Specifying Database Transformations in WOL *

Susan B. Davidson
Dept. of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
Email: susan@central.cis.upenn.edu

Anthony S. Kosky
Gene Logic Inc.
Bioinformatics Systems Division
2001 Center Str, Suite 600
Berkeley, CA 94704
Email: anthony@genelogic.com

Abstract

WOL is a Horn-clause language for specifying transformations involving complex types and recursive data-structures. Its declarative syntax makes programs easy to modify in response to schema evolution; the ability to specify partial clauses facilitates transformations when schemas are very large and data is drawn from multiple sources; and the inclusion of constraints enables a number of optimizations when completing and implementing transformations.

1 Introduction

Database transformations arise in a number of applications, such as reimplementing legacy systems, adapting databases to reflect schema evolution, integrating heterogeneous databases, and mapping between interfaces and the underlying database. In all such applications, the problem is one of mapping instances of one or more *source* database schemas to an instance of some *target* schema.

The problem is particularly acute within biomedical databases, where schema evolution is pushed by rapid changes in experimental techniques, and new, domain specific, highly inter-related databases are arising at an alarming rate. A listing of the current major biomedical databases indicates that very few of these databases use commercial database management systems (see `http://www.infobiogen.fr/services/dbcat/`). One reason for this is that the data is complex and not easy to represent in a relational model; the typical structures used within these systems include sets, deeply nested record structures, and variants. A transformation language for this environment must therefore be easy to modify as the underlying source database schemas evolve, and capture the complex types used. Mappings expressed in the language must also explicitly resolve incompatibilities between the sources and target at all levels – in the choice of data-model and DBMS, the representation of data within a model, and the values of instances.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*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.

As an example of a transformation, suppose we wish to integrate a database of US Cities-and-States with another database of European-Cities-and-Countries. Their schemas are shown in Figures 1 (a) and (b) respectively, and use graphical notation inspired by [AH87]. Boxes represent *classes* which are finite sets of objects, and arrows represent *attributes*, or functions on classes.

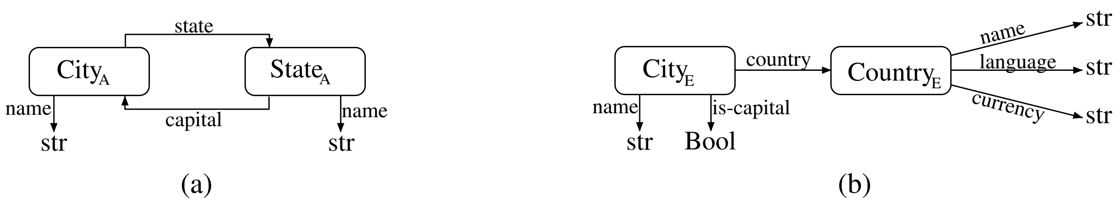

Figure 1: Schemas for US Cities and States (a) and European Cities and Countries (b)

The first schema has two classes: $City_A$ and $State_A$. The $City_A$ class has two attributes: *name*, representing the name of a city, and *state*, which points to the state to which a city belongs. The $State_A$ class also has two attributes, representing its name and its capital city. The second schema also has two classes, $City_E$ and $Country_E$. The $City_E$ class has attributes representing its name and its country, but in addition has a Boolean-valued attribute *is_capital* which represents whether or not it is the capital city of a country. The $Country_E$ class has attributes representing its name, currency and the language spoken.

A schema representing one possible integration of these two databases is shown in Figure 2, where the "plus" node indicates a variant. Note that the *City* classes from both source databases are mapped to a single class $City_T$ in the target database. The *state* and *country* attributes of the *City* classes are mapped to a single attribute *place* which takes a value that is either a *State* or a *Country*. 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. Constraints on the source database, ensuring that each *Country* has exactly one *City* for which the *is_capital* attribute is true, are also necessary in order for the transformation to be well defined.

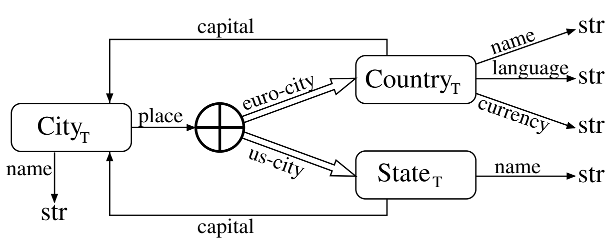

Figure 2: An integrated schema of European and US Cities

To specify exactly how this and other transformations involving complex types and recursive structures is to be performed, we have developed a language called *WOL* (Well-founded Object Logic). A number of considerations have gone into the design of this language. First, a data transformation language differs from a database query language in that entire database instances are potentially being manipulated and created. This implies a careful balance between expressivity and efficiency. Although the transformation language should be sufficiently expressive to specify all ways in which data might relate between one or more source databases and a target database, an implementation of a transformation should be performed in one pass over the source databases. This curtails the inclusion of expensive operations such as transitive closure.

Second, the size, number and complexity of schemas that may be involved in a transformation leads to a need for partiality of rules or statements of a transformation language, and for the ability to reason with constraints. Schemas – especially in biomedical applications, which formed the initial impetus for this work – can be complex, involving many, deeply nested attributes. Values for attributes of an object in a target database may be drawn from many different source database instances. In order to prevent the complexity of transforation rules becoming unmanageable, it is therefore necessary to be able to specify the transformation in a step-wise manner in which individual rules do not completely describe a target object.

Third, constraints can play a part in determining and optimizing transformations; conversely, transformations can imply constraints on their source and target databases. Many of the constraints that arise in transformations fall outside of those supported by most data-models (keys, functional and inclusion dependencies and so on) and may involve multiple databases. It is therefore important that a transformation language be capable of expressing and interacting with a large class of constraints.

In the remainder of this paper, we describe the database transformation language *WOL* and how transformation programs are implemented in a prototype system called *Morphase*[1].

2 Data Model

The data model underlying *WOL* supports object-identities, classes and complex data-structures. Formally, we assume a finite set $\mathcal{C}$ of *classes* ranged over by $C, C', \ldots$, and for each class C a countable set of *object identities* of class C. The *types* over $\mathcal{C}$, ranged over by $\tau, \ldots$, are given by the syntax

$$\tau \quad ::= \quad C \mid \underline{b} \mid (a : \tau, \ldots, a : \tau) \mid \langle\!| a : \tau, \ldots, a : \tau |\!\rangle \mid \{\tau\}$$

Here $\underline{b}$ are the built in *base types*, such as *integer* or *string*. *Class types* C, where $C \in \mathcal{C}$, represent object-identities of class C. $\{\tau\}$ are set types. $(a_1 : \tau_1, \ldots, a_k : \tau_k)$ constructs record types from the types $\tau_1, \ldots, \tau_n$, whereas $\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$ builds variant types from the types $\tau_1, \ldots, \tau_n$. A value of a record type $(a_1 : \tau_1, \ldots, a_k : \tau_k)$ is a tuple with k fields labeled by $a_1, \ldots, a_k$, such that the value of the ith field, labeled by a_i, is of type τ_i. A value of a variant type $\langle\!| a_1 : \tau_1, \ldots, a_k : \tau_k |\!\rangle$ is a pair consisting of a label a_i, where $1 \le i \le k$, and a value of type τ_i.

A database schema can be characterized by its classes and their associated types. For example, the US Cities and States schema has two classes representing cities and states. Each city has a *name* and a *state*, and each state has 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

$$\tau^{\mathit{City}_A} \equiv (\mathit{name} : \mathit{str},\ \mathit{state} : \mathit{State}_A),\quad \tau^{\mathit{State}_A} \equiv (\mathit{name} : \mathit{str},\ \mathit{capital} : \mathit{City}_A)$$

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

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

3 The *WOL* Language

WOL is a Horn-clause language designed to deal with the complex recursive types definable in the model. The specification of a transformation written in *WOL* consists of a finite set of *clauses*, which are logical statements describing either constraints on the databases being transformed, or part of the relationship between objects in

[1] *Morphase* has no relation to the god of slumber, Morpheus, rather it is an enzyme (-ase) for morphing data.

the source databases and objects in the target database. Each clause has the form *head* $\Longleftarrow$ *body* where *head* and *body* are both finite sets of *atomic formulae* or *atoms*.

The meaning of a clause is that, if all the atoms in the body are true, then the atoms in the head are also true. More precisely, a clause is *satisfied* iff, for any instantiation of the variables in the body of the clause which makes all the body atoms true, there is an instantiation of any additional variables in the head of the clause which makes all the head atoms true.

For example, to express the constraint that the capital city of a state must be in that state one would write

$$X.\mathit{state} = Y \Longleftarrow Y \in \mathit{State}_A, X = Y.\mathit{capital}; \qquad (C1)$$

This clause says that for any object Y occurring in the class State_A, if X is the *capital* city of Y then Y is the *state* of X. Here the body atoms are $Y \in \mathit{State}_A$ and $X = Y.\mathit{capital}$, and the head atom is $X.\mathit{state} = Y$. Each atom is a basic logical statement, for example saying that two expressions are equal or one expression occurs within another.

Constraints can also be used to define keys. In our database of Cities, States and Countries, we would like to say that a Country is uniquely determined by its *name*, while a City can be uniquely identified by its *name* and its *country*. This can be expressed by the clauses

$$X = \mathit{Mk}^{\mathit{City}_T}(\mathit{name} = N, \mathit{country} = C) \Longleftarrow X \in \mathit{City}_T, N = X.\mathit{name}, C = X.\mathit{country}; \qquad (C2)$$
$$Y = \mathit{Mk}^{\mathit{Country}_T}(N) \Longleftarrow Y \in \mathit{Country}_T, N = Y.\mathit{name}; \qquad (C3)$$

$\mathit{Mk}^{\mathit{City}_E}$ and $\mathit{Mk}^{\mathit{Country}_E}$ are examples of *Skolem functions*, which create new object identities associated uniquely with their arguments. In this case, the *name* of a City and the *country* object identity are used to create an object identity for the City.

In addition to expressing constraints about individual databases, *WOL* clauses can be used to express *transformation clauses* which state how an object or part of an object in the target database arises from various objects in the source and target databases. Consider the following clause

$$\begin{array}{l} X \in \mathit{Country}_T, X.\mathit{name} = E.\mathit{name}, \\ X.\mathit{language} = E.\mathit{language}, X.\mathit{currency} = E.\mathit{currency} \Longleftarrow E \in \mathit{Country}_E; \end{array} \qquad (T1)$$

This states that for every *Country* in our European Cities and Countries database there is a corresponding *Country* in our target international database with the same name, language and currency.

A similar clause can be used to describe the relationship between European *City* and *City* in our target database:

$$\begin{array}{l} Y \in \mathit{City}_T, Y.\mathit{name} = E.\mathit{name}, Y.\mathit{place} = \mathit{ins}_{\text{euro-city}}(X) \Longleftarrow E \in \mathit{City}_E, X \in \mathit{Country}_T, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad X.\mathit{name} = E.\mathit{country}.\mathit{name}; \end{array} \qquad (T2)$$

Note that the body of this clause refers to objects both in the source and the target databases: it says that if there is a City, E, in the European Cities database and a Country, X, in the target database with the same *name* as the *name* of the *country* of E, then there is a City, Y, in the target database with the same *name* as E and with *country* X. ($\mathit{ins}_{\text{euro-city}}$ accesses the *euro_city* choice of the variant).

A final clause is needed to show how to instantiate the *capital* attribute of *City* in our target database:

$$\begin{array}{l} X.\mathit{capital} = Y \Longleftarrow X \in \mathit{Country}_T, Y \in \mathit{City}_T, Y.\mathit{place} = \mathit{ins}_{\text{euro-city}}(X), E \in \mathit{City}_E, \\ \qquad\qquad E.\mathit{name} = Y.\mathit{name}, E.\mathit{state}.\mathit{name} = X.\mathit{name}, E.\mathit{is\_capital} = \mathit{True}; \end{array} \qquad (T3)$$

Notice that the definition of *Country* in our target database is spread over multiple *WOL* clauses: clause (T1) describes a country's *name*, *language* and *currency* attributes, while clause (T3) describes its *capital* attribute. This is an important feature of *WOL*, and one of the main ways it differs from other Horn-clause logic based query languages such as Datalog or ILOG[HY90] which require each clause to completely specify a target value. It is possible to combine clauses (T1), (T3) and (C3) in a single clause which completely describes how a Country object in the target database arises. However, when many attributes or complex data structures are involved, or a target object is derived from several source objects, such clauses become very complex and difficult to understand. Further if variants or optional attributes are involved, the number of clauses required may be exponential in the number of variants involved. Consequently, while conventional logic-based languages might be adequate for expressing queries resulting in simple data structures, in order to write transformations involving complex data structures with many attributes, particularly those involving variants or optional attributes, it is necessary to be able to split up the specification of the transformation into small parts involving partial information about data structures.

4 Implementing *WOL* Programs

Implementing a transformation directly using clauses such as (T1), (T2) and (T3) would be inefficient: to infer the structure of a single object we would have to apply multiple clauses. For example clauses (T1), (T3) and (C3) would be needed to generate a single *Country* object. Further, since some of the transformation clauses, such as (T1) and (T3), involve target classes and objects in their bodies, we would have to apply the clauses *recursively*: having inserted a new object into $\mathit{Country}_T$ we would have to test whether clause (T2) could be applied to that *Country* in order to introduce a new City_T object.

Since *WOL* programs are intended to transform entire databases and may be applied many times, we trade off compile-time expense for run-time efficiency. Our implementation therefore finds, at compile time, an equivalent, more efficient transformation program in which all clauses are in *normal form*. A transformation clause in normal form completely defines an insert into the target database in terms of the source database only. That is, a normal form clause will contain no target classes in its body, and will completely and unambiguously determine some object of the target database in its head. A transformation program in which all the transformation clauses are in normal form can easily be implemented in a single pass using some suitable database programming language. In our prototype implementation *Morphase*, the Collection Programming Language (CPL) [BDH+95, Won94] is used to perform the transformations.

Unfortunately, not all complete transformation programs have equivalent normal form transformation programs. Further it is not decidable whether a transformation program is *complete*, that is whether it completely describes how to derive all objects in the target database from the source databases, or whether such an equivalent normal form transformation program exists. Consequently *Morphase* imposes certain syntactic restrictions on transformation programs to ensure that they are *non-recursive*, which are easy to verify computationally and are satisfied by most natural transformations.

Within *Morphase*, constraints play an important role in completing as well as implementing transformations. Constraints on the target database can be used to complete transformations. As an example, *Morphase* will combine clauses (T1) and (T3) with the key constraint on $\mathit{Country}_T$ (C3) to generate a single clause which completely specifies how objects of class $\mathit{Country}_T$ are generated from objects in the US Cities and States database.

In a similar way, constraints on the source databases can play an important part in optimizing a transformation. As an example, suppose the following rule was generated as a result of combining several incomplete clauses:

$$
\begin{aligned}
&X = \mathit{Mk}^{\mathit{Country}_T}(N),\ X.\mathit{language} = L,\ X.\mathit{currency} = C \Longleftarrow \\
&\qquad Y \in \mathit{Country}_E,\ Y.\mathit{name} = N,\ Y.\mathit{language} = L,\ Z \in \mathit{Country}_E,\ Z.\mathit{name} = N,\ Z.\mathit{currency} = C
\end{aligned}
$$

Implementing the clause as written would mean taking the product of the source class $\mathit{Country}_E$ with itself, and trying to bind Y and Z to pairs of objects in $\mathit{Country}_E$ which have the same value on their *name* attribute. However if we had a constraint on the source database that specified *name* as a key for $\mathit{Country}_E$ the clause could be simplified to the following, more efficient, form

$$X = Mk^{Country_T}(N), X.language = L, X.currency = C \Longleftarrow Y \in Country_E, Y.name = N, \\ Y.language = L, Y.currency = C$$

5 Conclusions

Data transformation and integration in the context of biomedical databases has been a focus of research at Penn over the past six years. Two languages and systems have resulted: CPL and *WOL*. CPL has primarily been used for querying multiple heterogeneous databases, and has proven extremely effective; it is also used to implement language of *WOL* transformations. While CPL – or OQL, or any other database query language with a sufficiently rich data model – could be used for specifying transformations, they lack several useful features that are present in *WOL*. The first is a declarative syntax, which can be easily modified in response to schema evolution. The second is the ability to specify partial clauses, which we have found extremely useful when many variants are involved (as is the case with ACeDB [TMD92]). The third is the ability to capture and reason about database constraints. Reasoning about constraints is critical when partial clauses are used, since target constraints are used to normalize transformation programs. Complete details on *WOL* and Morphase can be found in [KDB95, Kos96, DK97].

The *WOL* language has also been used independently by researchers in the VODAK project at Darmstadt, Germany, in order to build a data-warehouse of protein and protein-ligand data for use in drug design. This project involved transforming data from a variety of public molecular biology databases, including SWISSPROT and PDB, and storing it in an object-oriented database, ReLiBase. *WOL* was used to specify structural transformations of data, and to guide the implementations of these transformations.

References

[AH87] Serge Abiteboul and Richard Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[BDH+95] Peter Buneman, Susan Davidson, Kyle Hart, Chris Overton, and Limsoon Wong. A data transformation system for biological data sources. In *Proceedings of 21st International Conference on Very Large Data Bases*, pages 158–169, Zurich, Switzerland, August 1995.

[DK97] S.B. Davidson and A. Kosky. WOL: A language for database transformations and constraints. In *Proceedings of the International Conference of Data Engineering*, April 1997.

[HY90] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Proceedings of 16th International Conference on Very Large Data Bases*, pages 455–468, 1990.

[KDB95] A. Kosky, S. Davidson, and P. Buneman. Semantics of database transformations. Technical Report MS-CIS-95-25, Department of Computer and Information Science, University of Pennsylvanic, Philadelphia, PA 19104-6389, July 1995. To appear in Semantics of Databases, edited by L. Libkin and B. Thalheim.

[Kos96] Anthony Kosky. *Transforming Databases with Recursive Data STructures*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, 1996. Available as UPenn Technical Report MS-CIS-96-18.

[TMD92] Jean Thierry-Mieg and Richard Durbin. ACeDB — A C. elegans Database: Syntactic definitions for the ACeDB data base manager, 1992.

[Won94] Limsoon Wong. *Querying Nested Collections*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, August 1994. Available as University of Pennsylvania IRCS Report 94-09.