Semantics of Database Transformations*

Susan Davidson[1], Peter Buneman[1], and Anthony Kosky[2]

[1] Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104
[2] Lawrence Berkeley National Laboratory, Berkeley, CA 94705.

Abstract. Database transformations arise in many different settings including database integration, evolution of database systems, and implementing user views and data-entry tools. This paper surveys approaches that have been taken to problems in these settings, assesses their strengths and weaknesses, and develops requirements on a formal model for specifying and implementing database transformations.

We also consider the problem of insuring the correctness of database transformations. In particular, we demonstrate that the usefulness of correctness conditions such as information preservation is hindered by the interactions of transformations and database constraints, and the limited expressive power of established database constraint languages. We conclude that more general notions of correctness are required, and that there is a need for a uniform formalism for expressing both database transformations and constraints, and reasoning about their interactions.

Finally we introduce *WOL*, a declarative language for specifying and implementing database transformations and constraints. We briefly describe the *WOL* language and its semantics, and argue that it addresses many of the requirements on a formalism for dealing with general database transformations.

1 Introduction

The need to implement transformations between distinct, heterogeneous databases has become a major factor in information management in recent years. Problems of reimplementing legacy systems, adapting application programs and user interfaces to schema evolution, integrating heterogeneous databases, and merging user views or mapping between data-entry screens and the underlying database all involve some form of transformation. The wide variety of data models in use, including those supporting complex data structures and object-identities, further complicate these problems.

A *database transformation* is a set of mappings from the instances of one or more *source* database schemas to the instances of some *target* schema. The schemas involved may be expressed in a variety of different data-models, and

* This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF BIR94-02292 PRIME, ARO AASERT DAAH04-93-G0129, ARPA N00014-94-1-1086 and DOE DE-AC03-76SF00098.

implemented using different DBMSs. Incompatibilities between the sources and target exist at all levels – the choice of data-model, the representation of data within a model, the data of an instance – and must be explicitly resolved within the transformation.

Much of the existing work on transformations concentrates on the restructuring of source database schemas into a target schema, either by means of a series of simple manipulations or by a description in some abstract language, and the mappings of the underlying instances are determined by the restructurings of schemas. In some cases this emphasis is at the expense of a formal treatment of the effect of transformations on instances, which is stated informally or left to the intuition. However there are, in general, many possible interpretations of a particular schema manipulation. For example, in a data model supporting classes of objects and optional attributes of classes, suppose we changed an attribute of an existing class from being optional to being required. There are a number of ways that such a schema manipulation can be reflected on the underlying data: we could insert a default value for the attribute where ever it is omitted, or we could simply delete any objects from the class for which the attribute is missing.

It is clear that there may be many transformations, with differing semantics, corresponding to the same schema manipulation, and that it is necessary to be able to distinguish between them. In contrast to existing work, our focus in this paper is therefore on how transformations effect the underlying data itself. We will use the term "*database transformations*", as opposed to the more common "schema transformations", in order to emphasize this distinction.

Implementations of database transformations fall into two camps: those in which the data is actually transformed into a format compatible with the target schema and then stored in a target database, and those in which the data remains stored in the source databases and queries against the target schema are translated into queries against the source databases. The first of these approaches can be thought of as performing a *one-time bulk transformation*, while the second approach evaluates transformations in a *call-by-need* manner.

For example, the most common approach adopted within federated database systems [14] is call-by-need [25,10,22]. This approach has the advantage that the source databases retain their autonomy, and updates to the various source databases are automatically reflected in the target database. However, in cases where accessing the component databases is costly and the databases are not frequently updated, actually merging the data into a local unified database may be more efficient. Furthermore, maintaining integrity constraints over a federated database system is a much more difficult task than checking data integrity for a single merged database [30,39]. As a result, the approach of performing a one-time bulk transformation is taken in [38].

Some work on schema evolution also advocates implementing transformations in a call-by-need manner [5,33,35]. In this case multiple versions of a schema are maintained, and data is stored using the version for which it was originally entered. The advantage of the approach is that major database reorganizations can be avoided, and applications implemented for an earlier version of a schema can still be used. However, for applications built on old versions of a schema to be applied to new data, reverse transformations must also be implemented. Furthermore the cost of maintaining multiple views and computing compounded transformations may be prohibitively expensive. These problems are especially significant when schema evolutions are frequent, and it is not possible a priori to tell when old views or data cease to be relevant. Consequently some practical work on implementing schema evolutions has been based on performing bulk transformations of data [27].

It is clear that the implementation method appropriate for a particular transformation will depend on the application and on the databases involved. However, the semantics of a transformation should be independent of the implementation method chosen as well as of the application area itself. Unfortunately, for much of the work in the area of database transformations this is not the case, primarily due to the fact that there is no independent semantics for the transformation. A focus of this paper is therefore to develop a semantics of database transformations, and examine various metrics for the "goodness" or "correctness" of such transformations.

We start in the next section by giving an informal example of a database transformation as it might arise within heterogeneous database integration, and surveying approaches that have been taken within this domain. The example is used to illustrate that while many of the ideas in these approaches are useful, none of them capture all necessary structural manipulations and that there is therefore a need for a more general and flexible formalism for expressing such transformations.

Section 3 formalizes the example by presenting a data model which gives a precise semantics to a database schema, instances and keys. The model is used in section 4 to examine the notion of information capacity preserving transformations. We argue that while this is an appealing "correctness" metric for database transformations, it is not always useful because it does not capture intuitively meaningful transformations, and fails to take into account implicit constraints on the databases being transformed. We conclude that there is a need to express and test more general correctness conditions, and to derive constraints on the databases being transformed from such conditions.

Section 5 presents a declarative language for expressing database transformations and constraints called *WOL* (Well-founded Object Language). We show that *WOL* is not only sufficient for expressing the transformations occuring in existing work, but that it is more expressive than existing transformation languages for the data-models being considered. *WOL* can also be used to

express the constraints on individual databases and *between* databases necessary to ensure correctness of transformations, hence filling a significant need in the field of database transformations.

2 Transformations in Database Integration

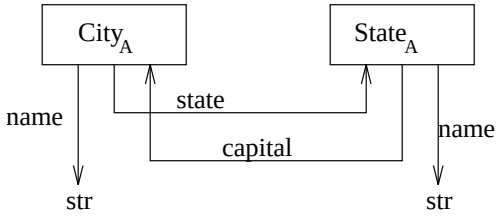
In this section we will look at some examples of database transformations, particularly in the context of database integration, and show how some parts of these examples are addressed by existing work, while others require more general transformation techniques. The context of database integration is particularly appropriate since much of the most significant work in database transformations stems from this field. In contrast, transformations proposed in say the area of schema evolution are comparatively simple [28,35,27,5,33], normally being based on a single model and a small set of basic schema modifications, such as introducing specialization and generalization classes, adding or removing attributes, and so on. It is not clear whether the reason for this is historical, since database integration became a significant problem earlier in terms of the need for formal tools and techniques, or because the transformations involved in database integration are inherently more difficult than those arising in other areas.

2.1 Database Integration: An Example

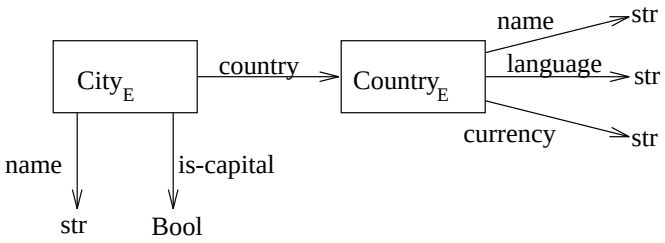
The objective of database integration is to make data distributed over a number of distinct, heterogeneous databases accessible via a single database interface, either by constructing a (virtual) view of the component databases to give them the appearance of a single database, or by actually mapping data from the component databases into a single unified database. In either case, the problem from the perspective of database transformations is how to transform data from the various formats and structures in which it is represented in the component databases into a form compatible with the integrated database schema.

Example 1. Figure 1 shows the schemas of two databases representing US Cities and States, and European Cities and Countries respectively. The graphical notation used here is inspired by [3]: the boxes represent *classes* which are finite sets of objects; the arrows represent *attributes*, or functions on classes; and *str* and *Bool* represent sets of *base values*. An instance of such a schema consists of an assignment of finite sets of objects to each class, and of functions on these sets to each attribute. The details of this model will be made precise in section 3.

The first schema has two classes: *City* and *State*. The City class has two attributes: *name*, representing the name of a city, and *state*, which points to



Schema of US Cities and States



Schema of European Cities and Countries

Fig. 1. Schemas for US Cities and European Cities databases

the state to which a city belongs. The *State* class also has two attributes, representing its name and its capital city.

The second schema also has two classes, this time *City* and *Country*. The *City* class has attributes representing its name and its country, but in addition has a Boolean-valued attribute *capital* which represents whether or not it is the capital city of a country. The *Country* class has attributes representing its name, currency and the language spoken.

Suppose we wanted to combine these two databases into a single database containing information about both US and European cities. A suitable schema is shown in figure 2, where the "plus" node indicates a variant. Here the *City* classes from both the source databases are mapped to a single class *City* in the target database. The *state* and *country* attributes of the *City* classes are mapped to a single attribute *place* which take a value that is either a *State* or a *Country*, depending on whether the *City* is a US or a European city. A more difficult mapping is between the representations of capital cities of European countries. Instead of representing whether a city is a capital or not by means of a Boolean attribute, the *Country* class in our target database has an attribute *capital* which points to the capital city of a country. To resolve this difference in representation a straightforward embedding of data will not be sufficient; we will need to do some more sophisticated structural transfor-

mations on the data. Further constraints on the source database, ensuring that each *Country* has exactly one *City* for which the *is_capital* attribute is true, are necessary in order for the transformation to be well defined. (The interaction between constraints and transformations will be explored in section 4.2).

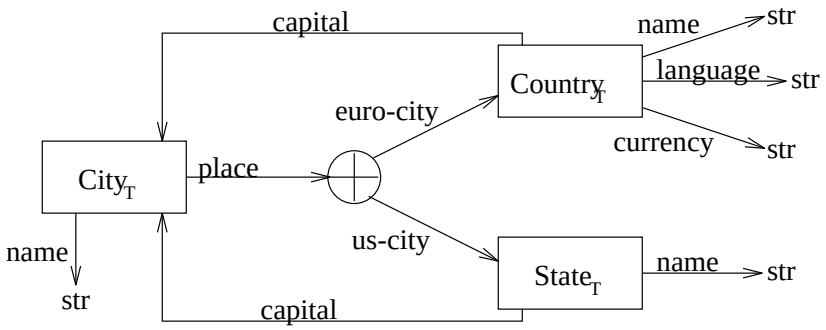


Fig. 2. An integrated schema of European and US Cities

The problem of database integration may therefore be seen as forming an *integrated schema*, which represents the relevant information in the component source databases, together with transformations from the source databases to this integrated schema.

2.2 Resolving Structural Conflicts in Database Integration

In [7] Batini et al. noted that schema integration techniques generally have two phases: conflict resolution and merging or unioning of schemas. Although schema merging has received a great deal of attention, it is only a small (and usually the last) step in the process of database integration. The more significant part of the process is manipulating the component databases so that they represent data in a compatible way. In order to do this it is necessary to resolve naming conflicts between the schemas (both homonyms and synonyms), and also to perform structural manipulations on data to resolve conflicts in the way data is represented. The structural manipulations required are usually computationally simple, and do not normally involve any unbounded computation (iteration or recursion). An example of such a structural manipulation was given by how the capital attribute was represented in the European Cities schema.

The order in which the conflict resolution and schema merging phases are carried out varies between different database integration methods. For example, in Motro[25] component schemas are first unioned to form disjoint

components of a "superview" schema, and the superview is then manipulated in order to combine concepts and resolve conflicts between the component schemas. In contrast, [26,8,32] assume that conflicts between schemas are resolved prior to the schema merging process, and [6] interleaves the two parts of this process.

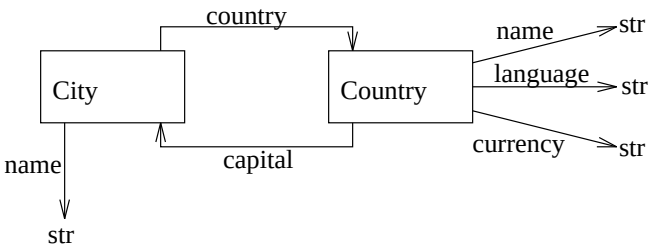


Fig. 3. A modified schema for a European cities and countries database

Example 2. Returning to example 1, it is necessary to perform a structural modification on the database of European Cities and Countries to replace the Boolean *is_capital* attribute of the *City* class with a *capital* attribute of class *Country* going to the class *City*. This yields an intermediate database with the schema shown in figure 3. It is then necessary to associate the classes and attributes of the two source databases, so that the *City* classes and *name* attributes, and also the *state* and *country* attributes, are associated, and the remainder of the transformation could be implemented by means of an automated schema-merging tool.

There are two basic approaches to systems for implementing transformations to resolve such structural conflicts: using a small set of simple transformations or heuristics that can be applied in series [25,5,24,32], or using some high-level language to describe the transformation [1,10]. Examples of such approaches will be given in section 2.3.

The advantage of using a small set of pre-defined atomic transformations is that they are simple to reason about and prove correctness for. For instance, one could prove that each transformation was information preserving [29,24], or if necessary associate constraints with each transformation in order for it to be information capacity preserving, and deduce that a series of applications of the transformations was information preserving. The disadvantage of this approach is that the expressivity of such a family of transformations is inherently limited. For example the family of transformations proposed in [25] are

insufficient to describe the transformation between an attribute of a class and a binary relation between classes: that is, one cannot transform from a class *Person* with an attribute *spouse* of class *Person* to a binary relation *Marriage* on the *Person* class. Although it might be easy to extend the family of atomic transformations to allow this case, which is a common source of incompatibility between databases, there would still be other important transformations that could not be expressed. The restructuring described in example 2 also can not be expressed using any of the families of transformations mentioned above.

A potentially much more flexible approach is to use some high-level language for expressing structural transformations on data. However programming and checking a transformation in such a language is a more laborious task. Further, if it is necessary to ensure that a transformation is information preserving then additional constraints may be needed on the source databases, and in general these constraints will not be expressible in any standard constraint language. This will be taken up again in more detail in section 4. We therefore believe that there is a need for a declarative language for expressing such transformations and constraints, which allows one to formally reason about the interaction between transformations and constraints and which is sufficiently simple to allow transformations to be programmed easily. Such a language will be presented in section 5.

2.3 Schema Integration Techniques

In [7] Batini et al. survey existing work on *schema integration*. They observe that schema integration arises from two tasks: *database integration*, which we have already discussed, and *integration of user views*, which occurs during the design phase of a database when constructing a schema that satisfies the individual needs of each of a set of user groups. However they fail to note that these two kinds of schema integration are fundamentally different. The reason for this can be seen by considering the direction in which data is transformed in each case. For database integration, instances of each of the source databases are transformed into instances of the merged schema. On the other hand, when integrating multiple user views instances of the merged schema must be transformed back into instances of the user views (see figure 4). The intuition is that when integrating user views *all* of the underlying information must be represented; no objects or attributes can be missing since some user may want the information. However, when integrating pre-existing databases the best that can be hoped for is that attributes of objects that are present in every underlying database will definitely be present in the integration; attributes that are present in some but not all of the underlying databases may be absent in the integration. In [8] it was observed that integrating user views corresponds to the "least upper bound" of the component schemas in some information ordering on schemas, while in database integration what

is required is the "greatest lower bound" of the component schemas in some information ordering on schemas. A good schema-integration method should therefore take account of its intended purpose and include a semantics for the underlying transformations of instances.

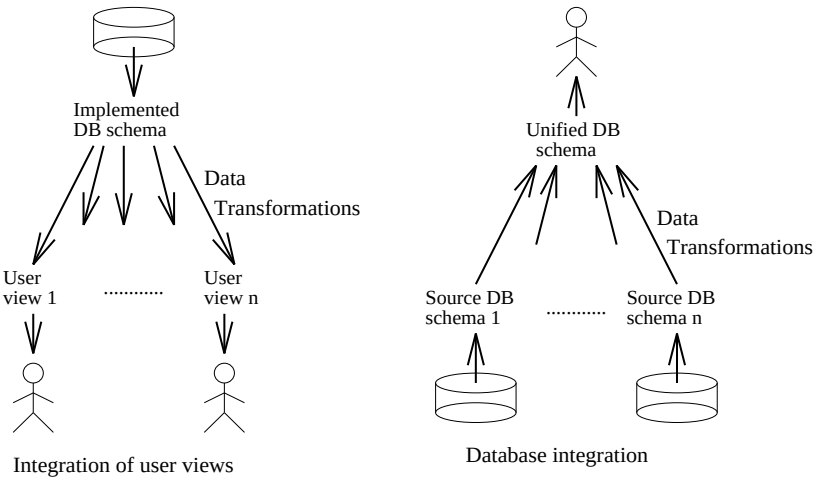


Fig. 4. Data transformations in applications of schema integration

In this section we will concentrate on methodologies intended for database-integration, and look at some representative examples of the various approaches to this problem.

Example 3. Continuing with our example of database integration, we can use the technique of Motro[25] to integrate the *Cities* and *States* database of figure 1 with the restructured *Cities* and *Countries* database of figure 3.[1] The process is illustrated in figure 5. First, a disjoint union of the two schemas is formed (a), and then a series of "macro" transformations are applied to form the desired integrated schema.[2] The transformations applied include introducing generalizations (b), deriving new attributes as compositions or combinations of existing attributes (c), and combining classes (d).

In this particular integration method, the semantics of the transformations are strongly linked to the implementation method. The intention is that the integrated database be implemented as a *view* of the component databases,

[1] Recall that this methodology is not expressive enough to express the transformation from the *Cities* and *Countries* database of figure 1 to that of figure 3.

[2] In the model of [25] generalizations are represented by classes with *isa* edges, though for consistency we present this example using variants instead.

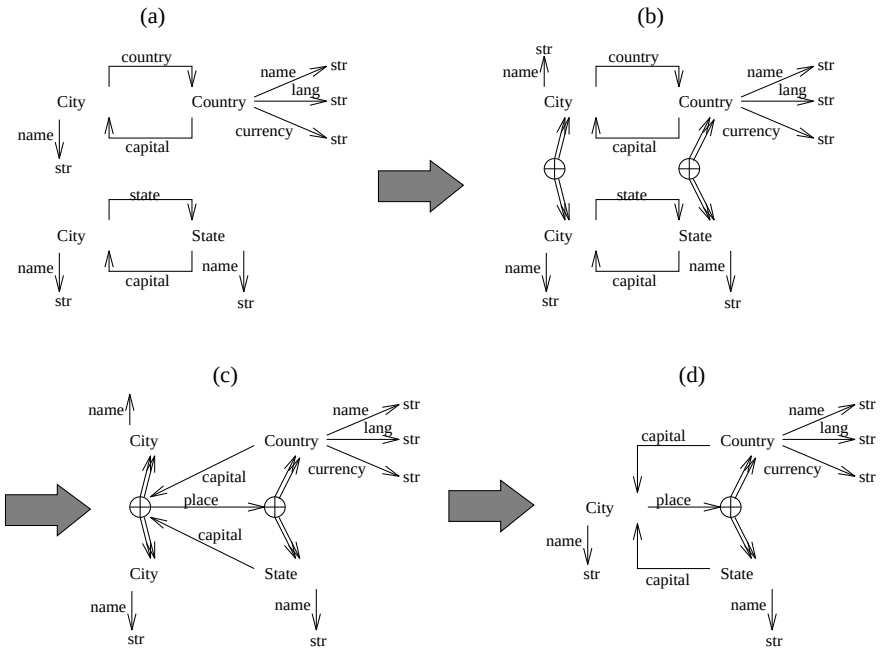


Fig. 5. A schema integration using the methodology of Motro

and that queries against the integrated database be executed by translating them into queries against the component databases and then combining the results. The semantics of the individual transformations are given by their effects on queries. However the lack of any independent characterization of their semantics makes it difficult to reason about or prove properties of the transformations, or to use any alternative implementation of the methodology.

A more expressive and flexible way of specifying transformations is to use some sort of high-level transformation language. An example of such an approach is the system of rewrite rules for nested relational structures proposed by Abiteboul and Hull in [1]. The model of [1] is purely *value-based*: there is no concept of object identity. Consequently it is necessary to use some notion of *keys* in order to represent recursive structures such as those of figure 1, and to reference values in one table or class from values in another (see section 3)

A number of other approaches to schema merging [26,8,32] take component schemas – such as those of figures 1 and 3 – together with constraints relating the elements of the schemas – for example saying that the *City* classes of the two schemas and the *state* and *country* attributes correspond – and apply an algorithm which returns a unified schema. In these approaches the transformations are generally simple embeddings of data and type coercions.

For most schema integration methodologies the outcome is dependent on the order in which schemas are integrated: that is, they are not associative. Intuitively this should not be the case, since the integration of a set of schemas should depend only on the schemas and the relations between them; the semantics of the integration should be independent of the algorithm used. As a consequence of this non-associativity, a schema integration method will specify an ordering in which schema integrations take place, such as a binary tree or ladder, or all at once, and possibly a way of ordering the particular schemas. For example [6] states that schemas should be ranked and then integrated in order of relevance, although no justification for this ordering is given: why shouldn't it be appropriate to integrate the most relevant schemas last, or in the middle, rather than first? Further enforcing such an ordering is not acceptable in a system in which new databases may be added to the system at a later date: if a database is added to an established federation the result should be the same as if the database had been present in the federation at the outset.

In [8] it was shown that the non-associativity of schema integration methodologies is due to new "implicit" nodes or classes that are introduced during the merging process. The variant of the *State* and *Country* classes in example 1 is an example of such an implicit node. By taking account of these implicit nodes and how they are introduced, an independent semantics can be given to the merge of a set of schemas and the relations between them, and an associative schema merging algorithm defined [8].

2.4 Merging Data

Once transformed into a suitable form, data from the component databases must be merged. In a value based model without additional constraints, this is simply a matter of taking the union of the relevant data. However, when more complex data models are used, such as those supporting object identity or inter-database constraints, this task becomes more difficult since it necessary to resolve conflicts and equate objects arising from different databases [34,17].

This problem is not apparent in our running example because the databases of *Cities* and *States* and *Cities* and *Countries* represent disjoint sets of objects. However suppose we were also interested in integrating a third database including international information about Cities and Countries with the schema shown in figure 6. This schema has three classes: *City*, *Country* and *Region*. Each *City* is in a *Region*, and each *Country* has a set (indicated by a "star" node) of *Regions*. The exact meaning of *Region* depends on the country to which it belongs. For example, in the United States, Regions would correspond to States (or Districts), while in Great Britain Regions might be counties. This database might contain data which overlaps with the other two databases. For example there might be objects representing the city

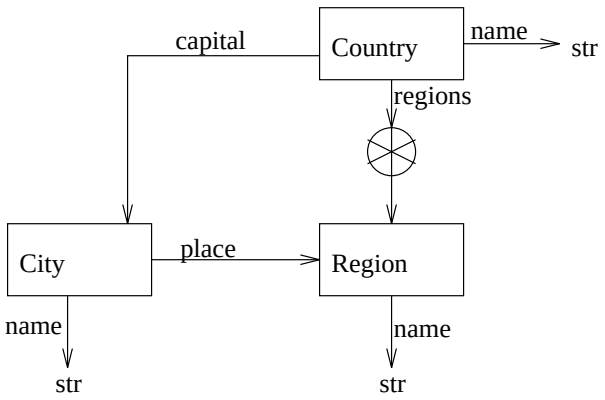


Fig. 6. A schema for a international database of Cities and Countries

Philadelphia in both the International Cities and Countries database and in the Cities and States database, in which case it would be necessary to map both objects to the same object in the integrated database. Equally there might be objects representing the same City or Country in both the International and the European Cities databases, which would need to be combined in the target database.

An important point to note here is that transformations from the various source databases to an integrated database are not independent: it is not sufficient to merely write a transformation from each individual database to the target database. Instead, we must write a transformation that takes a *set* of database instances, one for each source schema, and transforms them into a target instance.

The problem of resolving object identity over multiple databases with constraints is examined in [17,12,37]. [19] gives an analysis of the more general problem of how to compare and equate object identities, and concludes by recommending a system of *external keys* for identifying object identities.

3 Data Models for Database Transformations

Works on transformations between heterogeneous databases are usually based around some sufficiently expressive data-model, or *meta*-data-model, which naturally subsumes the models used for the component databases. Various data models have been used, ranging from relational and extended entity-relationship models to semantic and object-oriented models. The main requirements on such a meta-data-model are that the models of component databases being considered should be embeddable in it in a natural way, and that it be sufficiently simple and expressive to allow data to be represented in

multiple ways, so that conflicts between alternative representations of data can be resolved. In [31] the requirements on a model for transforming heterogeneous databases are examined, and the authors conclude that a model supporting complex data-structures (sets, records and variants), object-identity and specialization and generalization relations between object classes is desirable.

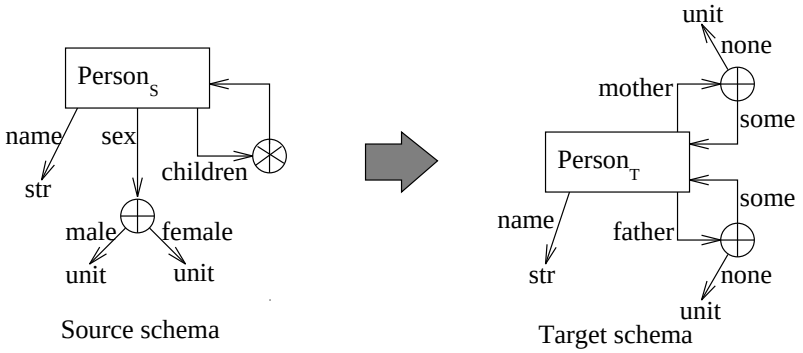


Fig. 7. A transformation between recursive data structures

Some notion of referencing, such as *object-identities* or *keys* is essential in order to represent recursive data-structures such as those of figures 1 and 2. However, in order to transform databases involving such recursive structures, it is also necessary to have a notion of *extents* or *classes* in which all objects of a database must occur. To see this, let us look at another example, namely the transformation between the two schemas shown in figure 7. Suppose we considered the first schema merely to define a recursive type $Persons_S$. A value of type $Persons_S$ would be a record with attributes *name*, *sex* and *children*, such that the *children* attribute would be a set of records of type $Persons_S$. In order to transform a source database consisting of a set of values of type $Persons_S$, we would have to recursively apply a restructuring transformation to each set of children of each person in the database. This recursion could be arbitrarily deeply nested, and, in the case of cyclic data, non-terminating.

Fortunately the source schema of figure 7 conveys some important information in addition to describing a recursive type: namely it tells us that our database consists of a *finite extent* or *class* $Persons_S$, and that all the people represented in the database are reachable as members of this extent. In particular it tells us that, if X is an object in the class $Persons_S$ and $Y \in X.children$ (Y is a child of X), then Y is also in the class $Persons_S$. Consequently, when transforming the database, we can iterate our transformation over the elements of the class $Persons_S$, and do not have to worry about recursively applying the transformation to the children of a person.

Note that in performing a transformation, it may be necessary to create and reference an object-identity before it has a value associated with it. In this example, if we perform the transformation by iterating over the class $Person_S$, it may be necessary to create an object in the target class $Person_T$, with *father* and *mother* attributes both set to *some* person, before the objects corresponding to the parents of the person being transformed have been encountered in the class $Person_S$. In this case it is necessary to create and reference object identities for the two parents, even though the corresponding values have not yet been formed. Keys provide a mechanism for such early creation and referencing of object identities.

Keeping these requirements in mind, we now present the data-model for *WOL* so that it can be used for examples throughout the remainder of this paper. It is presented in three stages: First we present schemas, then instances, and then keyed-schemas and instances.

3.1 The *WOL* Data-Model

The data-model for *WOL* supports object-identities, classes and complex data-structures. However we prefer to view specialization and generalization relations as particular examples of constraints which can be expressed separately using a general constraint language. The model is basically the same as that of [2] and is equivalent to the models implemented in various object-oriented databases [4], except for the omission of direct support for inheritance. It is also necessary to have some mechanism to create and reference object-identities. Since object identities themselves are generally considered to be abstract values, which are not directly visible, some value-based handle on them is desirable. We follow [18] in using surrogate keys for this purpose.

We assume a fixed set of *base types*, $\mathcal{B}$, ranged over by $\underline{b}, \underline{b}', \ldots$, and a countable set of *attribute labels*, $\mathcal{A}$, ranged over by $a, a', \ldots$, together with some fixed arbitrary ordering on $\mathcal{A}$. Our definition of types will be relative to a particular finite set of *classes*.

Assume a finite set $\mathcal{C}$ of *classes* ranged over by $C, C', \ldots$. The **types** over $\mathcal{C}$, ranged over by $\tau, \tau', \ldots$, consist of *base types*, $\underline{b}$, *class types* C, where $C \in \mathcal{C}$, *set types* $\{\underline{b}\}$ and $\{C\}$ for each base type $\underline{b}$ and class type C, *record types* $(a_1 : \tau_1, \ldots, a_k : \tau_k)$, where $a_1, \ldots, a_k$ are arranged according to the ordering on $\mathcal{A}$, and *variant types* $\langle\!|a_1 : \tau_1, \ldots, a_k : \tau_k|\!\rangle$. We write $Types^{\mathcal{C}}$ for the set of types over $\mathcal{C}$. The restriction on set types, that they be either sets of base or class type, can be relaxed and replaced by some more general but complicated constraints on set types and on expressions dealing with sets, as in [20]. A restriction of this nature is necessary in order to be able to navigate and identify elements in nested sets. We need to avoid types such as sets of sets $(\{\{\tau\}\})$, particularly in the target database of a transformation, where

sets may only be partially instantiated as the transformation progresses, and therefore cannot be compared or equated.

A **schema**, $\mathcal{S}$, consists of a finite set of classes, $\mathcal{C}$, and for each class $C \in \mathcal{C}$ a corresponding type $\tau^C \in \mathit{Types}^{\mathcal{C}}$ where τ^C is not a class type.

For each base type $\underline{b}$ we assume a countable set $\mathbf{D}^{\underline{b}}$ corresponding to the *domain* of $\underline{b}$. Suppose we have a schema $\mathcal{S}$ with classes $\mathcal{C}$, and for each $C \in \mathcal{C}$ we have a disjoint set σ^C of object identities of class C. The set of values associated with a particular type are dependent on the object identities present in an instance. For each type $\tau \in \mathit{Types}^{\mathcal{C}}$ we define a set $[\![\tau]\!]\sigma^{\mathcal{C}}$ as in figure 8.

$$
\begin{aligned}
[\![\underline{b}]\!]\sigma^C &\equiv \mathbf{D}^{\underline{b}} \\
[\![C]\!]\sigma^C &\equiv \sigma^C \\
[\![\{\underline{b}\}]\!]\sigma^C &\equiv \mathcal{P}_{fin}(\mathbf{D}^{\underline{b}}) \\
[\![\{C\}]\!]\sigma^C &\equiv \mathcal{P}_{fin}(\sigma^C) \\
[\![(a_1 : \tau_1 \ldots, a_k : \tau_k)]\!]\sigma^C &\equiv [\![\tau_1]\!]\sigma^C \times \ldots \times [\![\tau_k]\!]\sigma^C \\
[\![\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle]\!]\sigma^C &\equiv (\{a_1\} \times [\![\tau_1]\!]\sigma^C) \cup \ldots \cup (\{a_k\} \times [\![\tau_k]\!]\sigma^C)
\end{aligned}
$$

Fig. 8. The semantic operator on types

An **instance**, $\mathcal{I}$, of a database schema $\mathcal{S}$ consists of a family of sets of object identities, $\sigma^{\mathcal{C}}$, and for each class $C \in \mathcal{C}$, a mapping $\mathcal{V}^C : \sigma^C \rightarrow [\![\tau^C]\!]\sigma^{\mathcal{C}}$. We write $\mathit{Inst}(\mathcal{S})$ for the set of instances of schema $\mathcal{S}$.

Example 4. The first schema illustrated in example 1 has two classes representing *Cities* and *States*, with each city having a *name* and a *state*, and each state having a *name* and a *capital city*. The set of classes for the schema is therefore $\mathcal{C}_A \equiv \{\mathit{City}_A, \mathit{State}_A\}$ and the associated types are

$$
\begin{aligned}
\tau_A^{\mathit{City}_A} &\equiv (\mathit{name} : \mathit{str},\ \mathit{state} : \mathit{State}_A) \\
\tau_A^{\mathit{State}_A} &\equiv (\mathit{name} : \mathit{str},\ \mathit{capital} : \mathit{City}_A)
\end{aligned}
$$

The second schema has classes $\mathcal{C}_E \equiv \{\mathit{City}_E, \mathit{Country}_E\}$ and associated types

$$
\begin{aligned}
\tau_E^{\mathit{City}_E} &\equiv (\mathit{name} : \mathit{str},\ \mathit{is\_capital} : \mathit{Bool},\ \mathit{country} : \mathit{Country}_E) \\
\tau_E^{\mathit{Country}_E} &\equiv (\mathit{name} : \mathit{str},\ \mathit{language} : \mathit{str},\ \mathit{currency} : \mathit{str})
\end{aligned}
$$

An instance of the second schema would consist of two sets of object identities, such as

$$
\begin{aligned}
\sigma^{\mathit{City}_E} &\equiv \{\mathit{London}, \mathit{Manchester}, \mathit{Paris}, \mathit{Berlin}, \mathit{Bonn}\} \\
\sigma^{\mathit{Country}_E} &\equiv \{\mathit{UK}, \mathit{FR}, \mathit{GM}\}
\end{aligned}
$$

and functions $\mathcal{V}^{City}$ on σ^{City} and $\mathcal{V}^{State}$ on σ^{State}, such as

$$\begin{aligned}
\mathcal{V}^{City_E}(London) &\equiv (name \mapsto \text{“London”}, country \mapsto UK, is_capital \mapsto tt)\\
\mathcal{V}^{City_E}(Manchester) &\equiv (name \mapsto \text{“Manchester”}, country \mapsto UK,\\
&\qquad is_capital \mapsto ff)\\
\mathcal{V}^{City_E}(Paris) &\equiv (name \mapsto \text{“Paris”}, country \mapsto FR, is_capital \mapsto tt)\\
\mathcal{V}^{Country_E}(UK) &\equiv (name \mapsto \text{“United Kingdom”}, language \mapsto \text{“English”},\\
&\qquad currency \mapsto \text{“sterling”})\\
\mathcal{V}^{Country_E}(FR) &\equiv (name \mapsto \text{“France”}, language \mapsto \text{“French”},\\
&\qquad currency \mapsto \text{“franc”})
\end{aligned}$$

and so on.

A **key specification**, $\mathcal{K}$, for a schema $\mathcal{S}$ with classes $\mathcal{C}$ consists of a type κ^C for each $C \in \mathcal{C}$, where κ^C contains no class types, and for any instance $\mathcal{I} \in Inst(\mathcal{S})$, a family of functions $\mathcal{K}^C_{\mathcal{I}} : \sigma^C \to [\![\kappa^C]\!]$ for each $C \in \mathcal{C}$.[3]

An instance $\mathcal{I}$ of schema $\mathcal{S}$ is said to *satisfy* a key specification $\mathcal{K}$ on $\mathcal{S}$ iff for each class $C \in \mathcal{C}$ and any $o, o' \in \sigma^C$, if $\mathcal{K}^C_{\mathcal{I}}(o) = \mathcal{K}^C_{\mathcal{I}}(o')$ then $o = o'$.

A **keyed schema** consists of a schema $\mathcal{S}$, and a key specification $\mathcal{K}$ on $\mathcal{S}$. An instance of a keyed schema $(\mathcal{S}, \mathcal{K})$ is an instance $\mathcal{I}$ of $\mathcal{S}$ such that $\mathcal{I}$ satisfies $\mathcal{K}$. We write $Inst(\mathcal{S}, \mathcal{K})$ for the instances of $(\mathcal{S}, \mathcal{K})$.

In general we will use $\mathcal{S}, \mathcal{T}, \ldots$ to range over both keyed and un-keyed schemas, and will specify either a keyed or un-keyed schema when we are interested exclusively in one or the other.

Example 5. For the European Cities and Countries schema defined in example 4 we might expect each *Country* to be uniquely determined by its name, and each *City* to be uniquely determined by its name and the name of its country (two Countries might both contain Cities with the same name). The key specification for this schema might have types

$$\begin{aligned}
\kappa^{Country_E} &\equiv str\\
\kappa^{City_E} &\equiv (name : str, country_name : str)
\end{aligned}$$

and functions defined by

$$\begin{aligned}
\mathcal{K}^{Country_E}_{\mathcal{I}} &\equiv \lambda x \cdot x.name\\
\mathcal{K}^{City_E}_{\mathcal{I}} &\equiv \lambda x \cdot (name = x.name,\ country_name = x.name.name)
\end{aligned}$$

where the notation $x.a$ means if $x \in \sigma^C$ then take the value $\mathcal{V}^C(x)$, which must be of record type, and project out the attribute a.

[3] If τ is a type which does not involve any class types, then the value of $[\![\tau]\!]\sigma^{\mathcal{C}}$ is independent of the choice of object identities, $\sigma^{\mathcal{C}}$. In this case we write $[\![\tau]\!]$ for the set $[\![\tau]\!]\sigma^{\mathcal{C}}$ for an arbitrary choice of $\sigma^{\mathcal{C}}$.

3.2 Well-defined Key-Specifications

As we remarked earlier, object-identities are generally taken to be abstract entities that are not directly visible in a database. In practice they are frequently generated as they are needed by a DBMS. Consequently we would like the meaning of a database instance to be independent of the choice of object-identities in the instance, or the order in which they were generated, and to depend only on the data represented by the instance. In particular, if two instances differ only in their choice of object identities, we would like to consider them to be the same, and to ensure that any queries or operations on those two instances give equivalent results. We define the notion of *isomorphism* to represent when two instances differ only in their choice of object-identities.

Given two instances of an unkeyed schema $\mathcal{S}$, say $\mathcal{I}$ and $\mathcal{I}'$, with families of object identities $\sigma^{\mathcal{C}}$ and $\sigma'^{\mathcal{C}}$ respectively, and a family of functions $f^C : \sigma^C \to \sigma'^C$, for $C \in \mathcal{C}$, we can extend $f^{\mathcal{C}}$ to functions on general types $f^\tau : [\![\tau]\!]\sigma^{\mathcal{C}} \to [\![\tau]\!]\sigma'^{\mathcal{C}}$, so that $f^{\underline{b}}$ is the identity on $\mathbf{D}^{\underline{b}}$ for each base type $\underline{b}$, and f^τ is defined in the obvious manner for each higher type τ.

An *isomorphism* from instance $\mathcal{I}$ to instance $\mathcal{I}'$, consists of a family of *bijective* functions $f^C : \sigma^C \to \sigma'^C$ such that for each class $C \in \mathcal{C}$ and each $o \in \sigma^C$, $\mathcal{V}'^C(f^C(o)) = f^{\tau^C}(\mathcal{V}^C(o))$. We say instances $\mathcal{I}$ and $\mathcal{I}'$ are **isomorphic** and write $\mathcal{I} \cong \mathcal{I}'$ iff there is an isomorphism $f^{\mathcal{C}}$ from $\mathcal{I}$ to $\mathcal{I}'$.

A key specification $\mathcal{K}$ on schema $\mathcal{S}$ is said to be **well-defined** if for any two instances $\mathcal{I}$ and $\mathcal{I}'$ of $\mathcal{S}$ and isomorphism $f^{\mathcal{C}}$ from $\mathcal{I}$ to $\mathcal{I}'$, if $\mathcal{I}$ satisfies $\mathcal{K}$ then so does $\mathcal{I}'$, and further, for any class $C \in \mathcal{C}$ and $o \in \sigma^C$, $\mathcal{K}^C_{\mathcal{I}}(o) = \mathcal{K}^C_{\mathcal{I}'}(f^C(o))$. Intuitively a key-specification is well-defined if it is not dependent on the particular choice of object identifiers in an instance.

For the remainder we will assume that any key specifications are well-defined.

4 Information Dominance in Transformations

One of the important questions of database systems is that of *data-relativism*, or when one schema or data-structure can represent the same data as another. From the perspective of database transformations this can be thought of as asking when there is a transformation from instances of one schema to another such that all the information in the source database is preserved by the transformation. Such a transformation would be said to be *information preserving*.

There are a number of situations when dealing with database transformations where we might want to ensure that a transformation is information preserving. For example when performing a schema evolution, we might want to

ensure that none the information stored in the initial database is lost in the evolved database, or when integrating databases, we might wish to ensure that all the information stored in one of the component databases is reflected in the integrated database.

Example 6. For the schema integration described in example 1 the transformation from the database of US Cities and States to the schema of figure 2 is information preserving, in that all the information stored in an instance of the first schema will be reflected in the transformed instance. Equally the transformation from the restructured European Cities and Countries schema of figure 3 to the schema of figure 2 is information preserving.

However the transformation from the first European Cities and Countries schema in figure 1 to the restructured schema of figure 3, and hence to the schema of figure 2, is not information preserving. This is because the transformation to the restructured schema assumes that, for each *Country* in the original schema, there is exactly one *City* of that *Country* with its *is_capital* attribute set to *True*. However the original schema allows a country to have multiple capitals: there may be many *Cities* with their *is_capital* attribute set to *True*, in which case the transformation would not be defined. If we were able to associate an additional constraint with European Cities and Countries schema of figure 1 stating that each there can be at most one capital City in each Country, then the transformation would be information preserving, and we could say that the schema of figure 2 *dominates* both of the schemas of figure 1.

In section 4.1 we will briefly describe the notions of *information dominance* defined in [15], and see how they can be related to transformations using the data model of section 3. In section 4.2 we consider the recent work of Miller in [23,24] which studies various applications of database transformations, and the need for transformations to be information preserving in these situations.

4.1 Hull's hierarchy of information dominance measures

In [15] Hull defined four progressively more restrictive notions of information dominance between schemas, each determined by some reversible transformation between the schemas subject to various restrictions. Although [15] dealt only with simply keyed flat-relational schemas, the definitions and some of the results can be easily generalized to the more general model used here.

Given two schemas, $\mathcal{S}$ and $\mathcal{T}$, a transformation from $\mathcal{S}$ to $\mathcal{T}$ is a partial map σ from instances of $\mathcal{S}$ to instances of $\mathcal{T}$, $\sigma : Inst(\mathcal{S}) \rightarrow Inst(\mathcal{T})$. Intuitively the transformation is information preserving iff there is a second transformation from $\mathcal{T}$ back to $\mathcal{S}$, say ρ such that ρ recovers the instance of $\mathcal{S}$. That is, for any $\mathcal{I} \in Inst(\mathcal{S})$, $\mathcal{I} \cong (\rho \circ \sigma)(\mathcal{I})$. (Note that we are concerned here

with transformations which preserve instances up to isomorphism, since the particular choice of object identities is immaterial.) In such a situation we say that $\mathcal{T}$ **dominates** $\mathcal{S}$ **via** (σ, ρ).

A problem with this notion is that the transformations σ and ρ may be arbitrary mathematical functions, and will not necessarily provide a semantically meaningful interpretation of the instances of $\mathcal{S}$ in terms of the instances of $\mathcal{T}$. In [15] a series of progressively more restrictive notions of **information dominance** are defined by imposing various restrictions on the transformations σ and ρ that may be used to implement a dominance relation. This series ends with the notion of *calculus dominance*: $\mathcal{T}$ is said to **dominate** $\mathcal{S}$ **calculously** iff there are expressions in some fixed calculus representing transformations σ and ρ such that $\mathcal{T}$ dominates $\mathcal{S}$ via (σ, ρ).

An important conclusion of [15], however, is that none of these criteria capture an adequate notion of semantic dominance, that is, whether there is a semantically meaningful interpretation of instances of one schema as instances of another. Consequently the various concepts of information dominance can be used in order to test whether semantic dominance between schemas is plausible, or to verify that a proposed transformation is information preserving, but the task of finding a semantically meaningful transformation still requires a knowledge and understanding of the databases involved.

Another significant problem with this analysis is that it assumes that all possible instances of a source schema should be reflected by distinct corresponding instances of a target schema. However, in practice only a small number of instances of a source schema may actually correspond to real world data sets. That is, there may be implicit constraints on the source database which are not included in the source schema, either because they are not expressible in the data-model being used or simply because they were forgotten or not anticipated at the time of initial schema design. An alternative approach, pursued in [11], is to attempt to define information preserving transformations and valid schemas with respect to some underlying *"universe of discourse"*. However such characterizations are impossible or impractical to represent and verify in practice.

4.2 Information Capacity and Constraints

In [23,24] Miller et al. analyse the information requirements that need to be imposed on transformations in various applications. The restrictions on transformations that they consider are somewhat simpler than those of [15] in that they examine only whether transformations are injective (one-to-one) or surjective (onto) mappings on the underlying sets of instances. For example they claim that if a transformation is to be used to view and query an entire source database then it must be a total injective function, while if a database is to be updated via a view then the transformation to the view must also

be surjective. Having derived necessary conditions for various applications of transformations, they then go on to evaluate existing work on database integration and translations in the light of these conditions.

An important observation in [23] is that database transformations can fail to be information capacity preserving, not because there is anything wrong with the definition of the transformations themselves, but because certain constraints which hold on the source database are not expressed in the source database schema. However the full significance of this observation is not properly appreciated: in fact it is frequently the case that the constraints that must be taken into account in order to validate a transformation have not merely been omitted from the source schema, but are not expressible in any standard constraint language.

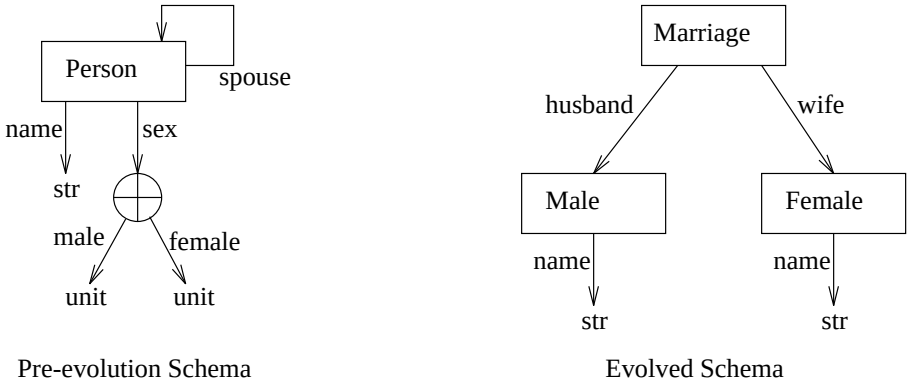


Fig. 9. An example schema evolution

Example 7. Consider the schema evolution illustrated in figure 9. The first schema has only one class, *Person*, with attributes representing a person's *name*, *sex* (a variant of *male* and *female*) and *spouse*. In our second (evolved) schema the *Person* class has been split into two distinct classes, *Male* and *Female*, perhaps because we wished to start storing some different information for men and women. Further the *spouse* attribute is replaced by a new class, *Marriage*, perhaps because we wished to start recording additional information such as dates of marriages, or allow un-married people to be represented in the database.

It seems clear that there is a meaningful transformation from instances of the first database to instances of the second. The transformation can be described

by the following *WOL* program:

$$
\begin{aligned}
&X \in \mathit{Male},\ X.\mathit{name} = N \Longleftarrow Y \in \mathit{Person},\ Y.\mathit{name} = N,\ Y.\mathit{sex} = \mathit{ins}_{\mathit{male}}();\\
&X \in \mathit{Female},\ X.\mathit{name} = N\\
&\qquad \Longleftarrow Y \in \mathit{Person},\ Y.\mathit{name} = N,\ Y.\mathit{sex} = \mathit{ins}_{\mathit{female}}();\\
&M \in \mathit{Marriage},\ M.\mathit{husband} = X,\ M.\mathit{wife} = Y\\
&\qquad \Longleftarrow X \in \mathit{Male}, Y \in \mathit{Female}, Z \in \mathit{Person},\ W \in \mathit{Person},\\
&\qquad\quad X.\mathit{name} = Z.\mathit{name},\ Y.\mathit{name} = W.\mathit{name},\ W = Z.\mathit{spouse};
\end{aligned}
$$

where "$Y.\mathit{sex} = \mathit{ins}_{\mathit{male}}()$" indicates that the sex is a male variant; details of *WOL* will be given in the next section. Although this transformation intuitively appears to preserve the information of the first database, in practice it is not information preserving. The reason is that there are instances of the *spouse* attribute that are allowed by the first schema that will not be reflected by the second schema. In particular the first schema does not require that the *spouse* attribute of a man goes to a woman, or that for each *spouse* attribute in one direction there is a corresponding *spouse* attribute going the other way. To assert these things we would need to augment the first schema with additional constraints, such as:

$$
\begin{aligned}
&X.\mathit{sex} = \mathit{ins}_{\mathit{male}}() \Longleftarrow Y \in \mathit{Person},\ Y.\mathit{sex} = \mathit{ins}_{\mathit{female}}(),\ X = Y.\mathit{spouse};\\
&Y.\mathit{sex} = \mathit{ins}_{\mathit{female}}() \Longleftarrow X \in \mathit{Person},\ X.\mathit{sex} = \mathit{ins}_{\mathit{male}}(),\ Y = X.\mathit{spouse};\\
&Y = X.\mathit{spouse} \Longleftarrow Y \in \mathit{Person},\ X = Y.\mathit{spouse};
\end{aligned}
$$

We can then show that the transformation is information preserving on those instances of the first schema that satisfy these constraints. Notice however, that these constraints are very general, and deal with values at the instance level of the database, rather than just being expressible at the schema level. They could not be expressed with the standard constraint languages associated with most data-models (functional dependencies, inclusion dependencies, cardinality constraints and so on).

This highlights one of the basic problems with information capacity analysis of transformations: Such an analysis assumes that schemas give a complete description of the set of possible instances of a database. In practice schemas are seldom complete, either because certain constraints were forgotten or were not known at the time of schema design, or because the data-model being used simply isn't sufficiently expressive. When dealing with schema evolutions, where information capacity preserving transformations are normally required, it is frequently the case that the transformation implementing a schema evolution appears to discard information, while in fact this is because the new schema is a better fit for the data, expressing and taking advantage of various constraints that have become apparent since the initial schema design.

Further, when dealing with transformations involving multiple source databases, even if the transformations from individual source databases to a

target database are information preserving, it is unlikely that the transformations will be jointly information preserving. This is in part due to the fact that the source databases may represent overlapping information, and inter-database constraints are necessary to ensure that the individual databases do not contain conflicting information. It may also be due to the fact that information describing the source of a particular item of data may be lost.

An additional limitation of the information capacity analysis of transformations is that it is very much an all-or-nothing property, and does not help us to establish other less restrictive correctness criteria on transformations. When dealing with database integration, we might only be interested in a small part of the information stored in one of the source databases, but wish to ensure that the information in this subpart of the database is preserved by the transformation. For example, we might be integrating our database of US cities and states with a database of European cities or towns and countries, and only be interested in those cities or towns with a population greater than a hundred thousand. However we would still like to ensure that our transformation does not lose any information about European cities and towns with population greater than one hundred thousand.

It therefore seems that a more general and problem specific correctness criteria for transformations is needed, such as relative information capacity. In addition, a formalism in which transformations and constraints can be jointly be expressed is needed in which to test these more general correctness criteria. As a first step in this direction we present the the language *WOL*, which provides a uniform framework for specifying transformations as well as constraints.

5 The *WOL* Language

WOL is a declarative language for specifying and implementing database transformations and constraints. It is based on the data-model of section 3, and can therefore deal with databases involving object-identity and recursive data-structures as well as complex and arbitrarily nested data-structures. Due to space limitations, we will omit or simplify certain details in the definitions and semantics of *WOL*. Full details can be found in [20].

The previous sections have shown that there are important interactions between transformations and the constraints imposed on databases: constraints can play a part in determining a transformation, and also transformations can imply constraints on their source and target databases. Although most data-models support some specific kinds of constraints, in general it is a rather ad hoc collection, included because of their utility in the particular examples that the designer of the system had in mind rather than on any sound theoretical basis. For example, relational databases will often support keys and

sometimes functional and inclusion dependencies [36], while semantic models might incorporate various kinds of cardinality constraints and inheritance [13,16]. The constraints that occur when dealing with transformations often fall outside such predetermined classes; further it is difficult to anticipate the kinds of constraints that will arise. We therefore propose augmenting a simple data-model with a general formalism for expressing constraints, such that the formalism makes it easy to reason about the interaction between transformations and constraints.

Example 8. For example, in our Cities and States database of example 4, we would want to impose a constraint that the capital City of a State is in the State of which it is the capital. We can express this as

$$X.state = Y \Leftarrow Y \in State_A, X = Y.capital$$

This can be read as "if Y is in class *State* and X is the *capital* of Y, then Y is the *state* of X". Suppose also that our States and Cities each had an attribute *population* and we wanted to impose a constraint that the population of a City was less than the population of the State in which it resides. We could express this as

$$X.population < Y.population \Leftarrow X \in City_A,\ Y = X.state;$$

Such a constraint cannot be expressed in the constraint languages associated with most data models.

We can also use constraints to express how the keys of a schema are derived:

$$X = Mk^{City_A}(name = N,\ state\_name = S)$$
$$\Leftarrow\ X \in City_A,\ N = X.name,\ S = X.state.name.$$

This constraint says that the key of an object of class *City* is a tuple built out of the *name* of the city, and the *name* of its *state*. Such constraints are important in allowing us to identify objects in transformations.

WOL is based on Horn clause logic expressions, using a small number of simple predicates and primitive constructors. However it is sufficient to express a large family of constraints including those commonly found in established data-models. In fact the only kinds of constraints which occur in established data-models but can not easily be expressed in *WOL* are finite cardinality constraints: these are constraints that might state, for example, that a certain set-valued attribute has cardinality between 2 and 3. Though it would be possible to extend *WOL* with operators to express such constraints, we have omitted them since they are of little theoretical interest and it is not clear that they are of any great practical significance.

The language *WOL* can also be used to express constraints that span multiple databases, and, in particular, can be used to specify transformations. A

transformation specification may viewed as a collection of constraints stating how data in a target database arises from data stored in a number of source databases. In general however there may be any number of target database instances satisfying a particular set of constraints for a particular collection of source instances. It is therefore necessary to restrict our attention to *complete* transformation specifications, such that for any collection of source database instances if there is a target instance satisfying the transformation specification then there is a *unique smallest* such target instance.

Possibly the closest existing work to *WOL* are the structural manipulations of Abiteboul and Hull [1] described in section 2. The rewrite rules in [1] have a similar feel to the Horn clauses of *WOL* but are based on pattern matching against complex data-structures, allowing for arbitrarily nested set, record and variant type constructors. *WOL* gains some expressivity over the language of [1] by the inclusion of more general and varied predicates (such as not-equal and not-in), though we have not included tests for cardinality of sets in *WOL*. The main contributions of *WOL* however lie in its ability to deal with object-identity and hence recursive data-structures, and in the uniform treatment of transformation rules and constraints.

The language of [1] allows nested rewrite rules which can generate more general types of nested sets, whereas in *WOL* we require that any set occuring in an instance is identifiable by some means external to the elements of the set itself. Recall that in the data-model presented here, a set occurs either as a class or as part of the value associated with some object identity. Comparing the expressive power of the two formalisms is difficult because of the difference between the underlying models, and because the expressive power of each language depends on the predicates incorporated in the language. However if the rewrite rules of [1] are extended to deal with the data-model presented here, and both languages are adjusted to support equivalent predicates (for example adding inequality and not-in tests to [1] and cardinality tests to *WOL*) then *WOL* can be shown to be at least as expressive as the rules of [1]. In particular, given the restrictions on types considered here, nested rewrite rules do not give any increase in expressive power.

5.1 Syntax and semantics of *WOL*

We will assume some fixed, keyed schema, $(\mathcal{S}, \mathcal{K})$ with classes $\mathcal{C}$, and define a version of *WOL* relative to this schema. We will write $WOL^{\mathcal{S}\mathcal{K}}$ when we wish to be explicit that the language is parameterized on a particular schema.

Terms and Atoms For each base type $\underline{b}$ we will assume a countable set of constant symbols ranged over by $c^{\underline{b}}, \ldots$, and for each type τ we will assume a countably infinite set of variables ranged over by $X^\tau, Y^\tau, \ldots$. The **terms**

of $WOL^{\mathcal{SK}}$, ranged over by $P, Q, \ldots$, are given by the abstract syntax:

$$\begin{array}{rll} P ::= & C & \text{— class} \\ | & c^{\underline{b}} & \text{— constant symbol} \\ | & X & \text{— variable} \\ | & \pi_a P & \text{— record projection} \\ | & \mathit{ins}_a P & \text{— variant insertion} \\ | & !P & \text{— dereferencing} \\ | & \mathit{Mk}^C P & \text{— object identity referencing} \end{array}$$

A term C represents the set of all object identities of class C. A term $\pi_a P$ represents the a component of the term P, where P should be a term of record type with a as one of its attributes. $\mathit{ins}_a P$ represents a term of variant type built out of the term P and the choice a. $!P$ represents the value associated with the term P, where P is a term representing an object identity. The term $\mathit{Mk}^C P$ represents the object identity of class C with key P.

We define the typing relation $\vdash$: on terms and types to be the smallest relation satisfying the rules:

$$\frac{}{\vdash C : \{C\}} \qquad \frac{}{\vdash c^{\underline{b}} : \underline{b}}$$

$$\frac{}{\vdash X^{\tau} : \tau} \qquad \frac{\vdash P : (a_1 : \tau_1, \ldots, a_k : \tau_k)}{\vdash \pi_{a_i} P : \tau_i}$$

$$\frac{\vdash P : \tau_i}{\vdash \mathit{ins}_{a_i} P : \langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle} \qquad \frac{\vdash P : C}{\vdash !P : \tau^C}$$

$$\frac{\vdash P : \kappa^C}{\vdash \mathit{Mk}^C P : C}$$

A term P is said to be **well-typed** iff there is a type τ such that $\vdash P : \tau$.

Atomic formulae or *atoms* are the basic building blocks of formulae in our language. An atom represents one simple statement about some values.

The **atoms** of $WOL^{\mathcal{SF}}$, ranged over by $\phi, \psi, \ldots$, are defined by the abstract syntax:

$$\begin{array}{rl} \phi ::= & P \doteq Q \\ | & P \dot{\neq} Q \\ | & P \dot{\in} Q \\ | & P \dot{\notin} Q \\ | & \mathsf{False} \end{array}$$

The atoms $P \doteq Q$, $P \dot{\neq} Q$, $P \dot{\in} Q$ and $P \dot{\notin} Q$ represent the obvious comparisons between terms. False is an atom which is never satisfied, and is used to represent inconsistent database states.

An atom ϕ is said to be **well-typed** iff

1. $\phi \equiv P \doteq Q$ or $\phi \equiv P \dot{\neq} Q$ and $\Gamma \vdash P : \tau$, $\Gamma \vdash Q : \tau$ for some τ; or
2. $\phi \equiv P \dot{\in} Q$ or $\phi \equiv P \dot{\notin} Q$ and $\Gamma \vdash P : \tau$, $\Gamma \vdash Q : \{\tau\}$ for some τ; or
3. $\phi \equiv \mathsf{False}$.

Intuitively an atom is well-typed iff that atom *makes sense* with respect to the types of the terms occuring in the atom. For example, for an atom $P \doteq Q$, it wouldn't make sense to reason about the terms P and Q being equal unless they were potentially of the same type.

Range restriction The concept of *range-restriction* is used to ensure that every term in a collection of atoms is bound to some constant or value occurring in a database instance. This is necessary to ensure that the truth of a statement of our logic depends only on the instance and not the underlying domains of the various types.

Suppose Φ is a set of atoms, and P is an *occurrence* of a term in Φ. Then P is said to be **range-restricted** in Φ iff one of the following holds:

1. $P \equiv C$ where $C \in \mathcal{C}$ is a class;
2. $P \equiv c^{\underline{b}}$ where $c^{\underline{b}}$ is a constant symbol;
3. $P \equiv \pi_a Q$ where Q is a range restricted occurrence of a term in Φ;
4. P occurs in a term $Q \equiv \mathit{ins}_a P$, where Q is a range-restricted occurrence of a term in Φ;
5. $P \equiv !Q$ where Q is a range-restricted occurrence of a term in Φ;
6. P occurs in an atom $P \doteq Q$ or $Q \doteq P$ or $P \dot{\in} Q$ in Φ, where Q is a range-restricted occurrence of a term in Φ;
7. $P \equiv X$, a variable, and there is a range-restricted occurrence of X in Φ.

Clauses A **clause** consists of two finite sets of atoms: the **head** and the **body** of the clause. Suppose $\Phi = \{\phi_1, \ldots, \phi_k\}$ and $\Psi = \{\psi_1, \ldots, \psi_l\}$. We write

$$\psi_1, \ldots, \psi_l \Longleftarrow \phi_1, \ldots, \phi_k$$

or

$$\Psi \Longleftarrow \Phi$$

for the clause with head Ψ and body Φ. Intuitively the meaning of a clause is that if the conjunction of the atoms in the body holds then the conjunction of the atoms in the head also holds.

For example, the clause

$$Y.\mathit{state} \doteq X \Longleftarrow X \dot{\in} \mathit{State}, Y \doteq X.\mathit{capital}$$

means that, for every object identity X in the class *State*, if Y is the capital of X then X is the state of Y.

A set of atoms Φ is said to be **well-formed** iff each atom in Φ is well-typed and every term occurrence in Φ is range restricted in Φ.

A clause $\Psi \Longleftarrow \Phi$ is said to be **well-formed** iff Φ is well-formed and $\Phi \cup \Psi$ is well-formed.

Intuitively a clause is well-formed iff it makes sense, in that all the terms in the clause range over values in a database instance, and all the types of terms are compatible with the various predicates that are applied to them. All the clauses we deal with in the remainder of this paper will be well-formed.

Semantics of *WOL* clauses An *environment* binds values to the variables occuring in a *WOL* term, atom or clause. Suppose $\mathcal{I}$ is an instance, with object-identifiers $\sigma^{\mathcal{C}}$. An **$\mathcal{I}$-environment**, ρ, is a partial function with finite domain on the set of variables such that $\rho(X^\tau) \in [\![\tau]\!]\sigma^{\mathcal{C}}$ for each variable $X^\tau \in dom(\rho)$.

If $\mathcal{I}$ is an instance, ρ an $\mathcal{I}$-environment and P a term of type τ with variables taken from $dom(\rho)$, then we define a value $[\![P]\!]\mathcal{I}\rho \in [\![\tau]\!]\sigma^{\mathcal{C}}$ by structural induction on P. We present some sample steps in the definition below. For full details see [20].

$$[\![X]\!]\mathcal{I}\rho \equiv \begin{cases} \rho(X) & \text{if } X \in dom(\rho) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![ins^a\, P]\!]\mathcal{I}\rho \equiv (a, [\![P]\!]\mathcal{I}\rho)$$

$$[\![!P]\!]\mathcal{I}\rho \equiv \begin{cases} \mathcal{V}^C([\![P]\!]\mathcal{I}\rho) & \text{if } [\![P]\!]\mathcal{I}\rho \in \sigma^C \text{ for some } C \in \mathcal{C} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![Mk^C(P)]\!]\mathcal{I}\rho \equiv \begin{cases} o & \text{if } [\![P]\!]\mathcal{I}\rho \in [\![\kappa^C]\!]\mathcal{I} \text{ and } o \in \sigma^C \\ & \text{such that } \mathcal{K}^C(o) = [\![P]\!]\mathcal{I}\rho \\ \text{undefined} & \text{otherwise} \end{cases}$$

For any well-typed atom ϕ with variables taken from $dom(\rho)$ we define a boolean value $[\![\phi]\!]\rho$. For example $[\![P \doteq Q]\!]\rho = \mathbf{T}$ iff $[\![P]\!]\rho = [\![Q]\!]\rho$; $[\![P \dot{\in} Q]\!]\rho = \mathbf{T}$ iff $[\![P]\!]\rho \in [\![Q]\!]\rho$; $[\![\mathsf{False}]\!]\rho = \mathbf{F}$ and so on.

The variables in the body of a clause are taken to be universally quantified, while any variables which occur only in the head of a clause are existentially quantified. Consequently a clause is said to be satisfied by an instance iff for any binding of the variables in the body of the clause which makes all the atoms in the body true, there is an instantiation of any remaining variables which makes all the atoms in the head true too.

Given a set of atoms Φ, we write $Var(\Phi)$ for the set of variables occuring in Φ.

Suppose $\Psi \Longleftarrow \Phi$ is a well-formed clause. An instance $\mathcal{I}$ is said to **satisfy** $\Psi \Longleftarrow \Phi$ iff for any $\mathcal{I}$-environment ρ such that $dom(\rho) = Var(\Phi)$ and $[\![\phi]\!]\rho = \mathbf{T}$ for each $\phi \in \Phi$, there is an extension ρ' of ρ (that is, $\rho'(X) = \rho(X)$ for each $X \in dom(\rho)$), such that $\rho'(\psi) = \mathbf{T}$ for each $\psi \in \Psi$.

Example 9. For the instance of the European Cities and States database described in example 4, suppose the environment ρ is given by:

$$\rho \equiv (X \mapsto UK, Y \mapsto London)$$

Then

$$\begin{aligned} [\![X \dot{\in} Country_E]\!]\rho &= \mathbf{T} \\ [\![Y \dot{\in} City_E]\!]\rho &= \mathbf{T} \\ [\![Y.country \dot{=} X]\!]\rho &= \mathbf{T} \\ [\![Y.is\_capital \dot{=} tt]\!]\rho &= \mathbf{T} \end{aligned}$$

Further we can check that, for any other binding of X to an element of $\sigma^{Country_E}$ which makes the first atom true, there is a binding of Y to an element of σ^{City_E} which makes the remaining three atoms true. Hence the instance satisfies the clause

$$Y \dot{\in} City_E,\ Y.country \dot{=} X,\ Y.is\_capital \dot{=} tt \Longleftarrow X \dot{\in} Country_E$$

5.2 Expressing database transformations using *WOL*

So far we have defined the language *WOL* to deal with a single database schema and instance. However in order to express transformations we need to be able to write *WOL* clauses concerning multiple databases. In particular we will need to write clauses involving one or more *source* databases and a distinguished *target* database.

Partitioning schemas and instances If $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are schemas with disjoint sets of classes then we can define $\mathcal{S} \equiv \mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$ by taking the classes of $\mathcal{S}$ to be the union of the classes of $\mathcal{S}_1, \ldots, \mathcal{S}_n$, and the type corresponding to each class in $\mathcal{S}$ to be the same as the type corresponding to that class in the relevant $\mathcal{S}_i$. $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are said to be a **partition** of $\mathcal{S}$.

Given instances $\mathcal{I}_1, \ldots, \mathcal{I}_n$ of disjoint schemas $\mathcal{S}_1, \ldots, \mathcal{S}_n$, we can form an instance $\mathcal{I} \equiv \mathcal{I}_1 \cup \ldots \cup \mathcal{I}_n$ of $\mathcal{S}_1 \cup \ldots \cup \mathcal{S}_n$ by taking the unions of the

components of $\mathcal{I}_1, \ldots, \mathcal{I}_n$. Further given an instance $\mathcal{I}$ of $\mathcal{S}$ and a partition $\mathcal{S}_1, \ldots, \mathcal{S}_n$ of $\mathcal{S}$, we can find unique instances $\mathcal{I}/\mathcal{S}_1, \ldots, \mathcal{I}/\mathcal{S}_n$ of $\mathcal{S}_1, \ldots, \mathcal{S}_n$ respectively, such that $\mathcal{I} = \mathcal{I}/\mathcal{S}_1 \cup \ldots \cup \mathcal{I}/\mathcal{S}_n$.

Given disjoint keyed-schemas, $(\mathcal{S}_1, \mathcal{K}_1), \ldots, (\mathcal{S}_n, \mathcal{K}_n)$, we can form a keyed schema $(\mathcal{S}, \mathcal{K}) \equiv (\mathcal{S}_1, \mathcal{K}_1) \cup \ldots \cup (\mathcal{S}_n, \mathcal{K}_n)$ in a similar manner (details may be found in [20]).

Transformation rules and constraints In looking at transformations we will concentrate on the case where we have a schema $(\mathcal{S}, \mathcal{K})$ with partition $(\mathcal{S}_{Src}, \mathcal{K}_{Src}), (\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$, and use the language $WOL^{\mathcal{SK}}$ in order to define transformations from $(\mathcal{S}_{Src}, \mathcal{K}_{Src})$ to $(\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$.

A term occuring in a set of atoms Φ is classified as a **source term** or a **target term** depending on whether it refers to a value in the source database or the target database. Note that it is possible for a term to be classified as both a source term and a target term.

For example, if $Country_E$ is a class in our source schema, $Country_T$ is a class in our target schema, and Φ is the set of atoms

$$\Phi \equiv \{X \dot{\in} Country_E,\ X.name \dot{=} N,\ Y \dot{\in} Country_T,\ Y.name \dot{=} N\}$$

then the terms X and $X.name$ are source terms, the terms Y and $Y.name$ are target terms, and the term N is both a source and a target term.

There are three kinds of clauses that are relevant in determining transformations:

target constraints — containing no source terms;
source constraints — containing no target terms; and
transformation clauses — clauses of the form $\Psi \Longleftarrow \Phi$ where each term occuring in Ψ is a target term, and $\Psi \cup \Phi$ contains no negative atoms involving target terms ($P \dot{\notin} Q$ or $P \dot{\neq} Q$), and no comparisons of set-valued target terms.

So a transformation clause is one which does not imply any constraints on the source database, and which only implies the existence of certain objects in the target database.

The restrictions against "negative" target atoms or comparisons of target sets are necessary to allow us to apply transformation clauses while the target database is only partially instantiated, and to ensure that any tests which become true at some point during the implementation of a transformation will remain true even if additional elements are added to the target database. For example, suppose we allowed the following transformation clause

$$1 \in X.a \Longleftarrow X \in C,\ Y \in C,\ X.a = Y.a$$

where C is a class with corresponding type $\tau^C \equiv (a : \{int\})$. Then suppose, at some point during the transformation, we were to find an instantiation of X and Y to two objects, say o_1 and o_2, of class C, such that the body of the clause was true at that point in the transformation. Then the clause would cause the constant 1 to be added to the set $X.a$, thus potentially making the body of the clause no longer true.

Transformation programs A transformation program, **Tr** from a schema $(\mathcal{S}_{Src}, \mathcal{K}_{Src})$ to a schema $(\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$ consists of a set of source and target constraints and transformation clauses in the language $WOL^{\mathcal{SK}}$, where $(\mathcal{S}, \mathcal{K}) = (\mathcal{S}_{Src}, \mathcal{K}_{Src}) \cup (\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$.

Example 10. Let us consider the transformation from the schema of European Cities and Countries from example 4 to the schema illustrated in figure 2. We will assume that the key for the class $Country_T$ is its *name* attribute, and the key for the class $City_T$ is a record of type (*name* : *str*, *place_name* : *str*), where the first attribute is the name of a City, and the second is the name of the Country or State pointed to by its *place* attribute.

Then we have the following source constraints:

$$Y \doteq City_E,\ Y.country \doteq X,\ Y.is\_capital \doteq tt \Longleftarrow X \doteq Country_E$$

$$X \doteq Y \Longleftarrow X \doteq City_E,\ Y \doteq City_E,\ X.country \doteq Y.country,\ X.is\_capital \doteq tt,\ .is\_capital \doteq tt$$

which state that every Country has a capital City, and that the capital City of a Country is unique.

We also need target constraints describing how the keys for our classes are generated:

$$Y \doteq Mk^{Country_T}(Y.name) \Longleftarrow Y \doteq Country_E$$

$$Y \doteq Mk^{State_T}(Y.name) \Longleftarrow Y \doteq State_E$$

$$X \doteq Mk^{City_T}(Z),\ Z.name \doteq X.name,\ Z.place\_name \doteq Y.name \Longleftarrow X \doteq City_T,\ Y \doteq Country_T,\ X.place \doteq ins_{euro\text{-}city}(Y)$$

$$X \doteq Mk^{City_T}(Z),\ Z.name \doteq X.name,\ Z.place\_name \doteq Y.name \Longleftarrow X \doteq City_T,\ Y \doteq State_T,\ X.place \doteq ins_{us\text{-}city}(Y)$$

Our transformation clauses are:

$$Y \dot{\in} City_T,\ Y.name \doteq X.name,\ Y.place \doteq ins_{euro\text{-}city}(Z)$$
$$\Longleftarrow\ X \dot{\in} City_E,\ Z \dot{\in} Country_T,\ Z.name \doteq X.country.name$$

$$Y \dot{\in} Country_T,\ Y.name \doteq Z.name,\ Y.currency \doteq Z.currency,$$
$$Y.language \doteq Z.language$$
$$\Longleftarrow\ Z \dot{\in} Country_E$$

$$Y.capital \doteq Z \Longleftarrow\ Y \dot{\in} Country_T,\ Z \dot{\in} City_T,\ Z.place \doteq ins_{euro\text{-}city}(Y),$$
$$X \dot{\in} City_T,\ X.name \doteq Z.name,\ X.country.name \doteq Y.name,$$
$$X.is\_capital \doteq tt$$

The first of the transformation clauses says that, for every object in the source class $City_E$ there is a corresponding object in the target class $City_T$ with the same value on its *name* attribute, and with its *place* attribute set to an object in class $Country_T$ with the same name as the name of the country of the city in class $City_E$. The second clause says that, for every country in the source class $Country_E$, there is a corresponding country with the same *name*, *currency* and *language*, in the target class $Country_T$, and the third clause tells us how to derive the *capital* attribute of an object in the class $Country_T$.

Note that, in the above example, a complete description of an object in the target database may be spread over several transformation rules, and that transformation rules may define one target object in terms of other target object. This highlights one of the strengths of *WOL*: it allows us to split transformations over large and complicated data-structures with many interdependencies into a number of small relatively simple rules.

The clauses of the transformation program above may however be unfolded in order to give an equivalent program in which all the clauses give complete descriptions of the target database in terms of the source database only, and which can then be implemented in a simple manner [21].

Semantics of transformation programs Suppose **Tr** is a transformation program, and $\mathcal{I}_{Src}$ an instance of $(\mathcal{S}_{Src}, \mathcal{K}_{Src})$. An instance $\mathcal{I}_{Tgt}$ of $(\mathcal{S}_{Tgt}, \mathcal{K}_{Tgt})$ is said to be an **Tr-transformation** of $\mathcal{I}_{Src}$ iff for every clause $\Psi \Longleftarrow \Phi$ in **Tr**, $\mathcal{I}$ satisfies $\Psi \Longleftarrow \Phi$, where $\mathcal{I} = \mathcal{I}_{Src} \cup \mathcal{I}_{Tgt}$.

Unfortunately the **Tr**-transformation of a particular instance will not in general be unique: a transformation program will imply that certain things must be included in the target database instance, but will not exclude other additional things from being included. Consequently there may be infinitely many **Tr**-transformations of a particular instance, representing the inclusion of arbitrary additional data. It is therefore necessary to characterize the *unique*

smallest **Tr**-transformation of an instance, when it exists. To do this, we construct a size ordering on instances, taking into account the fact that instances may have different sets of object identifiers. We refer the reader to [20] for details.

A transformation program, **Tr** is then said to be **complete** iff for any instance $\mathcal{I}_{Src}$ of $(\mathcal{S}_{Src}, \mathcal{K}_{Src})$, if there is a **Tr**-transformation of $\mathcal{I}_{Src}$, then there is a unique (up to isomorphism) smallest such **Tr**-transformation $\mathcal{I}_{Tgt}$.

Intuitively a complete transformation program is one in which the target database instance is determined unambiguously by the source instance. In particular, a transformation program is complete if, whenever it implies the existence of some object in the target database, it provides a "complete" description of that object. For example if the second transformation clause of example 10 was replaced by the clause

$$Y \dot{\in} Country_T,\ Y.name \doteq Z.name \iff Z \dot{\in} Country_E$$

(and no additional clauses were added) then the program would no longer be complete. This is because, for a suitable source instance, the above clause would imply the inclusion of an object identity in the target $Country_T$ class with some specific value for the *name* attribute of the associated record, but none of the clauses of the transformation program would assert what the *language* and *currency* attributes of the associated record should be. Consequently there would be many possible minimal instances of the target database satisfying the program, all including objects of class $Country_T$ with the appropriate *name* attribute but with arbitrary values assigned to their *language* and *currency* attributes.

Given a complete transformation program, the "unique smallest" transformation of an instance represents precisely the data whose presence in the target database is implied by the transformation program, and is therefore the transformation we are interested in.

Example 11. Consider transforming the instance described in example 4 taking **Tr** to be the transformation program of example 10. The choice of object identities in our target database is arbitrary. We will take them to be:

$$\begin{aligned} \sigma^{City_T} &\equiv \{London', Manchester', Paris', Berlin', Bonn'\} \\ \sigma^{Country_E} &\equiv \{UK', FR', GM'\} \end{aligned}$$

The mappings are then given by

$$\begin{aligned} \mathcal{V}^{City_T}(London') &\equiv (name \mapsto \text{"London"}, place \mapsto (euro\_city, UK')) \\ \mathcal{V}^{City_E}(Manchester') &\equiv (name \mapsto \text{"Manchester"}, place \mapsto (euro\_city, UK')) \\ \mathcal{V}^{City_E}(Paris') &\equiv (name \mapsto \text{"Paris"}, country \mapsto (euro\_city, FR')) \end{aligned}$$

$$\mathcal{V}^{Country_E}(UK') \equiv (name \mapsto \text{“United Kingdom”}, language \mapsto \text{“English”},$$
$$currency \mapsto \text{“sterling”}, capital \mapsto London')$$
$$\mathcal{V}^{Country_E}(FR') \equiv (name \mapsto \text{“France”}, language \mapsto \text{“French”},$$
$$currency \mapsto \text{“franc”}, capital \mapsto Paris')$$

and so on.

This is the smallest instance which is a **Tr**-transformation of the instance of example 4: there are many other **Tr**-transformations which can be formed by including additional objects to the instance. However any other *minimal* **Tr**-transformation will be isomorphic to this one. Since we can always find such a smallest **Tr**-transformation of an instance, if we can find a **Tr**-transformation at all, it follows that the transformation program is complete.

We therefore have a precise semantics for complete transformation programs. Unfortunately it is not in general decidable whether a transformation program is complete. However it is possible to construct fairly general syntactic conditions which ensure that a transformation program is complete. For programs which meet these syntactic conditions, it is also possible to efficiently compute the unique smallest transformation of a set of source instances (see [20] for details). We have prototyped such a system for a subset of *WOL* and are currently testing it on a number of sample biological database transformations [9].

6 Conclusions

There is a considerable need for database transformations in the areas of reimplementing legacy systems, reacting to schema evolutions, merging user views and integrating existing heterogeneous databases, amongst others. Though work exists to address certain aspects of these problems, a general formal approach to specifying and implementing such complex structural transformations has not yet been completely developed. Many existing approaches lack formal semantics, while others are limited in the types of transformations that can be expressed, or in the data model being considered. In addition it is frequently necessary to ensure the “correctness” of such database transformations. Notions of correctness or information preservation of transformations should therefore be tied to database transformation techniques.

In this paper we surveyed various approaches to database transformations and notions of information preservation, and reached a number of conclusions. Firstly, approaches which allow a fixed set of well-defined transformations to be applied in series are inherently limited in the class of transformations

that can be expressed. As an example we demonstrated a complex structural manipulation which could not be expressed in one such methodology, but which commonly arises in practice. Using a high-level language for expressing transformations can provide greater expressive power, but makes it more difficult to reason about and prove properties of transformations. We concluded that a high-level language is necessary in order to express general transformations, but that such a language should be *declarative* and should have a well-defined formal semantics, in order to minimize the problems of reasoning about transformations.

Secondly, the choice of an underlying data model impacts the types of transformations that can be expressed. The main requirement of the model underlying a transformation language is that it subsume the various models which might be used in the databases being transformed. In particular, it should include support for complex data-structures (sets, records and variants), object-identity and recursive structures. To reason about transforming recursive structures, it is also necessary to have a notion of extents or classes in which all the objects in a database must occur.

Thirdly, to reason that a transformation is correct, constraints should be expressed in the same formalism as the transformation. Constraints on the source and target databases are crucial to notions of information preservation, but typically are not – or cannot – be expressed in the models of the underlying databases. Furthermore, when integrating multiple heterogeneous databases it is necessary to reason about inter-database constraints. Since such constraints are crucial to the correctness of transformations they should be expressed as part of the transformation program.

These conclusions have driven the design of the transformation language *WOL*. As a declarative language built on Horn clause logic expressions, it allows a general class of transformations to be expressed and unifies the treatment of transformations and constraints. The class of constraints that can be expressed in *WOL* encompasses those found in most data models, such as keys, functional dependencies and inclusion dependences. Furthermore, our experience in using *WOL* to specify database transformations within biological databases [9] indicates that it is intuitive and easy to use since transformations over large and complicated data structures can be split into a number of relatively small and simple rules. However, while the mechanics for checking information preservation appear to be in place for *WOL* we feel that more general, problem specific notions of correctness need to be developed as well as sound techniques for proving these properties.

References

1. S. Abiteboul and R. Hull. Restructuring hierarchical database objects. *Theoretical Computer Science*, 62:3–38, 1988.

2. S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 159–173, Portland, Oregon, 1989.
3. Serge Abiteboul and Richard Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.
4. F. Bancilhon. Object-oriented database systems. In *Proceedings of 7th ACM Symposium on Principles of Database Systems*, pages 152–162, Los Angeles, California, 1988.
5. J. Banerjee, W. Kim, H. Kim, and H. Korth. Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Record*, 16(3):311–322, 1987.
6. C. Batini and M. Lenzerini. A methodology for data schema integration in the entity-relationship model. *IEEE Transactions on Software Engineering*, SE-10(6):650–663, November 1984.
7. C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.
8. P. Buneman, S. Davidson, and A. Kosky. Theoretical aspects of schema merging. In *LNCS 580: Advances in Database Technology — EDBT '92*, pages 152–167. Springer-Verlag, 1992.
9. S. B. Davidson, A. S. Kosky, and B. Eckman. Facilitating transformations in a human genome project database. In *Proc. Third International Conference on Information and Knowledge Management (CIKM)*, pages 423–432, December 1994.
10. U. Dayal and H. Hwang. View definition and generalisation for database integration in Multibase: A system for heterogeneous distributed databases. *IEEE Transactions on Software Engineering*, SE–10(6):628–644, November 1984.
11. C. Eick. A methodology for the design and transformation of conceptual schemas. In *Proceedings of the 17th International Conference on Very Large Databases, Barcelona, Spain*, pages 25–34, September 1991.
12. F. Eliassen and R. Karlsen. Interoperability and object identity. *SIGMOD Record*, 20(4):25–29, December 1991.
13. N. Hammer and D. McLeod. Database description with SDM: A semantic database model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.
14. Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3), July 1985.
15. R. Hull. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing*, 15(3):865–886, August 1986.
16. Richard Hull and Roger King. Semantic database modeling: Survey, applications, and research issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.
17. W. Kent. The breakdown of the information model in multi-database systems. *SIGMOD Record*, 20(4):10–15, December 1991.
18. Setrag N. Khoshafian and George P. Copeland. Object identity. In Stanley B. Zdonik and David Maier, editors, *Readings in Object Oriented Database Systems*, pages 37–46. Morgan Kaufmann Publishers, San Mateo, California, 1990.

19. Anthony Kosky. Observational properties of databases with object identity. Technical Report MS-CIS-95-20, Dept. of Computer and Information Science, University of Pennsylvania, 1995.
20. Anthony Kosky. *Transforming Databases with Recursive Data Structures*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, November 1995.
21. Anthony Kosky. Types with extents: On transforming and querying self-referential data-structures. PhD Thesis Proposal, Technical Report MS-CIS-95-21, University of Pennsylvania, May 1995.
22. W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.
23. R. J. Miller, Y. E. Ioannidis, and R Ramakrishnan. The use of information capacity in schema integration and translation. In *Proc. 19th International VLDB Conference*, pages 120–133, August 1993.
24. R. J. Miller, Y. E. Ioannidis, and R Ramakrishnan. Schema equivalence in heterogeneous systems: Bridging theory and practice. *Information Systems*, 19, 1994.
25. A. Motro. Superviews: Virtual integration of multiple databases. *IEEE Transactions on Software Engineering*, SE-13(7):785–798, July 1987.
26. S. Navathe, R. Elmasri, and J. Larson. Integrating user views in database design. *IEEE Computer*, 19(1):50–62, January 1986.
27. D. Penney and J. Stein. Class modification in the gemstone object-oriented dbms. *SIGPLAN Notices (Proc. OOOPSLA '87)*, 22(12):111–117, October 1987.
28. John F. Roddick. Schema evolution in database systems — An annotated bibliography. *SIGMOD Record*, 21(4):35–40, December 1992.
29. A. Rosenthal and D. Reiner. Theoretically sound transformations for practical database design. In S. T. March, editor, *Entity-Relationship Approach*, pages 115–131, 1988.
30. M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *IEEE Computer*, December 1991.
31. F. Saltor, M. Castellanos, and M. Garcia-Solaco. Suitability of data models as canonical models for federated databases. *SIGMOD Record*, 20(4):44–48, December 1991.
32. P. Shoval and S. Zohn. Binary-relationship integration methodology. *Data and Knowledge Engineering*, 6:225–249, 1991.
33. Andrea H. Skarra and Stanley B. Zdonik. Type evolution in an object oriented database. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object Oriented Programming*, pages 392–415. MIT Press, Cambridge, Massachusetts, 1987.
34. S. Spaccapietra and C. Parent. Conflicts and correspondence assertions in interoperable dbs. *SIGMOD Record*, 20(4):49–54, December 1991.
35. M. Tresch and M. Scholl. Schema transformation without database reorganization. *SIGMOD Record*, 22(1):21–27, March 1993.
36. Jeffrey D. Ullman. *Principles of Database and Knowledgebase Systems I.* Computer Science Press, Rockville, MD 20850, 1989.
37. S. Widjojo, R. Hull, and D. S. Wile. A specificational approach to merging persistent object bases. In Al Dearle, Gail Shaw, and Stanley Zdonik, editors, *Implementing Persistent Object Bases.* Morgan Kaufmann, December 1990.

38. S. Widjojo, D. S. Wile, and R. Hull. Worldbase: A new approach to sharing distributed information. Technical report, USC/Information Sciences Institute, February 1990.
39. G. Wiederhold and X. Qian. Modeling asynchrony in distributed databases. *Proc. 1987 International Conference on Data Engineering*, pages 246–250, 1987.