5), variety (6), serotype (7), serogroup (8),
        serovar (9), cultivar (10), pathovar (11), chemovar (12), biovar (13), biotype (14),
        group (15), subgroup (16), isolate (17), common (18), acronym (19), dosage (20),
        nat-host (21), sub-species (22), other (255) },
    subname VisibleString ,
    attrib VisibleString OPTIONAL }              -- attribution/source of name

BinomialOrgName ::= SEQUENCE {
    genus VisibleString ,                        -- required
    species VisibleString OPTIONAL ,             -- species required if subspecies used
    subspecies VisibleString OPTIONAL }

MultiOrgName ::= SEQUENCE OF OrgName            -- first will be used to assign division

PartialOrgName ::= SEQUENCE OF TaxElement       -- when we don't know the genus

TaxElement ::= SEQUENCE {
    fixed-level INTEGER {
       other (0), family (1), order (2), class (3) } ,
    level VisibleString OPTIONAL ,
    name VisibleString }

BioSource ::= SEQUENCE {
    genome INTEGER {                             -- biological context
        unknown (0), genomic (1), chloroplast (2), chromoplast (3), kinetoplast (4),
        mitochondrion (5), plastid (6), macronuclear (7), extrachrom (8), plasmid (9),
        transposon (10), insertion-seq (11), cyanelle (12), proviral (13),
       virion (14) } DEFAULT unknown ,
    origin INTEGER {
      unknown (0), natural (1), natmut (2), mut (3), artificial (4),
      synthetic (5), other (255) } DEFAULT unknown ,
```

```
    org Org-ref ,
    subtype SEQUENCE OF SubSource OPTIONAL }

SubSource ::= SEQUENCE {
    subtype INTEGER {
        chromosome (1), map (2), clone (3), subclone (4), haplotype (5), genotype (6),
        sex (7), cell-line (8), cell-type (9), tissue-type (10), clone-lib (11), dev-stage (12),
        frequency (13), germline (14), rearranged (15), lab-host (16), pop-variant (17),
        tissue-lib (18), plasmid-name (19), transposon-name (20), insertion-seq-name (21),
        plastid-name (22), other (255) } ,
    name VisibleString ,
    attrib VisibleString OPTIONAL }                 -- attribution/source of this name

Prot-ref ::= SEQUENCE {
    name SET OF VisibleString OPTIONAL ,            -- protein name
    desc VisibleString OPTIONAL ,                   -- description (instead of name)
    ec SET OF VisibleString OPTIONAL ,              -- E.C. number(s)
    activity SET OF VisibleString OPTIONAL ,        -- activities
    db SET OF Dbtag OPTIONAL ,                      -- ids in other dbases
    processed ENUMERATED {                          -- processing status
       not-set (0), preprotein (1), mature (2), signal-peptide (3),
       transit-peptide (4) } DEFAULT not-set }

Txinit ::= SEQUENCE {
    name VisibleString ,    -- descriptive name of initiation site
    syn SEQUENCE OF VisibleString OPTIONAL,         -- synonyms
    gene SEQUENCE OF Gene-ref OPTIONAL ,            -- gene(s) transcribed
    protein SEQUENCE OF Prot-ref OPTIONAL ,         -- protein(s) produced
    rna SEQUENCE OF VisibleString OPTIONAL ,        -- rna(s) produced
    expression VisibleString OPTIONAL ,             -- tissue/time of expression
    txsystem ENUMERATED {                           -- transcription apparatus used
        unknown (0) ,
        pol1 (1), pol2 (2), pol3 (3), bacterial (4), viral (5), rna (6),
        organelle (7), other (255) } ,
    txdescr VisibleString OPTIONAL ,                -- modifiers on txsystem
    txorg Org-ref OPTIONAL ,                        -- org spply transcription apparatus
    mapping-precise BOOLEAN DEFAULT FALSE ,              -- mapping precise or approx
    location-accurate BOOLEAN DEFAULT FALSE ,            -- Seq-loc reflect mapping
    inittype ENUMERATED {
        unknown (0), single (1), multiple (2), region (3) } OPTIONAL ,
    evidence SET OF Tx-evidence OPTIONAL }

Tx-evidence ::= SEQUENCE {
    exp-code ENUMERATED {
        unknown (0), rna-seq (1), rna-size (2), np-map (3), np-size (4), pe-seq (5),
```

```
        cDNA-seq (6), pe-map (7), pe-size (8), pseudo-seq (9), rev-pe-map (10),
        other (255) } ,
    expression-system ENUMERATED {
        unknown (0), physiological (1), in-vitro (2), oocyte (3), transfection (4),
        transgenic (5), other (255) } DEFAULT physiological ,
    low-prec-data BOOLEAN DEFAULT FALSE ,
    from-homolog BOOLEAN DEFAULT FALSE }  -- experiment actually done on

Seq-id ::= CHOICE {
    local Object-id, gibbsq INTEGER, gibbmt INTEGER, giim Giimport-id,
    genbank Textseq-id, embl Textseq-id, pir Textseq-id, swissprot Textseq-id,
    patent Patent-seq-id, other Textseq-id, general Dbtag, gi INTEGER,
    ddbj Textseq-id, prf Textseq-id, pdb PDB-seq-id }

Patent-seq-id ::= SEQUENCE {
    seqid INTEGER ,                          -- number of sequence in patent
    cit Id-pat }                             -- patent citation

Textseq-id ::= SEQUENCE {
    name VisibleString OPTIONAL ,
    accession VisibleString OPTIONAL ,
    release VisibleString OPTIONAL ,
    version INTEGER OPTIONAL }

Giimport-id ::= SEQUENCE {
    id INTEGER ,                             -- the id to use here
    db VisibleString OPTIONAL ,              -- dbase used in
    release VisibleString OPTIONAL }         -- the release

PDB-seq-id ::= SEQUENCE {
    mol PDB-mol-id ,                         -- the molecule name
    chain INTEGER DEFAULT 32 ,               -- a single ASCII character, chain id
    rel Date OPTIONAL }                      -- release date, month and year

PDB-mol-id ::= VisibleString                 -- name of mol, 4 chars

Seq-loc ::= CHOICE {
    null NULL, empty Seq-id, whole Seq-id, int Seq-interval, packed-int Packed-seqint,
    pnt Seq-point, packed-pnt Packed-seqpnt, mix Seq-loc-mix, equiv Seq-loc-equiv,
    bond Seq-bond, feat Feat-id }

Seq-interval ::= SEQUENCE {
    from INTEGER ,
    to INTEGER ,
    strand Na-strand OPTIONAL ,
```

```
    id Seq-id ,
    fuzz-from Int-fuzz OPTIONAL ,
    fuzz-to Int-fuzz OPTIONAL }

Packed-seqint ::= SEQUENCE OF Seq-interval

Seq-point ::= SEQUENCE {
    point INTEGER ,
    strand Na-strand OPTIONAL ,
    id Seq-id ,
    fuzz Int-fuzz OPTIONAL }

Packed-seqpnt ::= SEQUENCE {
    strand Na-strand OPTIONAL ,
    id Seq-id ,
    fuzz Int-fuzz OPTIONAL ,
    points SEQUENCE OF INTEGER }

Na-strand ::= ENUMERATED {                  -- strand of nucleid acid
    unknown (0), plus (1), minus (2), both (3), both-rev (4), other (255) }

Seq-bond ::= SEQUENCE {                     -- bond between residues
    a Seq-point ,                           -- connection to a least one residue
    b Seq-point OPTIONAL }                  -- other end may not be available

Seq-loc-mix ::= SEQUENCE OF Seq-loc         -- this will hold anything

Seq-loc-equiv ::= SET OF Seq-loc            -- for a set of equivalent locations

Seq-graph ::= SEQUENCE {
    title VisibleString OPTIONAL ,
    comment VisibleString OPTIONAL ,
    loc Seq-loc ,                           -- region this applies to
    title-x VisibleString OPTIONAL ,        -- title for x-axis
    title-y VisibleString OPTIONAL ,
    comp INTEGER OPTIONAL ,                 -- compression (residues/value)
    a REAL OPTIONAL ,                       -- for scaling values
    b REAL OPTIONAL ,                       -- display = (a x value) + b
    numval INTEGER ,                        -- number of values in graph
    graph CHOICE {
        real Real-graph, int Int-graph, byte Byte-graph } }

Real-graph ::= SEQUENCE {
    max REAL ,                              -- top of graph
    min REAL ,                              -- bottom of graph
```

```
   axis REAL ,                              -- value to draw axis on
   values SEQUENCE OF REAL }

Int-graph ::= SEQUENCE {
   max INTEGER ,
   min INTEGER ,
   axis INTEGER ,
   values SEQUENCE OF INTEGER }

Byte-graph ::= SEQUENCE {                        -- integer from 0-255
   max INTEGER ,
   min INTEGER ,
   axis INTEGER ,
   values OCTET STRING }

Bioseq-set ::= SEQUENCE {                        -- just a collection
   id Object-id OPTIONAL ,
   coll Dbtag OPTIONAL ,                         -- to identify a collection
   level INTEGER OPTIONAL ,                      -- nesting level
   class ENUMERATED {
      not-set (0), nuc-prot (1), segset (2), conset (3), parts (4), gibb (5), gi (6), genbank (7),
      pir (8), pub-set (9), equiv (10), swissprot (11), pdb-entry (12), mut-set (13),
      pop-set (14), phy-set (15), other (255) } DEFAULT not-set ,
   release VisibleString OPTIONAL ,
   date Date OPTIONAL ,
   descr Seq-descr OPTIONAL ,
   seq-set SEQUENCE OF Seq-entry ,
   annot SET OF Seq-annot OPTIONAL }

Seq-entry ::= CHOICE {
      seq Bioseq ,
      set Bioseq-set }

Seq-submit ::= SEQUENCE {
   sub Submit-block ,
   data CHOICE {
      entrys  SET OF Seq-entry, annots  SET OF Seq-annot, delete  SET OF Seq-id } }

Submit-block ::= SEQUENCE {
   contact Contact-info ,                        -- who to contact
   cit Cit-sub ,                                 -- citation for this submission
   hup BOOLEAN DEFAULT FALSE ,                   -- hold until publish
   reldate Date OPTIONAL ,                       -- release by date
   subtype INTEGER {                             -- type of submission
      new (1), update (2), revision (3), other (255) } OPTIONAL ,
```

```
    tool VisibleString OPTIONAL,                          -- used to make submission
    user-tag VisibleString OPTIONAL,                      -- user supplied id
    comment VisibleString OPTIONAL }                      -- user comments/advice

Contact-info ::= SEQUENCE {                               -- who to contact
    name VisibleString OPTIONAL ,
    address SEQUENCE OF VisibleString OPTIONAL ,
    phone VisibleString OPTIONAL ,
    fax VisibleString OPTIONAL ,
    email VisibleString OPTIONAL ,
    telex VisibleString OPTIONAL ,
    owner-id Object-id OPTIONAL ,                         -- for owner accounts
    password OCTET STRING OPTIONAL ,
    last-name VisibleString OPTIONAL ,                    -- structured to replace name
    first-name VisibleString OPTIONAL ,
    middle-initial VisibleString OPTIONAL ,
    contact Author OPTIONAL }
```

APPENDIX B

HAEMOPHILUS ANNOTATION FILE

```
"Most of the entities and relationships are related to the same object, so we set it up here.
 In addition, most use the same userdefined value - the selection of the sequence gi's from
the hinf.val file."


(defaultEntity
        (defaultCoercions ('Genbank Bacteria.seq_set.set.seq_set.seq' 1))
        (userDefined (
                |sess res ans|
                sess := source getSession.
                sess prepare:
                        'SELECT a.seq.inst.seq_ext.seg.int.id.gi FROM hinf.val a';
                        execute.
                ans := sess answer.
                res := List new.
                Stream endOfStreamSignal handle: [:sig| sig] do: [ |temp|
                        [true] whileTrue: [
                                temp := ans next.
                                (temp class = List) ifTrue: [res addLast: temp first].
                                ].
                        ].
                sess disconnect.
                res.
                )
)

(defaultRelationship
        (defaultCoercions ('Genbank Bacteria.seq_set.set.seq_set.seq' 1))
        (userDefined (
                |sess res ans|
                sess := source getSession.
                sess prepare:
                        'SELECT a.seq.inst.seq_ext.seg.int.id.gi FROM hinf.val a';
                        execute.
                ans := sess answer.
                res := List new.
                Stream endOfStreamSignal handle: [:sig| sig] do: [ |temp|
                        [true] whileTrue: [
                                temp := ans next.
                                (temp class = List) ifTrue: [res addLast: temp first].
                                ].
                        ].
                sess disconnect.
```

```
                res.
                )
)
```

"There are a couple of entities, and attributes, that we don't want to match any of the source attributes, since they don't have any direct matches"

```
(Aliases        (defaultCoercions (noMatchesInSource 0)))
(Alias List     (defaultCoercions (noMatchesInSource 0)))
(comment        (alias noMatchesInSource))
(source         (alias neverUsedDuringTranslation))
(quality        (alias noMatchesInSource))
(orgXchr.ind            (alias noMatchesInSource))
```

"Some of the reference attributes just need to be renamed in order to have the correct matches detected"

```
(translation            (alias ncbieaa))
(function               (alias title))
(Organisms.comment      (alias comment))
(Organisms.gaoId        (alias taxname))
(Chromosones.gaoId      (alias taxname))
(Sequences.gaoId        (alias title))
(seqXgene.start         (alias from))
(seqXgene.stop                  (alias to))
(seqXgene.direction     (alias strand))
(chrXseq.start          (alias starts))
(chrXseq.length                 (alias lens))
```

"Unfortunately, lineage is an attribute of a bioseq-set not a bioseq, so this won't match on the default. Changing the defaultCoercions for Organism will result in this attribute getting the correct variable, but not the others"

```
(classification (alias lineage))
```

"The sequence variable needs to match the ncbi4na variable in the source schema. Since this is a binary value, the type is expanded to allow the correct conversion to take place."

```
(sequence       (typeInformation (String Binary))
                (alias ncbi4na))
```

```
"These would match a source variable, so are aliased to prevent that.In addition, the
required values are used as placeholders for mapping that will occur later."

(geometry     (alias noMatchesInSource)
              (mandatoryConversion (1)))
(unnamed      (alias noMatchesInSource)
              (mandatoryConversion (1)))
(type         (alias noMatchesInSource)
              (mandatoryConversion (1)))

(garId        (mandatoryConversion (reference nextId)))

"The source gi value is used by the conversion, so the alias allows it to be located
automatically."

(gaoId        (alias gi)
              (mandatoryConversion
                      (library hinfAnnotateAliasFile: referenceDB
                                              for: curSrc_gi printString
                                              to: reference nextId
                                              idCode: 2)))

(seqXgene
              (defaultCoercions
              ('Genbank Bacteria.seq_set.set.seq_set.seq.annot.data.ftable.location'
                      1)))


"There is only one relationship that maps to the Hinf Entry directly"
(chrXseq      (defaultCoercions
                      ('Hinf Genome Entry.inst.hist.assembly.segs.denseg' 1)))

(Organisms    (defaultCoercions ('Genbank Bacteria.seq_set.set' 1)))

(Genes                (userDefined   (nil)))
```

REFERENCES

[1] Rateb Abu-Hamdeh, James Cordy, and Patrick Martin. Schema translation using structural transformation. In *Proceedings of the 1994 CAS Conference*, pages 202-215, October 1994.

[2] Sibel Adali and Ross Emery. A uniform framework for integrating knowledge in heterogeneous knowledge systems. In *Eleventh International Conference on Data Engineering*, pages 513-520, March 1995.

[3] Shailesh Agarwal, Arthur M. Keller, Gio Wiederhold, and Krishna Saraswar. Flexible relation: An approach for integrating data from multiple, possibly inconsistent databases. In *Eleventh International Conference on Data Engineering*, pages 495-504, March 1994.

[4] Rafi Ahmed, Philippe De Smedt, Weimin Du, William Kent, Mohammad A. Ketabchi, Witold A. Litwin, Abbas Rafii, and Ming-Chien Shan. The Pegasus heterogeneous multidatabase system. *Computer*, 24(12):19-26, December 1991.

[5] Jose Andany, Michel Leonard, and Carole Palisser. Management of schema evolution in databases. In *Proceedings of the 17th International Conference on Very Large Databases*, pages 161-170, September 1991.

[6] M. Andersson, Y. Dupont, S. Spaccapietra, K. Ye'tongnon, M. Tresch, and H. Ye. FEMUS: *A Federated Multilingual Database System : The Integration Process*, volume 759 of *Lecture Notes in Computer Science*, chapter 18.4, pages 371-376. Springer-Verlag, 1993.

[7] Martin Andersson, Yann Dupont, Stefano Spaccapietra, and Kokou Yetongon. FEMUS a federated multilingual database system. Technical Report adbs93, Lausanne Switzerland, 1993.

[8] Martin Andersson. Extracting an entity relationship schema from a relational database through reverse engineering. In *Conference on the Entity Approach*, December 1994.

[9] Malcom Atkinson, Francios Bancilhon, David DeWitt, Klaus Ditrich, David Maier, and Stanley Zdonik. The object-oriented database manifesto. In W. Kim, J-M Nicolas, and S.Nishio, editors, *First International Conference on Deductive and Object-Oriented Databases*, pages 223-240. Elsevier Science Pub. Co., December 1989.

[10] Tom Atwood, Joshua Duhl, Guy Ferran, Mary Loomis, and Drew Wade. *The Object Database Standard*: ODMG-93. Release 1.1

[11] Guruduth Banavar, Gary Lindstrom, and Douglas Orr. Type-safe composition of object modules. Technical Report UUCS-94-001, Department of Computer Science, University of Utah, Salt Lake City, Utah, 84112, January 1994.

[12] Jay Banerjee and Won Kim. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD Annual Conference*, pages 311-322, May 1987.

[13] C. Batini and M. Lenzerini. A methodology for data schema integration in the entity-relationship model. *Entity-Relationship Approach to Software Engineering*, pages 413-420, October 1983.

[14] Carlo Batini and Maurizio Lenzerini. A methodology for data schema integration in the entity relationship model. *Transactions on Software Engineering*,SE-10(6):650-664, November 1984.

[15] C. Batini and M. Lenzerni. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323-364, 1986.

[16] E. Bertino, M. Negri, and G. Pelagatti. Integration of heterogeneous database applications through an object-oriented interface. *Information Systems*, 14(5):407-420, 1989.

[17] Elisa Bertino. Integration of heterogeneous data repositories by using object oriented views. In *First Workshop on Interoperability in Multidatabase Systems*, pages 22-29. IEEE, April 1991.

[18] Elisa Bertino and Lorenzo Martino. Object-oriented database management systems: Concepts and issues. *Computer*, pages 33-47, 1991.

[19] Joachim Biskup and Bernhard Convent. A formal view integration method. In *Proceedings of the SIGMOD 86 International Conference on Management of Data*, pages 398-407, May 1986.

[20] Micheal Bonjour and Gilles Falquet. Concept bases: A support to information systems integration. In *Proceedings of CAiSE*94*, June 1994.

[21] Mohamed Bouneffa and Nacer Boudjlida. Managing schema changes in object-relationship databases. In *OOER'95: Object Oriented and Entity Relationship Modeling*, pages 113-122, December 1995.

[22] Mokrane Bouzeghoub and Isabelle Comyn-Wattiau. View integration by semantic unification and transformation of data structures. In H Kangassalo, editor, *Proceedings of the Ninth International Conference on the Entity- Relationship Approach*, pages 381-398. Elsevier Science Publishers B. V., October 1989.

[23] Yuri Breitbart, Peter L. Olson, and Glen R. Thompson. Database integration in a distributed heterogeneous database system. In *ICDE*, pages 301-310, 1986.

[24] M. W. Bright, A. R. Hurson, and Simin H. Pakzad. A taxonomy and current issues in multidatabase systems. *Computer*, 25(3):50-59, March 1992.

[25] M. W. Bright, A. R. Hurson, and Simin H. Pakzad. Automated resolution of semantic heterogeneity in multidatabases. *Transactions of Database Systems*, pages 212-253, June 1994.

[26] P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *Advances in Database Technology EDBT 92*, 1992.

[27] P. Buneman, S. Davidson, A. Kosky, and M. VanInwegen. A basis for interactive schema merging. In *Proceedings Hawaii International Conference on System Sciences*, 1992.

[28] M. A. Casanova and V. M. P. Vidal. Towards a sound view integration methodology. In *Proceedings of the Second Annual ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 36-47, March 1983.

[29] Malu' Castellanos and Felix Saltor. Semantic enrichment of database schemas: An object oriented approach. In *First Workshop on Interoperability in Multidatabase Systems*, pages 258-261. IEEE, April 1991.

[30] Peter Pin-Shan Chen. The entity-relationship model - toward a unified view of data. *Transactions on Database Systems*, pages 9-36, March 1976.

[31] Ferda N. Civelek, Asuman Dogac, and Stefano Spaccapietra. An expert system approach to view definition and integration. In C. Batini, editor, *The Seventh International Conference on the Entity-Relationship Approach*, pages 229-249, 1988.

[32] Christine Collet, Michael N. Huhns, and Wei-Min Shen. Resource integration using a large knowledge base in Carnot. *Computer*, 24(12):55-62, December 1991.

[33] Bogdan Czejdo and David W. Embley. An approach to schema integration and query formulation in federated database systems. In *ICDE*, pages 477-484, 1987.

[34] Bogdan Czedjo and Malcolm Taylor. Integration of database systems using an object-oriented approach. In *First Workshop on Interoperability in Multi-database Systems*, pages 30-37. IEEE, April 1991.

[35] S. B. Davidson, A. S. Kosky, and B. Eckman. Facilitating transforms in a human genome project database. Technical report, University of Pennsylvania, 1993.

[36] Umeshwar Dayal and Hai yann Hwang. View definition and generalization for database integration in a multidatabase system. *Transactions on Software Engineering*, SE-10(6):628-644, November 1984.

[37] J. M. de Souza. SIS - a schema integration system. In E. A. Oxborrow, editor, *Proceedings of the Fifth British National Conference on Databases*, pages 167-185. Cambridge University Press, July 1986.

[38] S. M. Deen, R. R. Amin, and M. C. Taylor. Data integration in distributed databases. *Transactions on Software Engineering*, SE-13(7):860-864, July 1987.

[39] Jurgen Diet and Frederick H. Lochovsky. Interactive specification and integration of user views using forms. In F. H. Lochovsky, editor, *Proceedings of the Eighth International Conference on the Entity-Relationship Approach*, pages 171-185. Elsevier Science Publishers B. V., October 1989.

[40] David M. Dilts and Wenhua Wu. Using knowledge-based technology to integrate CIM databases. *Transactions on Knowledge and Data Engineering*, 3(2):237-245, June 1991.

[41] Meeting report DOE informatics summit, April 1993. Baltimore, MD.

[42] Yann Dupont. Resolving fragmentation conflicts in schema integration. In *ER'94: Thirteenth International Conference on the Entity-Relationship Approach*, pages 513-532, December 1994.

[43] Ramez Elmasri and Sham Navathe. Object integration in logical database design. In *International Conference on Data Engineering*, pages 426-433. IEEE, Computer Society Press, April 1984.

[44] R. Elmasri, J. Weeldreyer, and A. Hevner. The category concept: An extension to the entity-relationship model. *Data and Knowledge Engineering*, pages 75-116, June 1985.

[45] Alexander Endrikat and Ralf Michalski. Application-oriented integration of distributed heterogeneous knowledge sources. In D. Karagiannis, editor, *Database and Expert Systems Applications: Proceedings of the International Conference in Berlin*, pages 327-332, 1991.

[46] P. Frankhauser and E. J. Neuhold. Knowledge based integration of heterogeneous databases. In *Proceedings of IFIP Conference DS-5 on Semantic Interoperable Database Systems*, November 1992.

[47] Karen A. Frenkel. The human genome project and informatics. *CACM*, 34:41-51, November 1991.

[48] Manuel Garcia-Solaco, Felix Saltor, and Malu Castellanos. A structure based schema integration methodology. In *Eleventh International Conference on Data Engineering*, pages 505-512, March 1995.

[49] Georges Gardarin. Integrating classes and relations to model and query geographical databases. In *DEXA'93: Fourth International Conference on Database and Expert Systems Applications*, pages 365-372, September 1993.

[50] Philip Gaudette, Steve Trus, and Sarah Collins. A free value tool for ASN.1. Technical Report NCSL/SNA-89/1, National Institute of Standards and Technology, February 1989.

[51] James Geller, Yehoshua Perl, Erich Neuhold, and Amit Sheth. Structural schema integration with full and partial correspondence using the dual model. *Information Systems*, 17(6):443-464, 1992.

[52] Willi Gotthard, Peter C. Lockemann, and Andrea Neufeld. System-guided view integration for object-oriented databases. *Transactions on Knowledge and Data Engineering*, 4(1):1-22, February 1992.

[53] J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, and D. Roland. Database evolution: The DB-MAIN approach. In *ER'94: Thirteenth International Conference on the Entity Relationship Approach*, pages 112-131, December 1994.

[54] T. A. Halpin and H. A. Proper. Database schema transformation and optimization. In *OOER'95 Fourteenth International Conference on Object-Oriented and Entity Relationship Modeling*, pages 191-203, December 1995.

[55] Stephen Hayne and Sudha Ram. Multi-user view integration system MUVIS: An expert system for view integration. In *ICDE*, pages 402-407, 1990.

[56] Paul Johannesson. A logical basis for schema integration. In *Third International Workshop on Research Issues in Data Engineering - Interoperability in Multidatabase Systems Conference*, pages 171-181, 1993.

[57] Paul Johannesson. Supporting schema integration by linguistic instruments. In *First International Workshop on Natural Languages In Databases*, pages 128-134, 1995.

[58] Burton S. Kaliski Jr. A layman's guide to a subset of ASN.1, BER, and DER, June 1991. Available via anonymous FTP from ftp.uni-erlangen.de.

[59] Kamalakar Karlapalem, Qing Li, and Chung-Dak Shum. HODFA: An architectural framework for homogenizing heterogeneous legacy databases. *SIGMOD Record*, 24(1), March 1995.

[60] M. Kaul, K. Drosten, and E.J. Neuhold. ViewSystem: Integrating heterogeneous information bases by object-oriented views. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 2-10, 1990.

[61] Daniel A. Keim, Hans-Peter Kriegel, and Andreas Miethsam. Integration of relational databases in a multidatabase system based on schema enrichment. Technical Report 9307, Ludwig-Maximilians-Universat Munchen, April 1993.

[62] Daniel A. Keim, Hans-Peter Kriegel, and Andreas Miethsam. Object-Oriented modeling of meta information for semantic schema enrichment and (semi-)automatic schema transformation. Technical Report 9306, Ludwig-Maximilians-Universat Munchen, April 1993.

[63] William Kent. Solving domain mismatch and schema mismatch problems with an object-oriented database programming language. In *Seventeenth International Conference on Very Large Data Bases*, pages 147-160, September 1991.

[64] Won Kim, Jorge F. Garza, Nathaniel Ballou, and Darrell Woelk. Architecture of the ORION next-generation database. *Transactions on Knowledge and Data Engineering*, 2(1):109-124, March 1990.

[65] Won Kim and Jungyun Seo. Classifying schematic and data heterogeneity in multidatabase systems. *Computer*, 24(12):12-18, December 1991.

[66] Anthony Kosky. A formal model for databases with applications to schema merging. In Harper and Norrie, editors, *Specifications of Database Systems*, Glasgow, 1991.

[67] Anthony Kosky. Modeling and merging database schemas. Technical report, University of Pennsylvania, September 1991.

[68] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore database system. *CACM*, 34:50-63, October 1991.

[69] James A. Larson, Shamkant B. Navathe, and Ramez Elmasri. A theory of attribute equivalence in databases with application to schema integration. *Transactions on Software Engineering*, 15(4):449-463, April 1989.

[70] Mong Li Lee and Tok Wang Ling. Resolving structural conflicts in the integration of entity relationship schemas. In *OOER'95: Object Oriented and Entity Relationship Modeling*, pages 424-433, December 1995.

[71] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. In Norman Meyrowitz, editor, *ECCOP/OOPSLA '90 Conference on Object-Oriented Programming: Systems, Languages and Applications European Conference on Object-Oriented Programming*, pages 67-76, October 1990.

[72] Ee-Peng Lim and Satya Prabhakar. Entity identification in database integration. In *Ninth International Conference on Data Engineering*. IEEE, 1993.

[73] Ee-Peng Lim, Jaideep Srivastava, and Shashi Shekhar. Resolving attribute incompatibility in database integration: An evidential reasoning approach. In *Tenth International Conference on Data Engineering*, pages 154-163. IEEE, February 1994.

[74] Chien-Tsai Liu, Panos K. Chrysanthis, and Shi-Kou Chang. Database schema evolution through the specification and maintenance of changes on entities and relationships. In *ER'94: Thirteenth International Conference on the Entity Relationship Approach*, pages 132-151, December 1994.

[75] Michael V. Mannino and Wolfgang Effelsberg. Matching techniques in global schema design. In *International Conference on Data Engineering*, pages 418-425. IEEE, Computer Society Press, April 1984.

[76] Leo Mark and Nick Roussopoulos. Metadata management. *Computer*, 19(12):26-36, December 1986.

[77] Victor M. Markowitz and Johann A. Makowsky. Incremental restructuring of relational schemas. In *Fourth Conference on Data Engineering*, pages 276-284. IEEE, February 1988.

[78] Tom Marr. Genome topographer. www.cb.cshl.org/gt/index-actual.html.

[79] Patrick Martin and Wendy Powley. Database integration using multidatabase views. In *Proceedings of CASCON93*, pages 779-788, October 1993.

[80] Patrick Martin, James R. Cordy, and Rateb Abu-Hamdeh. Information capacity preserving translations of relational schemas using structural transformations. Technical Report 95-392, Dept. of Computing and Information Science Queen's University at Kingston, November 1995.

[81] Peter McBrien and Alex Poulovassille. Formalisation of semantic schema integration. Technical Report 96-01, Dept. of Computer Science King's College London, January 1996.

[82] Dennis Mcleod. A learning based approach to meta-data evolution in an object-oriented database. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems: Second International Workshop on Object Oriented Database Systems*, volume 334 of *Lecture Notes in Computer Science*, pages 291-224. Springer-Verlag, September 1988.

[83] R. J. Miller, Y. E. Ioannidis, and R. Ramakrishnan. The use of information capacity in schema integration and translation. In *Nineteenth International Conference on Very Large Data Bases*, pages 120-133, 1993.

[84] Simon Monk and Ian Sommerville. Schema evolution in OODBs using class versioning. *SIGMOD Record*, 22(3):16-22, September 1993.

[85] Amihai Motro. Superviews: Virtual integration of multiple databases. *Transactions on Software Engineering*, SE-13(7):785-798, July 1987.

[86] Shamkant B. Navathe and Suresh G. Gadgil. A methodology for view integration in logical database integration. In *Eighth International Conference on Very Large Data Bases*, pages 142-155, 1982.

[87] S. B. Navathe, T. Sashidhar, and R. Elmasri. Relationship merging in schema integration. In *Tenth International Conference on Very Large Data Bases*, pages 78-90, August 1984.

[88] Shamkant Navathe, Ramez Elmasri, and James Larson. Integrating user views in database design. *Computer*, 19(1):50-62, January 1986.

[89] NCBI software development toolkit. Technical Report Version 1.9, National Center for Biotechnology Information, August 1994. Available via ftp from www.ncbi.nlm.nih.gov.

[90] G. T. Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data and Knowledge Engineering*, 4(1):43-67, July 1989.

[91] Maria E. Orloswska and C. A. Ewald. Schema evolution - the design and integration of fact based schemata. In B. Srinivasan and J. Zeleznikow, editors, *Research and Practical Issues in Databases: Proceedings of the Third Australian Database conference*, pages 306-320. World Scientific Publishing Company, February 1992.

[92] Sylvia L. Osborn. The role of polymorphism in schema evolution in an object-oriented database. *Transactions on Knowledge and Data Engineering*, 1(3):310-317, September 1989.

[93] *Visual Works 2.5 Object Reference*. ParcPlace Digitalk Inc., 1995.

[94] *Visual Works 2.5 User's Guide*. ParcPlace Digitalk Inc., 1995.

[95] Randal J. Peters and M. Tamer Ozsu. Axiomization of dynamic schema evolution in objectbases. In *Eleventh International Conference on Data Engineering*, pages 156-164, March 1995.

[96] Jaroslav Pokorny. Semantic relativism in conceptual modeling. In *DEXA'93 Fourth International Conference on Database and Expert Systems Applications*, pages 48-55, September 1993.

[97] Ferdi Put. Schema translation during design and integration of databases. In H. Kangassalo, editor, *Proceedings of the Ninth International Conference on the Entity-Relationship Approach*, pages 399-421. Elsevier Science Publishers B. V., October 1990.

[98] M. A. Qutaishat, N. J. Fiddian, and W. A. Gray. Association merging in a schema meta-integration system for a heterogeneous object-oriented database environment. In P. M. D. Gray and R. J. Lukas, editors, *Advanced Database Systems: Tenth British National Conference on Databases*, pages 209-226. Springer-Verlag, July 1992.

[99] Young-Gook Ra and Elke A. Rudensteiner. A transparent object-oriented schema change approach using view evolution. In *Eleventh International Conference on Data Engineering*, pages 165-172, March 1995.

[100] M. P. Reddy, B. E. Prasad, P. G. Reddy, and Amar Gupta. A methodology for integration of heterogeneous databases. *Transactions on Knowledge and Data Engineering*, 6(6):920-933, December 1994.

[101] John F. Roddick. Schema evolution in database systems - an annotated bibliography. *SIGMOD Record*, 21(4):35-40, December 1992.

[102] Arnon Rosenthal and David Reiner. Tools and transformations - rigorous and otherwise - for practical database design. *Transactions on Database Systems*, 19(2), June 1994.

[103] Marek Rusinkiewicz, Amit Sheth, and George Karabatis. Specifying inter-database dependencies in a multidatabase environment. *Computer*, 24(12):46-53, December 1991.

[104] Rob Sargent, Dave Fuhrman, Terence Critchlow, Tony Di Sera, Robert Mecklenburg, Gary Lindstrom, and Peter Cartwright. The design and implementation of a database for human genome research. In *The Eighth International Conference on Scientific and Statistical Database Management*. IEEE Computer Society Press, June 1996.

[105] Ashoka Savasere, Amit Sheth, Sunit Gala, Shamkant Navathe, and Howard Marcus. On applying classification to schema integration. In *First Workshop on Interoperability in Multidatabase Systems*, pages 258-261. IEEE, April 1991.

[106] Ingo Schmitt. Flexible integration and derivation of heterogeneous schemata in federated database systems. Technical Report Preprint Nr. 10, Fakultat fur Informatik, Universitat Magdeburg, Magdeburg Germany, November 1995.

[107] Ingo Schmitt and Gunter Saake. Schema integration and view derivation by resolving intensional and extensional overlappings. In *9th ICSA International Conference on Parallel and Distributed Computing Systems*, September 1996.

[108] Micheal Schrefl and Erich J. Neuhold. A knowledge based approach to overcome structural differences in object oriented database integration. In Robert A. Meersman, Zhongzhi Shi, and Chen-Ho Kung, editors, *Proceedings of the IFIP Working Conference on the Role of Artificial Intelligence in Database and Information Systems*, pages 265-304. Elsevier Science Publishers B. V., July 1988.

[109] Peter Schwarz and Kurt Shoens. Managing change in the Rufus system. In *Tenth International Conference on Data Engineering*, pages 170-179. IEEE, February 1994.

[110] Edward Sciore, Michael Siegel, and Amon Rosenthal. Using semantic values to facilitate interoperability among heterogeneous information systems. *Transactions of Database Systems*, pages 254-290, June 1994.

[111] Len Seligman and Arnon Rosenthal. A metadata resource to promote data integration. In *Proceedings of IEEE Metadata Conference*, April 1996.

[112] Amit P. Sheth and James A. Larson. A tool for integrating conceptual schemas and user views. In *International Conference on Data Engineering*, pages 176-183. IEEE, 1988.

[113] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed heterogeneous and autonomous databases. *ACM Computing Surveys*, 22(3):183-236, September 1990.

[114] Amit P. Sheth, Sunit K. Gala, and Shamkant B. Navathe. On automatic reasoning for schema integration. *International Journal of Intelligent Cooperative Information Systems*, 2(1):23-50, 1993.

[115] Peretz Shoval and Sara Zohn. Binary-relationship integration methodology. *Data and Knowledge Engineering*, 6(3):225-250, May 1991.

[116] Micheal Siegel and Stuart E. Madnick. A metadata approach to resolving semantic conflicts. In *Seventeenth International Conference on Very Large Data Bases*, pages 133-145, September 1991.

[117] William W. Song, Paul Johannesson, and Janis A. Bubenko Jr. Semantic similarity relations in schema integration. In G. Pernul and A. M. Tjoa, editors, *Eleventh International Conference on the Entity-Relationship Approach*, volume 645 of *Lecture Notes in Computer Science*, pages 97-120. Springer-Verlag, October 1992.

[118] Stefano Spaccapietra and Christine Parent. View integration: a step forward in solving structural conflicts. Technical Report tdke93, Institute of Technology in Lausanne, Lausanne Switzerland, 1993. To appear in IEEE TKDE 1993.

[119] Stefano Spaccapietra, Christine Parent, and Yann Dupont. Model independent assertions for integration of heterogeneous schemas. Technical Report VLDBJournal92, Institute of Technology in Lausanne, Lausanne Switzerland, 1993.

[120] Frederick N. Springsteel. Object-based schema integration for heterogeneous database: A logical approach. In *DEXA'93: Fourth International Conference on Database and Expert Systems Applications*, pages 166-180, September 1993.

[121] Telecommunication Standardization Sector of International Telecommunication Union. *Specification of Abstract Syntax Notation One (ASN.1) ITU-T Recommendation X.208*, 1988. Extracted from the Blue Book.

[122] Telecommunication Standardization Sector of International Telecommunication Union. *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1) ITU-T Recommendation X.209*, 1988. Extracted from the Blue Book.

[123] Christiaan Thieme and Arno Siebes. An approach to schema integration based on transformations and behavior. Technical Report CS-R9403, CWI, 1994.

[124] Markus Tresch and Marc H. Scholl. Meta object management and its application to database evolution. In G. Pernul and A. M. Tjoa, editors, *Eleventh International Conference on the Entity-Relationship Approach*, volume 645 of *Lecture Notes in Computer Science*, pages 299-321. Springer-Verlag, October 1992.

[125] Markus Tresch and Marc H. Scholl. Schema transformation without database reorganization. *SIGMOD Record*, 22(1):21-27, March 1993.

[126] Susan D. Urban and Jian Wu. Resolving semantic heterogeneity through the explicit representation of data model semantics. *SIGMOD Record*, 20(4):55-58, December 1991.

[127] Vania M.P. Vidal and Marianne Winslett. A rigorous approach to schema restructuring. In *OOER'95: Fourteenth International Conference on Object-Oriented and Entity-Relationship Modeling*, pages 101-112, December 1995.

[128] Y. Richard Wang and Stuart E. Madnick. The inter-database instance identification problem in integrating autonomous systems. In *Fifth International Conference on Data Engineering*, pages 46-55. IEEE, 1989.

[129] W.K. Whang, S.B. Navathe, and S. Chakravarthy. Logic based approach for realizing a federated information system. In *First Workshop on Interoperability in Multidatabase Systems*, pages 258-261. IEEE, April 1991.

[130] Whan-Kyu Whang, Sharma Chakravarthy, and Shankant B. Navathe. Heterogeneous databases: Inferring relationships for merging component schemas, and a query language. Technical Report 048, University of Florida, December 1992.

[131] Jian Yang, Mike P. Papazoglou, and Louis Marinos. Knowledge-based schema analysis in a multi-database framework. In D. Karagiannis, editor, *Database and Expert Systems Applications: Proceedings of the International Conference in Berlin*, pages 315-320, 1991.

[132] S. Bing Yao, V. E. Waddle, and Barron C. Housel. View modeling and integration using the functional data model. *Transactions on Software Engineering*, SZE-8(6):544-553, November 1982.

[133] Roberto Zicari. A framework for schema updates in an object-oriented database system. In *Seventh International Conference On Data Engineering*, pages 2-13. IEEE, April 1991.