Effecting Database Transformations Using *Morphase*[*]

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

Anthony S. Kosky
Lawrence Berkeley National Laboratory,
1 Cyclotron Road,
Berkeley, CA 94705.
Email: Anthony_Kosky@lbl.gov

February 23, 1996

Contact author: Anthony Kosky, Phone: (510) 486 5471, Fax: (510) 486 4004

Abstract

Database transformations are a frequent problem for data managers supporting scientific databases, particularly those connected with the Human Genome Project. The databases involved frequently contain complex data-structures not typically found in conventional databases, such as arbitrarily nested records, sets, variants and optional fields, as well as object identities and recursive data-structures. Furthermore, programs implementing the transformations must be frequently modified since the databases involved evolve rapidly, as often as 3 to 4 times a year. We present in this paper a language (WOL) for specifying transformations between such databases and describe its implementation in a system called Morphase. Optimizations are performed at all stages, with significant impact on the compilation and execution time of sample transformations.

1 Introduction

Scientific data of importance to biologists involved in the Human Genome Project (HGP) resides in a variety of databases, including relational and object-oriented systems, as well as in a mixture of data formats used for storage and data exchange, such as ASN.1 and ACeDB. These data sources frequently contain complex data types not found in conventional databases, such as arbitrarily nested records, sets, variants and optional fields, as well as object identities and recursive data-structures. As the scope of the HGP has grown, the heterogeneity and complexity of data has become increasingly problematic, and a wide variety of problems involving data transformations have arisen. Examples include sharing data between different database and software systems,

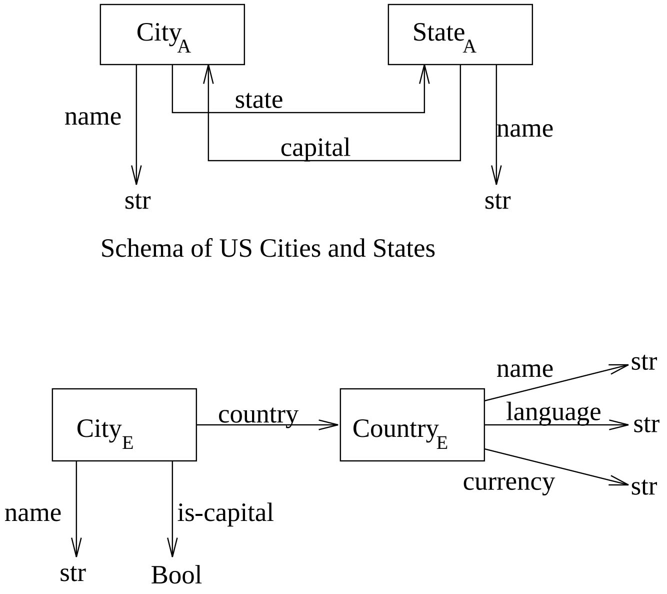
[*]This research was supported in part by DOE DE-FG02-94-ER-61923 Sub 1, NSF BIR94-02292 PRIME, ARO AASERT DAAH04-93-G0129, ARPA N00014-94-1-1086 and DOE DE-AC03-76SF00098.

querying multiple databases, supporting database evolution, migrating data from one system to another, and implementing data input utilities and user views, which typically represent data in a very different form from how it is actually stored.

In each of the data transformations alluded to above, the problem is one of mapping instances of one or more *source* database schemas to instances of some *target* database schema. Incompatibilities between the sources and target may exist at all levels — in the choice of data-model, the representation of data within the model, and the data within a particular instance — and must explicitly be resolved.

The problems of data transformation are not unique to the HGP but exist in business, academia, military and many other applications. Throughout the paper, we therefore use simple, intuitive examples that typify the complexities we have encountered rather than using actual biological examples that require significant background to understand.

Example 1.1: As a simple example, consider the problem of integrating the US Cities-and-States and European-Cities-and-Countries databases shown in Figure 1. 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. 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. (See [10] for details).



Schema of US Cities and States

Schema of European Cities and Countries

Figure 1: Schemas for US Cities and European Cities databases

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 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 *is_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 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*, 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. 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. ■

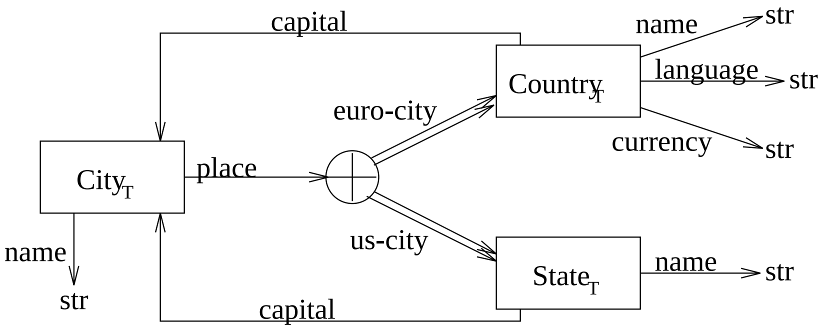


Figure 2: An integrated schema of European and US Cities

To date within the HGP, these transformation problems have been attacked by writing special-purpose programs doing explicit data conversions between fixed schemas. However, since the underlying data sources represent the results of scientific experiments, the schemas evolve rapidly in response to changing experimental techniques and requirements. These special purpose programs therefore quickly become obsolete and cannot be easily modified. It is also difficult to reason about the correctness of the transformation implemented.

Within the database research community, work in data transformation typically focuses on database integration. The most common approach taken is to apply a series of small transformations or heuristics to source schemas in order to transform them into the target schema [13, 4, 11, 14]. Unfortunately such work generally focuses on schema manipulation but neglects to describe the effect of the transformations on the actual data. Further the expressibility of such approaches is inherently limited by the selection of transformations or heuristics supported. For example, none of the systems mentioned would be able to deal with the transformation between the Boolean *is_capital* attribute of Cites and the *capital* attribute Countries in the example above. An alternative approach is to use some high-level language to describe the transformations as in [1, 8]. The problem here is to find a language which is declarative and easy to specify transformations in, but is sufficiently expressive to deal with all the transformations and data-structures likely to arise, and can be implemented in an efficient manner. The rewrite rules of [1] are the closest existing work to satisfying these goals; however, they are limited in their expressibility and cannot deal with transformations involving recursive data-structures or object identity.

In this paper, we present a declarative, Horn-clause language, *WOL* (*Well-founded Object Logic*), for specifying data transformations, and describe its implementation in *Morphase*.[1] *WOL* is based on a very general data-model, supporting complex data-types and object-identity, which is capable of representing the data-types found throughout the HGP databases as well as in more established data-models. The language, though similar in flavor to other Horn-clause based languages such as Prolog and Datalog, differs from these in allowing partial descriptions of complex data objects. Object identities are handled using methods similar to those of [9]. Section 2 gives an informal description of *WOL* through a series of examples, and illustrates how it can be used to express a wide variety of database constraints as well as transformations. We also show how constraints and transformation clauses interact with each other. Section 3 discusses how to effect non-recursive *WOL* transformations in one-pass over the underlying data sources by translation to a normal form. Although this translation significantly reduces the run-time cost of a *WOL* transformation, the translation itself (a compile time cost) is inherently expensive. We therefore present a number of optimization techniques to improve the performance of the translation process. In Section 4, we describe the *Morphase* system which implements *WOL*. Morphase translates complete, non-recursive *WOL* programs to CPL, a database programming language also developed at Penn, whose implementation within the Kleisli system can be used to query and combine data from numerous data sources important within the HGP [5]. We also describe trials in which *Morphase* was used on a transformation problem arising within the HGP at the Medical School of the University of Pennsylvania. Section 5 summarizes our contributions and describes future work.

2 The *WOL* Language

In this section we will give an informal description of the language *WOL*, based on a series of examples. A formal definition of the *WOL* language, its semantics, and the various requirements for a well-defined *WOL* transformation program may be found in [10].

The data model on which *WOL* is based supports named finite extents of object-identities (*classes*) as well as arbitrarily nested set, list, record and variant constructors. The model is basically equivalent to that of [2] and includes the features normally found in object-oriented, nested-relational or flat-relational models.

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

2.1 Formulae and Clauses

A 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 the *source* databases and objects in the *target* database. Each clause has the form

$$\textit{head-atoms} \Longleftarrow \textit{body-atoms}$$

where *head-atoms* and *body-atoms* are both finite sets of *atomic formulae* or *atoms*. An example of a simple clause for the Cities and States database shown in figure 1 would be

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

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.

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.

So the clause above 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. This is an example of a *constraint*. We can use constraints to define the *keys* of a schema that can be used to uniquely identify objects. In our database of Cities and States, we would like to say that a State is uniquely determined by its *name*, while a City can be uniquely identified by its *name* and its *state* (one can have two Cities with the same name belonging to different States). This can be expressed by the clauses

$$\begin{aligned} X = \mathit{Mk}^{\mathit{City}_A}(\mathit{name} = N, \mathit{state} = S) \Longleftarrow & \quad (C2) \\ X \in \mathit{City}_A,\ N = X.\mathit{name},\ S = X.\mathit{state}; & \\ Y = \mathit{Mk}^{\mathit{State}_A}(N) \Longleftarrow Y \in \mathit{State}_A,\ N = Y.\mathit{name}; & \quad (C3) \end{aligned}$$

$\mathit{Mk}^{\mathit{City}_A}$ and $\mathit{Mk}^{\mathit{State}_A}$ 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 *state* object identity are used to create an object identity for the City.

WOL can be used to express a wide variety of constraints, including functional and existence dependencies, key constraints, and other kinds of constraints supported by established data-models. It can also express constraints which cannot typically be expressed in the constraint languages of databases. For example, suppose that *State* and *City* 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

$$X.\mathit{population} < Y.\mathit{population} \Longleftarrow X \in \mathit{City}_A, Y = X.\mathit{state}; \quad (C4)$$

2.2 Well-formed Clauses

Not all syntactically correct *WOL* clauses are meaningful. We require two conditions to hold on a well-formed *WOL* clause, namely that it be *well-typed* and *range-restricted.* A clause is said to be *well-typed* iff we can assign types to all the variables in the clause in such a way that all the atoms of the clause make sense. For example a clause containing the atom $X < Y.population$ and an atom $X \in City_A$ would not be well-typed. For the first atom to make sense X would have to have type *integer*, and for the second it would have to be an object of class $City_A$.

The concept of *range-restriction* is used to ensure that every variable in the clause is bound to some object or value occurring in the database instance in order for the atoms of a clause to be true. This is similar to the idea of *safety* in Datalog clauses. For example the in clause

$$X.population < Y \;\Longleftarrow\; X \in City_A$$

the variable Y is not range restricted.

All the clauses we consider in this paper will be both well-typed and range-restricted.

2.3 Expressing transformations using *WOL*

In addition to expressing constraints about individual databases, *WOL* clauses can be used to express relationships between the objects of distinct databases.

Consider the clause

$$\begin{array}{l} X \in Country_T,\, X.name = E.name,\, X.language = E.language,\, X.currency = E.currency \\ \qquad \Longleftarrow\; E \in Country_E; \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (T1) \end{array}$$

This states that, for every *Country* in our European Cities and Countries database (figure 1), there is a corresponding *Country* in our target international database (figure 2) with the same name, language and currency. This is an example of a *transformation clause*, which states how an object or part of an object in the target database arises from various objects in the source and target databases.

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

$$\begin{array}{l} Y \in City_T,\, Y.name = E.name,\, Y.place = ins_{euro\text{-}city}(X) \qquad\qquad (T2) \\ \qquad \Longleftarrow\; E \in City_E,\, X \in Country_T,\, X.name = E.country.name; \end{array}$$

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. ($ins_{euro\text{-}city}$ accesses the *euro_city* choice of the variant).

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

$$\begin{aligned}&X.capital = Y \Longleftarrow && (T3)\\ &\qquad X \in Country_T,\ Y \in City_T,\ Y.place = ins_{euro\text{-}city}(X)\\ &\qquad E \in City_E,\ E.name = Y.name,\ E.state.name = X.name,\ E.is\_capital = True;\end{aligned}$$

Notice that the definition of *Country* in our target database is spread over multiple *WOL* clauses: the first clause describes a country's *name*, *language* and *currency* attributes, while the third clause describes its *capital* attribute. This is an important feature of *WOL*: in order to simplify writing transformations involving complex data-structures with many fields, and particularly those involving variants, it is useful to be able to split up the specification of the transformation into small parts involving partial information about the data-structures.

Another point to notice is that, in order to combine these two clauses to get a full description of a Country in the target database, we need some way of determining when the two clauses refer to the same object in the $Country_T$ class; that is, a way of uniquely identifying objects. To do this we can combine these two clauses with a key constraint for the class $Country_T$:

$$X = Mk^{Country_T}(N) \ \Longleftarrow\ X \in Country_T,\ N = X.name; \quad (C5)$$

This illustrates an important principle on which *WOL* is based: there are important interactions between database transformations and constraints. Firstly, as we have just shown, constraints may play a part in determining transformations; secondly, in order for a transformation to be well-defined it may imply certain constraints on the source and target databases; and thirdly, constraints on source databases are important in optimizing transformations as will be seen in section 3.2.

A *transformation program* therefore consists of a finite set of transformation clauses and constraints for some source and target database schemas. Given such a transformation program, say **Tr**, a **Tr**-transformation of an instance of the source database would be an instance of the target database such that the two instances satisfy all the clauses in **Tr**.

Since *WOL* clauses represent logical statements, there may be many instances of a target database satisfying a set of clauses for a particular source database. For example the clauses (T1), (T2) and (T3) above would imply that there are objects in our target *Country* class corresponding to each *Country* in our source database, but would not rule out the possibility of having lots of other Countries, not related to any in our source database. When dealing with transformation programs, we are therefore interested in the *unique smallest* transformation of a particular source database.

A transformation program **Tr** is said to be *complete* iff whenever there is a **Tr**-transformation of a particular source database instance, there is a unique smallest such **Tr**-transformation (upto renaming of object identities). In general, if a transformation program is not complete, it is because the programmer has left out some part of the description of the transformation. The algorithms used in the *Morphase* system, described below, will detect incomplete transformation programs and indicate to the programmer where additional information is needed.

3 Normal-Form *WOL* Programs

To implement a transformation directly using clauses such as (T1), (T2) and (T3) would be difficult and computationally expensive: to infer the structure of a single object we would have to apply multiple clauses, for example clauses (T1), (T3) and (C5) would be needed for 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 $Country_T$ we would have to test whether clause (T2) could be applied to that *Country* in order to introduce a new $City_T$ object.

Since a *WOL* program may be used to transform entire databases and may be used multiple times for different instances, we choose an implementation strategy which at compile time finds 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. For example a normal form transformation clause for our target class $Country_T$ would be

$$\begin{array}{l} X \in Country_T,\ X = Mk^{Country_T}(N),\ X.name = N,\ X.language = L,\ X.currency = C, \qquad (N1) \\ X.capital = Mk^{City_T}(name = Z.name, place = N) \\ \qquad\qquad \Longleftarrow\ Y \in Country_E,\ Y.name = N,\ Y.language = L,\ Y.currency = C, \\ \qquad\qquad\qquad Z \in City_E,\ Z.country = Y,\ Z.is\_capital = True; \end{array}$$

Given a transformation program in which all the transformation clauses are in normal form, the transformation may then be easily implemented in a single pass using some suitable database programming language. Such a transformation program is said to be in *normal form.* Implementing the normal-form transformation program rather than the original program significantly improves the run-time performance of the transformation.

Unfortunately, not all complete transformation programs have equivalent normal form transformation programs, and it is not decidable whether a transformation program is complete or such an equivalent normal form transformation program exists. We therefore place syntactic restrictions on transformation programs so that they are *non-recursive*, such that most natural transformations satisfy these restrictions.

3.1 Computing Normal Form Transformation Programs

The algorithm for computing equivalent normal form transformation programs works by starting with a clause which *describes* one object in the database, and repeatedly unfolds that clause using other clauses of the *WOL* program until a normal form clause is reached. The *description clauses*, with which the algorithm starts, have a set of atoms completely describing an object as both their head and their body. For example a description clause for the *Country*$_T$ class would be:

$$\begin{aligned} & X \in Country_T,\, X = Mk^{Country_T}(Y),\, X.name = N, \qquad (A1)\\ & X.language = L,\, X.currency = C,\, X.capital = Z \\ & \quad \Longleftarrow\ X \in Country_T,\, X = Mk^{Country_T}(Y),\, X.name = N, \\ & \qquad X.language = L,\, X.currency = C,\, X.capital = Z; \end{aligned}$$

The process of *unfolding* such a clause involves unifying its variables with the variables of a program clause, removing any atoms from its body for which equivalent atoms occur in the head of the program clause, and replacing them with the atoms in the body of the program clause. Note that atoms can only be removed from the body of the clause being unfolded if doing so does not break the range restriction of the clause. For example unfolding (A1) on the clause (C5) and unifying $(Y \mapsto N,\, X \mapsto X,\, N \mapsto N,\, C \mapsto C,\, L \mapsto L,\, Z \mapsto Z)$ would give:

$$\begin{aligned} & X \in Country_T,\, X = Mk^{Country_T}(N),\, X.name = N, \qquad (A2)\\ & X.language = L,\, X.currency = C,\, X.capital = Z \\ & \quad \Longleftarrow\ X \in Country_T,\, X.name = N, X.language = L,\, X.currency = C,\, X.capital = Z; \end{aligned}$$

Unfolding (A2) on the clause (T1) gives

$$\begin{aligned} & X \in Country_T,\, X = Mk^{Country_T}(E.name),\, X.name = E.name, \qquad (A3)\\ & X.language = E.language,\, X.currency = E.currency,\, X.capital = Z \\ & \quad \Longleftarrow\ X \in Country_T,\, X.capital = Z,\, E \in Country_E; \end{aligned}$$

As each unfolding is performed, the algorithm decorates each atom by marking it with the transformation clauses that have been used in generating it. Further, any atoms introduced by an unfolding are decorated with the marks of any of the atoms that are matched by the unfolding. For example, the atoms $X \in Country_T$ and $X.name = N$ in $(A2)$ and the atom $X \in Country_T$ in $(A3)$ would be marked as generated by $(C5)$; the atom $E \in Country_E$ in $(A3)$ would be marked with both $(C5)$ and $(T1)$. We disallow an unfolding using a transformation clause if no atoms that have not already been marked with this clause are matched and removed by the unfolding. This process continues until no more unfoldings are possible, or until the clause is in normal-form.

Pseudo-code for this naive algorithm for computing equivalent normal form transformation programs is shown in figure 3. The next subsection will describe a number of optimizations to improve it.

We can prove that this algorithm always terminates; however, if it terminates in a failure there might actually be an equivalent normal form *WOL* program which was not found. Since determining whether there is an equivalent normal form *WOL* program is an undecidable property, we have imposed syntactic conditions to characterize a *WOL* program as *recursive* if it has an infinite unfolding sequence. This characterization of recursion is a finer notion than that of Datalog[6], in that it allows clauses such as (T3) which define part of an object in terms of other parts of the same object or other objects in the same class. However clauses such as the following, which represents

```
Generate all description clauses;
REPEAT
    FOR each transformation clause $\Delta$ DO
        Generate unfoldings of description clauses on $\Delta$;
        FOR each unfolding $\mathcal{U}$ DO
            IF all atoms matched in $\mathcal{U}$ are marked with $\Delta$
            THEN FAIL ("recursive program");
            ELSE add unfolded clause to description clauses;
        OD
        mark atoms introduced in description clauses with $\Delta$;
    OD
    UNTIL no more unfoldings are possible;
Test for incomplete clauses;
RETURN normal form clauses;
```

Figure 3: Naive algorithm for generating normal form clauses

transitive closure, are clearly recursive:

$$\begin{aligned} &W \in C,\, W.a = X,\, W.b = Y \;\Longleftarrow \\ &\qquad U \in C,\, U.a = X,\, U.b = Z,\, V \in C,\, V.a = Z,\, V.b = Y; \end{aligned}$$

If we included this clause in a transformation program then we could unfold it infinitely many times, never reaching a normal form. (See [10] for a complete characterization of recursive *WOL* programs).

If the normal form program terminates with a failure, the user may use error information provided to reformulate the original program in a way that could be normalized.

3.2 Optimizing the normalization algorithm

The algorithm suggested in section 3.1 first generates description clauses for each class, and then finds maximal sequences of unfoldings starting from each description clause to see which ones end in normal form clauses. Generating the maximal unfolding sequences involves doing a breadth-first unfolding of each description clause on all the transformation rules, a process that is inherently exponential. In addition, at each stage there may be many possible unifiers for each transformation rule, and the number of description clauses is potentially exponential in the number of set and variant type constructs in the schema. It is clear that such a naive algorithm would be infeasible.

Fortunately, we can reduce the search space and restrict our attention to a small subset of relevant unfolding sequences. Many unfolding sequences are equivalent in that they differ only in the order in which rules are applied, but result in the same final clauses, while others are subsumed by more general unfolding sequences or can never reach a normal form. We therefore have two objectives in optimizing the normalization process:

1. explore as few equivalent unfolding sequences as possible; and

2. discontinue unproductive unfolding sequences as early as possible.

We have developed a number of optimizations aimed at these two goals, which combine to make a practically feasible normalization algorithm. We describe some of these optimizations below.

Using maximal unifiers. The idea behind maximal unifiers is to unify as many variables as possible at each stage. For example if we are unfolding the clause

$$\Psi \Longleftarrow X = C, Y \in X, U = Y.a, V = Y.b;$$

using the transformation clause

$$X' = C, Y' \in X', U' = Y'.a \Longleftarrow \Phi;$$

(where Ψ and Φ denote some sets of atoms), there is an obvious unifier which may be written $(X \mapsto X', Y \mapsto Y', U \mapsto U')$ (X matches X', Y matches Y' and so on). However there are also many other possible unifiers, such as $(U \mapsto U')$ and $(Y \mapsto Y', U \mapsto U')$ and so on. In general the number of unifiers is exponential in the number of variables. However we can prove that we will not lose anything if we unify as many variables as possible at each stage.

Ordering transformation rules. In general, there may be many equivalent unfolding sequences for a particular target clause, differing only in the order in which they apply clauses. For example, suppose we have a target class C with attributes a and b, and we are unfolding the clause

$$\Psi \Longleftarrow X \in C, X.a = Y, X.b = Z; \quad (A4)$$

with the transformation clauses

$$\begin{array}{ll} X \in C, X.a = Y \Longleftarrow \Phi_1; & (T4) \\ X \in C, X.b = Z \Longleftarrow \Phi_2; & (T5) \end{array}$$

Then we can either unfold $(A4)$ first on $(T4)$ and then on $(T5)$, or first on $(T5)$ and then on $(T4)$. In either case the result is

$$\Psi \Longleftarrow \Phi_1, \Phi_2$$

so it is unnecessary to consider both unfolding sequences.

We can avoid such multiple unfolding paths by assuming some arbitrary ordering on the transformation clauses of a program. For example, if we decided that $(T4)$ came before $(T5)$ in the ordering, then we would not attempt to unfold on $(T4)$ after having unfolded on $(T5)$.

A problem with this approach is that unfolding on one clause might enable an unfolding on another clause which was not previously possible. For example, suppose we have two transformation clauses:

$$\begin{array}{ll} X \in C, X.a = Y \Longleftarrow \Phi_1; & (T4') \\ X \in C, X.b = Z \Longleftarrow X \in C, X.a = Y, \Phi_2; & (T5') \end{array}$$

where $(T4')$ comes before $(T5')$ in our ordering, and we were trying to unfold a clause

$$\Psi \Longleftarrow X \in C,\, X.b = Z; \quad (A4')$$

Then $(A4')$ is not unfoldable on $(T4')$. However we can unfold $(A4')$ on $(T5')$ followed by $(T4')$, to get the clause

$$\Psi \Longleftarrow \Phi_1, \Phi_2;$$

This unfolding sequence should be allowed, even though $(T4')$ comes before $(T5')$ in the ordering, because the unfolding on $(T4')$ involves matching some atoms that were not available at the start of the unfolding sequence. We can use the markings of atoms discussed in section 3.1 in order to avoid this problem: an unfolding on $(T4')$ is allowed after an unfolding on $(T5')$ provided that the unfolding on $(T4')$ matches some atom that is marked as being generated by $(T5')$, or by some other clause that comes after $(T4')$ in the ordering.

Avoiding redundant unfoldings. An unfolding on one clause can also be made redundant by an unfolding on some later clause in an unfolding sequence. For example if we had transformation clauses

$$\begin{array}{lr} Z \in C,\, X = Z.a,\, Y = Z.b \Longleftarrow W \in D_1,\, X = W.a,\, Y = W.b; & (T6) \\ Z \in C,\, X = Z.a,\, Y = Z.c \Longleftarrow W \in D_2,\, X = W.a,\, Y = W.c; & (T7) \\ Z \in C,\, X = Z.a,\, Y = Z.b,\, X = Z.c \Longleftarrow W \in D_3,\, X = W.a,\, Y = W.b; & (T8) \end{array}$$

and we are unfolding a clause

$$\Psi \Longleftarrow Z \in C,\, X = Z.a,\, Y = Z.b, V = Z.c; \quad (A5)$$

Then unfolding $(A5)$ first on $(T6)$ and then on $(T8)$ yields the clause

$$\Psi[Y/V] \Longleftarrow U \in D_1,\, X = U.a,\, Y = U.b,\, W \in D_3,\, X = W.a,\, Y = W.b;$$

which is subsumed by the clause reached by just unfolding $(A5)$ on $(T8)$:

$$\Psi[Y/V] \Longleftarrow W \in D_3,\, X = W.a,\, Y = W.b;$$

Equally an application of $(T7)$ would be made redundant by a following application of $(T8)$, though an application of $(T6)$ could usefully follow an application of $(T7)$ and vice versa. In general an unfolding on some clause will make an earlier unfolding redundant if it matches all the atoms matched by the earlier unfolding. We can use a system of markings of atoms, similar to those described before, to detect and avoid such unfoldings.

Dynamically altering transformation programs. The rules of a transformation program will in general define the objects of one target class in terms of the objects of other target classes. Consequently it is often necessary to repeat a series of unfoldings in several different unfolding sequences, possibly for different target classes.

For example, suppose our target schema had four classes, $\{C_1, C_2, C_3, C_4\}$, our source schema contained three classes, $\{D_1, D_2, D_3\}$, and our transformation program **Tr** consisted of the clauses:

$$\begin{array}{lr}
X \in C_1,\ U = X.a,\ V = X.b & (T9) \\
\qquad \Longleftarrow Y \in C_3,\ U = Y.a,\ V = Y.b,\ Z \in D_1,\ U = Z.a; & \\
X \in C_2,\ U = X.a,\ V = X.b & (T10) \\
\qquad \Longleftarrow Y \in C_3,\ U = Y.a,\ V = Y.b,\ Z \in D_2,\ V = Z.b; & \\
Y \in C_3,\ U = Y.a, V = Y.b \Longleftarrow Z \in C_4,\ U = Z.a,\ V = Z.b; & (T11) \\
Z \in C_4,\ U = Z.a, V = Z.c \Longleftarrow W \in D_3,\ U = W.a,\ V = W.b; & (T12)
\end{array}$$

Then in order to get a normal form clause for the class C_1 it is necessary to unfold a description clause on first $(T9)$, then $(T11)$ and then $(T12)$; to find one for C_2 it is necessary to unfold on $(T10)$ then $(T11)$ and then $(T12)$; and to find one for C_3 it is necessary to unfold on $(T11)$ then $(T12)$. Clearly, there is duplication of effort here, and we could improve efficiency by "memo-izing" the result of unfolding $(T11)$ then $(T12)$. In particular, we can first generate a new rule, say $(T11')$, by unfolding $(T11)$ on $(T12)$:

$$Y \in C_3,\ U = Y.a, V = Y.b \Longleftarrow W \in D_3,\ U = W.a,\ V = W.b; \quad (T11')$$

We can then use this rule in place of $(T11)$ in our transformation program. This replaces the repeated unfolding sequences with single unfoldings on $(T11')$.

Note that, in general, unfolding a transformation clause may introduce new atoms into its head. Therefore, it is possible that replacing rules with memo-ized rules this way will increase the number of rules in our program which can match a particular set of atoms. For example, suppose we had rules

$$\begin{array}{ll}
X \in C, Y = X.a \Longleftarrow X \in C,\ Z = X.b, \Phi_5; & (T13) \\
X \in C,\ Z = X.b,\ W = X.c \Longleftarrow \Phi_6; & (T14)
\end{array}$$

where Φ_5 and Φ_6 contain no target terms. Then unfolding $(T13)$ on $(T14)$ returns the partial normal form rule

$$X \in C, Y = X.a,\ W = X.c \Longleftarrow \Phi_5, \Phi_6; \quad (T13')$$

If we replaced $(T13)$ with $(T13')$, there would then be two clauses with heads containing atoms to match $X \in C$ and $W = X.c$ instead of one. Consequently, when unfolding other clauses which had relatively base variables with this term path, there would be more clauses to unfold them on.

This increase in the number of clauses matching a particular set of atoms could potentially outweigh any advantage gained by converting rules into partial normal form. However we can avoid this problem by only considering the sets of atoms in the head of the original program clause when we replace it with a memo-ized clauses. In the above example we would then count the head atoms of $(T13')$ as being $X \in C$, $Y = X.a$ — the same as those of $(T13)$ — and only consider these when trying to unfold clauses on $(T13')$. It is safe to do this since any new atoms in the head of $(T13')$ would have been generated by some other clause, and will therefore already be implied by other clauses in the transformation program.

The optimizations described above and others are described in more detail in [10].

4 *Morphase*

The system which compiles *WOL* programs into CPL using the algorithms and techniques of the previous section is called *Morphase*. A prototype of *Morphase* written in ML [12] and based on a restricted version of the *WOL* language, has been implemented and used in several trial transformations within the Philadelphia Genome Center for Chromosome 22 (see [7, 10] for details). In this section, we describe *Morphase* and how transformations are effected, and briefly review the results of one of our trials.

4.1 Architecture

The architecture for the *Morphase* system is illustrated in figure 4. As shown in this figure, *WOL* transformation rules are typically written by the user of the system; however a large number of constraints which complete the transformation rules can frequently be automatically generated from the meta-data associated with the source and target databases. The kind of constraints that can be derived in this way depend on the particular DBMSs being used, but frequently include type information, keys and some other dependencies. Such constraints represent a significant part of a transformation program, but are time consuming and tedious to program by hand. Deriving them directly from meta-data therefore reduces the amount of grunge work that needs to be done by the programmer, and allow him or her to concentrate on the structural part of a transformation.

The translation of a *WOL* transformation program has several stages. The program is first translated into *semi-normal form* (*snf*). This involves a rewriting of the *WOL* clauses in order to reduce the number of forms the atoms of a clause can take, so that any two equivalent clauses or sets of atoms will differ only in their choice of variables. The purpose of the semi-normal form is to simplify the unification of clauses, and also to make keeping track of certain information for the optimizations simpler. (However we avoided using snf clauses for the examples of the previous section, since they are generally larger and more difficult to read). The snf program is then transformed into a normal-form program if possible, using the methods of section 3.

Having transformed a *WOL* program to normal form, it must then be executed against the source databases to produce the target database. The problem, as alluded to in the introduction, is that the source as well as target databases are heterogeneous. We therefore compile complete, normal form *WOL* programs into CPL, a database programming language developed at the University of Pennsylvania. The reason for using CPL as an implementation language is that it supports many

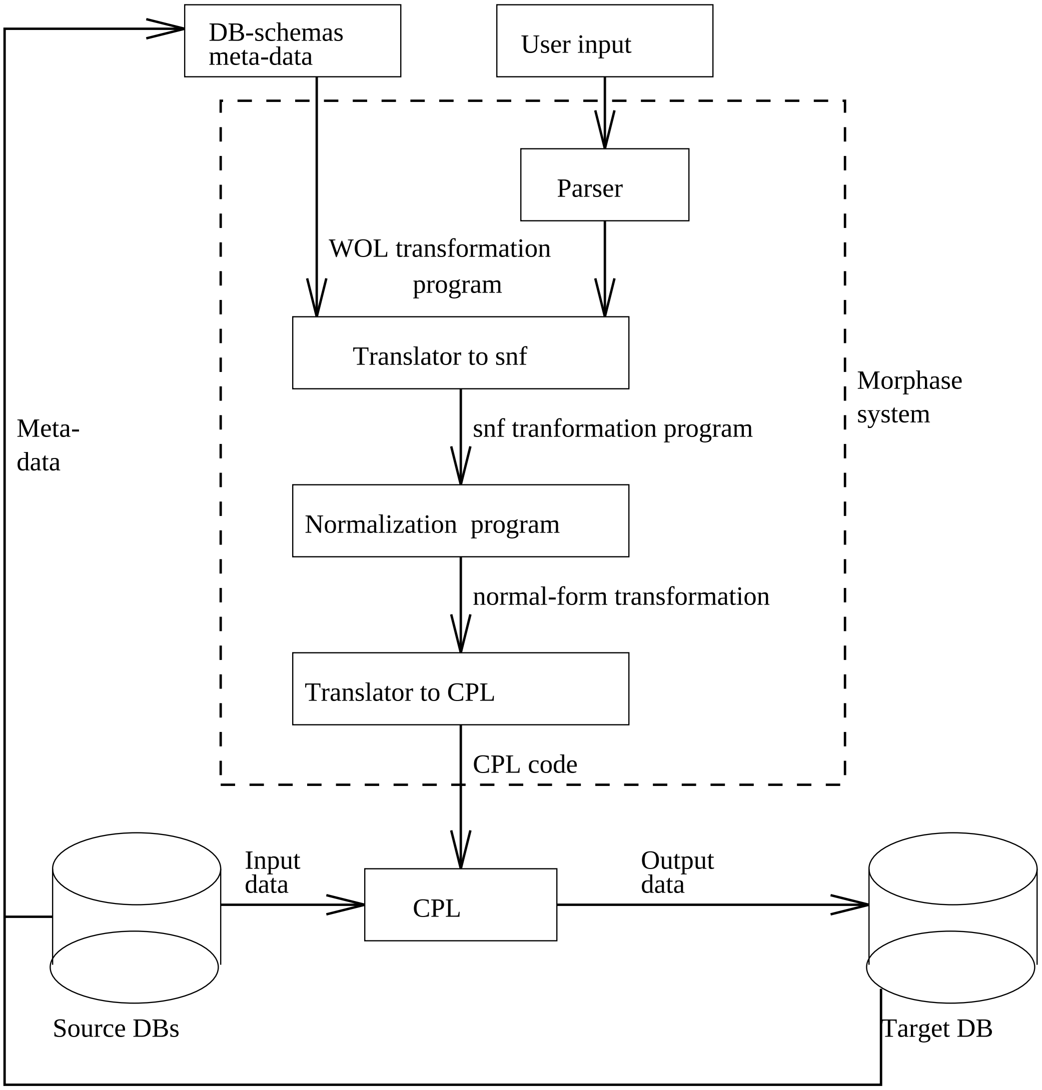


Figure 4: Architecture of the *Morphase* system

of the data-types that we are interested in and is easily extensible. The Kleisli system accesses a wide variety of database systems, including those that we wished to use in our trials; furthermore, it is an easy task to add additional data drivers as new database systems are encountered (see [5] for details of the Kleisli system). However translating normal form *WOL* programs into some other sufficiently expressive DBPL should be a straightforward task, so the implementation should be easily adaptable to other systems.

4.2 Trials

In order to test the feasibility of *Morphase* a series of trials were carried out using a prototype implementation of *WOL*. These trials gave us significant insights into optimizations for the normalization procedure, including those described in section 3.2, and they were subsequently incorporated into the prototype. We are currently working on a full implementation of *WOL* in which we expect to achieve significant performance improvements over the prototype by using more efficient data-structures.

Rather than go into details of all the trials, we will focus on one involving the two primary centers of information about Chromosome 22, ACe22DB and Chr22DB, since the transformation was structurally complex. ACe22DB is an ACeDB[2] database located at the Sanger Centre in Cambridge, England. ACeDB represents data in tree-like structures with object identities, and is well suited for representing "sparsely populated" data [15]. Chr22DB is a Sybase relational database maintained by the Philadelphia Genome Center for Chromosome 22. The informatics groups supporting these databases must routinely download data from the complementary database to keep their local database up-to-date, aid in planning experiments, and carry out mapping activities. Unfortunately, data is structured very differently in the two databases, since it is based on incompatible data-models, as well as on different interpretations of the underlying data and how it should be structured.

A series of *WOL* programs transforming various molecular biology data in Chr22DB to ACe22DB were written and tested along three measures: ease of use, compilation time, and size and complexity of the resulting normal form program. The size and complexity of the normal form program is an indirect indicator of the execution time of the actual transformation, since the Kleisli optimizer will actually rewrite the CPL code to a more efficient form. Furthermore, we felt that measuring the execution time of the CPL program reflects more on Kleisli than on *Morphase*.

Ease of use is a qualitative measure, so we can only report our observations. *WOL* transformations were programmed by researchers connected with the Philadelphia Genome Center, who used their domain knowledge to formulate the transformations. Learning *WOL* and using it to express the transformations was found to be an easy and natural process. The most difficult part of the process was in fact understanding the foreign databases, and the semantic differences between the databases. As ACe22DB and Chr22DB have evolved over time, it has also been relatively easy to

[2] A c. elegans Database (AceDB). While "C elegans" may sound like a contradiction in terms to a computer scientist, it is actually a small worm.

Program	Compilation Time	Number of CPL Primitives
STS.direct	5:33.63	37
STS.xref	1:53:32.70	515
STS.constr	37:28.16	205

Figure 5: Comparison of performance of various versions of the Chr22DB to ACe22DB transformation program

modify the original *WOL* program to reflect schema changes.

The compilation time and complexity of the normal form program depend heavily on how the initial *WOL* program is written. Various versions of the transformation program were therefore tried to see how they affected these measures. The versions divided into two basic types: *"direct"* programs and *"external-reference"* programs. All transformation clauses of the direct versions were already in partial normal-form: that is, they contained no target atoms in their bodies. Consequently normalizing the direct versions involved no sequences of multiple unfoldings, and resulted in the smallest possible number of normal form clauses. The time taken to normalize direct programs showed us how much time was taken by the system on such tasks as parsing, type-checking, converting clauses to semi-normal form, building description clauses and so on. In a sense this provided a upper bound on the performance we might hope to achieve. The *external reference* versions of the program included various external reference constraints of the ACe schema, and used these constraints to generate many of the attributes of certain classes rather than deriving them directly from the source database. These versions of the program involved a considerable number of unfoldings in order to build normal form clauses.

In addition the programs were tried with and without various key and other constraints, which provided an indication of how effective optimizations based on such constraints were, and how important proper constraint information is to performing efficient transformations. It was found that certain constraints, in particular *keys*, were essential in obtaining acceptable performance from the system. For this reason various optimizations for the application of key constraints will be incorporated in the full implementation of *Morphase*.

Figure 5 shows some results of these experiments; the times quoted were measured using the unix *timex* utility. The *STS.direct* program was a direct implementation of the transformation using system-generated object identities. The *STS.xref* and *STS.constr* versions make use of the ACe external reference constraints in order to instantiate the target tables. In *STS.xref* we avoided using system generated object identities, and replaced them with values for which key constraints were not available. This implementation was somewhat unrealistic, but illustrated the importance of keys in implementing transformations efficiently. The *STS.constr* also makes use of system generated Skolem functions to implement object identities for each class.

As can be seen from these results, the use of transformation programs based on external-references may increase the time taken for normalizing a program by a factor of six compared to the direct versions. Nevertheless the times remained tolerable for a one-time operation. Also notable was the fact that the size of the resultant normal form programs varied dramatically: in this particular example ranging from 205 clauses down to 37 for otherwise similar programs. This phenomenon could be attributed to an unusual feature of the ACeDB data-model: virtually all attributes are optional. Because of this, a large number of normal form clauses which only partially instantiate target data-structures are generated, even though we would not expect these clauses to be used in practice. It can be noted that more complete constraint information rules out many of these extra normal form clauses, and provides better performance as well as a normal form program which is closer to that of the direct version of the program.

5 Conclusions

The *WOL* language is well-suited for the structural manipulations of complex data-types found in data transformation problems within the HGP. It is simple, declarative, and allows clauses which only partially specify a target object. These qualities make it easy for non-database specialists to use, and make *WOL* programs easy to modify as the schemas of source and target databases evolve.

Although the examples in section 2 only illustrate manipulations involving sets, records, variants and object identities, in general *WOL* allows arbitrary nesting of these types as well as lists and bags. Furthermore, although our trials of Morphase only discuss ACeDB and Sybase data sources, it is quite easy to adapt the system to accommodate new data sources. Beyond adding new data-drivers to CPL (we currently support ASN.1 in addition to ACeDB and Sybase), the only other task is to add mechanisms for reading meta-data from the new data sources and translating the meta-data into *WOL* constraints.

Since *WOL* is compiled to CPL, which is being used to query and combine results from the multiple, heterogeneous databases, a natural question is why don't we directly use CPL for these transformations and eliminate the compile-time overhead of *Morphase*? There are several answers to this questions.

First, no one language is right for every task. CPL is a query language for these complex types, and is therefore strictly more powerful than *WOL*. In contrast, *WOL* is a specification language which has been designed be simple and easy to reason about. It also captures constraints in the same paradigm as transformations; *Morphase* capitalizes on interactions between the two during compilation to CPL.

Second, we want deal with structural manipulations that can be performed efficiently on large quantities of data, and to avoid recursive transformation programs. The practical motivation for this is that computing recursive programs, such as the transitive closure of an entire database, could have a very high run-time cost. We also feel that people don't really want to *store* something like the transitive closure of a graph, and prefer more compact representations.

Third, much of the compile time overhead of *WOL* arises because of features introduced to simplify the specification of a structural transformation. For example, in the trials we conducted people frequently took advantage of the ability to write partial clauses. They also frequently wrote incomplete programs, which were completed by the automatic inclusion of constraints from the source databases. The trade-off is the compile-time of *WOL* programs versus programmer coding time.

In future work, we want to complete the implementation of *WOL* in Morphase. We also need to consider how well the implementation scales to larger problems than we have considered (HGP databases are typically rather small, less than a gigabyte). It is also clear that there is a potential for graphical schema manipulation tools generating *WOL* transformation programs, which would improve the user-interface to the system.

Acknowledgements. We would like to thank Peter Buneman for his help, suggestions and examples, and Barbara Eckman and Carmem Hara for their help with the *Morphase* trials.

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] 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.

[5] P. Buneman, S.B. Davidson, K. Hart, C. Overton, and L. Wong. A data transformation system for biological data sources. In *Proceedings of 21st VLDB*, September 1995. Also Technical report MS-CIS-95-10, Dept. of Computer and Information Science, University of Pennsylvania. March 1995.

[6] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog and never dared to ask. *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[7] S. B. Davidson, A. S. Kosky, and B. Eckman. Facilitating transformations in a human genome project database. Technical Report MS-CIS-93-94/L&C 74, University of Pennsylvania, Philadelphia, PA 19104, December 1994.

[8] 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.

[9] 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.

[10] Anthony Kosky. *Transforming Databases with Recursive Data Structures*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, November 1995.

[11] R. J. Miller, Y. E. Ioannidis, and R Ramakrishnan. Schema equivalence in heterogeneous systems: Bridging theory and practice. *Information Systems*, 19, 1994.

[12] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

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

[14] P. Shoval and S. Zohn. Binary-relationship integration methodology. *Data and Knowledge Engineering*, 6:225–249, 1991.

[15] J. Thierry-Mieg and R. Durbin. Syntactic Definitions for the ACEDB Data Base Manager. Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, Cambridge,CB2 2QH, UK, 1992.