Solving Domain Mismatch and Schema Mismatch Problems with an Object-Oriented Database Programming Language

William Kent
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303-0971, USA
kent@hplabs.hp.com

Abstract

Domain mismatch and schema mismatch are two of the important semantic integration problems for interoperating heterogeneous databases. This paper structures the domain mismatch problem, illustrates approaches to its solution, and then extends this to the schema mismatch problem. Structuring of the problem and solution includes notions of conceptual territory, spheres, domain groups, localized and integrator functions, and type and function groups. Despite this structuring, the full generality of the problem still requires a rich language in which to describe the rules for reconciling discrepancies. Example solutions are illustrated in the Iris Programming Language (IPL) extension of Iris OSQL.

Keywords: heterogeneous databases, interoperability, semantic integration, domain mismatch, schema mismatch, database programming language, object orientation.

Contents

1 Introduction

Domain mismatch and schema mismatch are two of the important semantic integration problems for interoperating heterogeneous databases, as illustrated in [Ke, BL, DH]. Domain mismatch generally arises when several databases treat some common conceptual territory in different ways, the simplest example being differences in units of measurement. Schema mismatch is much the same thing at the schema level; an interesting "cross-over" problem arises when things in the data of one database correspond to things in the schema of another.

In this paper we structure the domain mismatch problem, illustrate approaches to its solution, and then extend this to the schema mismatch problem. Structuring of the problem and solution includes notions of conceptual territory, spheres, domain groups, localized and integrator functions, and type and function groups. Despite this structuring, the full generality of the problem still requires a rich language in which to describe the rules for reconciling discrepancies. Example solutions are illustrated in the Iris Programming Language (IPL) extension of Iris OSQL [An, Ly, F1, F2], with some further extensions being proposed as well.

To isolate these problems from other integration problems, we assume that the participating databases have been mapped into a single model, namely the Iris functional object model [AR]. We also avoid naming and identification problems, as well as other problems associated with the integration of heterogeneous databases. For the purpose of this paper, it hardly even matters whether things are in different databases; such semantic discrepancies could arise as well within a single database.

This work is part of the Pegasus project at HP Labs [AD, PP], which is prototyping an extension of Iris to integrate heterogeneous databases.

2 The Domain Mismatch Problem

2.1 Territories, Spheres, and Groups

The domain mismatch problem begins when some common conceptual territory is treated in different ways by different domains in different spheres. Spheres are usually different databases, but could also be subsets of the schema and data of one database, and might also span multiple databases. One sphere might be included in another. In Figure 1, the spheres might be databases in different countries for a multi-national corporation.

Conceptual Territories	Spheres			
	US	France	Germany	
money	m_1	m_2	m_3	←domain group
jobs	j_1	j_2	j_3	←domain group
colors	c_1	c_2	c_3	←domain group

Figure 1: Domains.

A *domain group* is a set of domains d_i which cover some conceptual territory. Each domain d_i is typically in a distinct sphere s_i. In Figure 1, the domains m_1, m_2, m_3 might be different currencies in which money is represented.

When the conceptual territory is a measured quantity, such as weight, the different domains in the group would simply be different representations expressed in different units; this mismatch is easily reconciled by arithmetic. The spheres might be different databases, or they might be different sets of measurements in the same database, e.g., the weights used for cars and the weights used for horses.

More complex discrepancies arise when the same conceptual territory is perceived as being populated, or partitioned, in different ways. The concept of "job" might be common to several spheres, yet each sphere has a different notion of what the specific jobs are. One sphere might have engineer, secretary, and salesman as jobs, while the jobs in another might include technician, designer, engineer, secretary, administrative assistant, and customer representative. The same thing might arise with the sets of skills one might possess, or with ethnic groupings, or with organizational units within different business entities (projects, departments, sections, labs, divisions, groups, operations, sectors, etc.).

Other examples might include different palettes of colors covering the same spectrum, different grading systems at different schools, different rating systems for restaurants (or for movies, or for hotels, etc.), terms in different languages for the same or similar concepts, different kinds of geographic units (counties vs. postal codes vs. voting districts vs. assessment districts, etc.). Another kind of mismatch arises if things are represented in one sphere as character strings but in another as persistent objects.

Many examples in this paper deal with employees receiving salaries in different currencies. Some examples try to reconcile a sphere in which jobs are represented as character strings ($JobSphere_1$) with another sphere in which they are represented as persistent objects ($JobSphere_2$). When we consider schema mismatch, we will introduce $JobSphere_3$ in which jobs occur as types. We will also use a stock market example, as in [KL], to illustrate schema mismatch.

2.2 Domain Mappings

The existence of different domains is not in itself a problem. So what if different databases represent money in different currencies? It only becomes a problem when there is a need to see them all in some integrated way. An important part of the solution is to be able to map between the domains in a group.

A mapping $\mu^D_{i,j}$ translates from elements of d_i to elements of d_j in a group D. The mappings for the group might be provided as a single *domain group mapping* $\mu^D(d_i, d_j, x)$ which takes as arguments a source and target domain, together with an element of the source domain; it returns a corresponding element of the target domain. It might simply invoke a corresponding direct mapping

$$\mu^D(d_i, d_j, x) = \mu^D_{i,j}(x),$$

or it might do the conversion directly, based on appropriate conversion factors.

2.3 Localized Functions and Integrators

Still, one doesn't simply look at currencies; what we want to integrate are some facts involving those currencies. The situation isn't interesting until the various spheres have some stock prices, or some employee salaries, or other such facts. In a functional model, such facts are represented as functions; the facts of interest correspond to a set of functions

$$f_i : t_i \rightarrow d_i$$

associated with the spheres s_i. These might, for example, be salaries of different sets of employees in a multi-national corporation, expressed in different currencies.

The existence of such functions and domains is still just a situation. It only becomes a problem when we want to see these facts in an integrated way, via an *integrator function*

$$f^* : t^* \rightarrow d^*$$

which might, for example, provide the salary of any employee, or the price of any stock.

Thus in most cases, the treatment of domain mismatch can be separated into two parts:

- Mappings between domains (corresponding to the mappings in [KL]).
- Integrator facilities which use such mappings (corresponding to the rules in [KL]).

Currency conversions represent mappings between different domains, i.e., different ways of representing the territory of money values. The mapping is independent of usage.

Integrator facilities depend on how the domains are being used. The paradigm for reconciling stock prices in different currencies may differ from the paradigm for reconciling salaries in different currencies, even though the same currency conversions are used. We might wish to see the average of stock prices but the sum of salaries.

Domain mappings aren't always independent of usage. We will also examine usage-dependent mappings.

In later sections we will examine the definition and maintenance of domain mappings in detail, and then we will examine the definition and update of integrator functions.

2.4 Identifying Domains

Domains often correspond to types in the various spheres. One would ideally hope to find the relevant domain specified as the result type[1] in a function signature, e.g.,

[1] "Domain" in the context of domain mismatch does *not* necessarily mean the domain of a function. A domain here can correspond either to the argument type or result type of a function.

Salary: Employee $\rightarrow$ *Dollars.*

Unfortunately, domains often aren't modeled as types. When literal subtypes can't be defined, the result types of many functions are typically given as literal data types, such as Real or Char, without identifying the unit of measure, currency, or other relevant domain. Currencies might simply be identified by character string names, in which case the domain group Currency might simply be an enumerated subtype of Char consisting of the names of the currencies.

The actual domain might not be recorded anywhere, or it might be specified in the schema (dictionary) as an auxiliary property of the function. It might occasionally be returned with the function result, in some self-describing format (e.g., a Salary function might return both a money value and a currency code). It might conceivably be deduced from the type or some other property of the argument, e.g., the nationality of the employee.

We can't manage domain mismatch without identifying the domains involved. We therefore postulate some function $\delta(f, x)$ which identifies the domain of the result returned by $f(x)$. It may or may not depend on the argument value x. As mentioned, in the ideal case $\delta(f, x)$ would simply return the result type from the signature of f, which might often be a literal subtype.

Whether or not a domain d_i is a type, we can model it as a predicate such that $d_i(x)$ is true if and only if x belongs to the domain. The populations of domains might be defined by various rules, in much the same way as derived types [Appendix A.2]. One might be defined as the current set of results for some function; e.g., in $JobSphere_1$ the domain of jobs might be whatever jobs people happen to be holding at the moment:

$$JobName ::= \{x \mid \exists y \;\; AsgJob(y) = x\},$$

or it might be defined as the set of jobs about which some data is maintained:

$$JobName ::= \{x \mid \exists y \;\; JobSpecs(x) = y\}.$$

This latter form might correspond to a domain defined as a primary key in a relational database.

Domains could be aggregate types, such as sets or tuples, but our examples only show atomic domains.

3 Schema Mismatch

Schema mismatch arises when similar concepts are expressed differently in the schema. A common byproduct is that data instances in one sphere correspond to schema elements in another. Depending on the model, the schema elements might be such thing as relations and attributes, entities and relationships, classes and methods, types and functions, etc. Our work will be expressed in terms of the types and functions in the Iris functional object model [F1, F2].

Many schema mismatch problems are really domain mismatch problems, except that some of the domains are in the schema instead of in the data. Jobs, for example, are often modeled as types, i.e., subtypes of Employee. Instead of $AsgJob(Sam) = Engineer$, we know that Sam is an engineer because he is an instance of the type, i.e., $Engineer(Sam)$ is true. We thus have a $JobSphere_3$ in which jobs are types. In this case the domain itself is a set of types, i.e., the JobType type group [Appendix A.3].

An example involving functions is adapted from the stock market examples of [KL]. We have a sphere $StockSphere_1$ containing a base stock market *Activity* function on three arguments

$$\textit{Activity: Company} \times \textit{Reading} \times \textit{Date} \rightarrow \textit{Price}$$

whose extension at the moment is shown in Figure 2.

Activity

Company	Reading	Date	Price
hp	close	1/3/91	50
hp	close	1/4/91	51
hp	high	1/3/91	52
hp	high	1/4/91	53
ibm	close	1/3/91	52
ibm	close	1/4/91	51
ibm	high	1/3/91	55
ibm	high	1/4/91	54

Figure 2: $StockSphere_1$.

Another sphere $StockSphere_2$ might maintain the same data in separate functions for each company, such as

$$\textit{HPActivity: Reading} \times \textit{Date} \rightarrow \textit{Price}$$
$$\textit{IBMActivity: Reading} \times \textit{Date} \rightarrow \textit{Price}$$
$$\vdots$$

The corresponding extensions of two of these functions are shown in Figure 3.

In $StockSphere_1$ the domain of interest is a set of Company instances in the data. In $StockSphere_2$ the corresponding domain is a set of functions, i.e., it is a function group *ACFuncs*.

HPActivity

Reading	Date	Price
close	1/3/91	50
close	1/4/91	51
high	1/3/91	52
high	1/4/91	53

IBMActivity

Reading	Date	Price
close	1/3/91	52
close	1/4/91	51
high	1/3/91	55
high	1/4/91	54

Figure 3: The *ACFuncs* function group in $StockSphere_2$.

4 The Nature of Domain Mappings

Mappings among domains may generally have a wide variety of characteristics [BL, DH]:

- Domain mappings could be multi-valued, e.g., a job in one company might correspond to a set of possible jobs in another company, or a color in one palette might correspond to several possible colors in another.
- Domain mappings might be usage-dependent, involving auxiliary rules inseparable from the integrator functions. Thus an 85 might be a B for undergraduate courses, but an A for graduate courses. The mapping for jobs might depend on other attributes of the job-holder, such as length of time in job, or education level.
- The mapping might be natural, like a units or currency conversion, or arbitrary, like mappings between jobs or colors, or the mapping from numeric grades to letter grades. It might be an arbitrary estimate, such as a mapping from letter grades into numeric: A→95, B→85, etc. Such estimates might be provided to facilitate statistical computations over large sets of students receiving both letter and number grades, even if there is some loss of accuracy.
- If the mapping is not 1:1, then it does not have a (single-valued) inverse. There is no natural inverse of the mapping from numeric grades to letter grades. If an arbitrary estimate is introduced to serve as an inverse, then identity may not be preserved in composition: a 90 might map to an A, then map back to a 95.
- Mappings might be provided only among existing domains, or a new domain might be introduced to serve as a common denominator. ECU (European Currency Units) is such a common denominator for national currencies. Or, different systems for grading restaurants (movies, etc.) might be arbitrarily mapped into "low", "medium", and "high".
- Domain mappings might be extended to yield auxiliary information besides a target domain value. The result might also include information about the source domain, or about the mapping process. Thus a conversion to dollars might yield the result $< 55.45, UK, 1.85 >$, i.e., a dollar value, the country of origin, and the conversion used.

That catalog of mapping characteristics illustrates the complexity of the domain mismatch problem, showing that a rather rich language is required for its solution. For our present purposes, we make the simplifying assumption that most useful domain mappings are usage-independent, and return simple single-valued results.

We should assume mappings are identities on a single domain: $\mu^D(d_i, d_i, x) = \mu^D_{i,i}(x) = x$.

The actual algorithms of domain mappings can be very rich and complex, involving various forms of computation and assertion, requiring a "computationally complete" language for their expression. (It could be procedural or declarative.) Following are a few examples in IPL.

Simple numeric conversion:

```
CREATE FUNCTION MapFoot2Inch(Number f) → Number AS
RETURN 12 * f;
```

Non-algorithmic conversion might be done by some form of conditional (case statement, rule, etc.):

```
CREATE FUNCTION MapColorsUS2French(Char us) → Char AS
RETURN
  IF us = 'red' THEN 'rouge';
  ELSE IF us = 'white' THEN 'blanc';
  ⋮
  ELSE 'unknown';
```

If the domains are large, or the mapping is frequently updated, it might be defined as a stored function

```
CREATE FUNCTION MapColorsUS2French(Char us) → Char fr AS STORED;
```

to be maintained by assertions such as

```
MapColorsUS2French('red') := 'rouge';
MapColorsUS2French('white') := 'blanc';
⋮
```

A domain group mapping for currencies, using a stored table of conversion rates:

```
CREATE FUNCTION ConvRate(Currency c1, Currency c2) → Number AS Stored;
ConvRate(US,UK) ::= 1.85;
⋮
```

```
CREATE FUNCTION
 MapCurrency(Currency c1, Currency c2, Number x) → Number AS
RETURN
  IF c1 = c2 THEN x
  ELSE x * ConvRate(c1,c2);
```

Note the use of the domain group Currency as a type in the signature.

5 Maintaining Mappings and Domains

The problem of maintaining mappings does not arise if the domains are fixed and the mappings are totally defined, e.g., by a computation on data values. This is the case for units conversions, or string mappings based on concatenation or similar operations.

A mapping $\mu^D_{i,j}$ might be a partial function, i.e., not defined for all values of the source domain d_i. For example, the mapping from letter grades to numeric may be defined as a mapping from character strings to integers, but only have defined values for five or six letters. The mapping would have to be adjusted if a new letter grade became meaningful.

The maintenance problem most often arises from changes in the domains d_i or d_j. Things might be added to d_i or removed from d_j; a result value of $\mu^D_{i,j}$ might no longer exist, or might no longer belong to d_j. Literal data types constitute fixed domains; their populations can't change. In general, though, the source and target domains d_i and d_j might each have variable populations. Restricted literal types, such as enumerated types, might be fixed or variable, depending on whether they are subject to re-definition. A domain defined by primary key values in a relational database is usually variable. Non-literal object types typically constitute variable domains, but they could sometimes be considered fixed (e.g., the set of Earth's planets).

When the population of a domain changes, it may affect mappings from and to this domain. When an element is added to d_k, it may be necessary to find or create corresponding elements in the other domains d_j in the group, and to adjust the mappings $\mu^D_{k,j}$. When an element is removed from d_k, it may be necessary to remove or destroy corresponding elements in the other domains d_i in the group, and to adjust the mappings $\mu^D_{i,k}$.

The general problems:

- When and how are such population changes detected, and the necessary adjustments initiated?
- How are the corresponding elements in other domains discovered or created? This is more complex if certain initializations are required.
- How are the mappings adjusted?

The need seems to arise in two contexts:

- When it is necessary to enumerate the elements of some target domain d_j, with the expectation that it include the images of all the other domains d_i. This is much the same problem as enumerating the instances of a derived type [Appendix A.2].

- When a mapping is invoked, e.g., when someone wants to see the jobs of all or certain people, as mapped into some target domain d_j. In this case, the adjustments could be triggered by a "mapping fault" when the mapping recognizes that it is not defined for some argument. This will be discussed in subsequent sections.

The easiest solution to implement puts the burden of responsibility on users, requiring them to manually maintain the domains and mappings by appropriately creating and deleting objects, and by modifying mapping rules or data. In this case, when a mapping encounters an unfamiliar value it simply returns an error. The complexity of the problem still requires this solution as a fall-back for the general case.

The following sections identify some of the problems involved, and illustrate algorithmic solutions for some of the simpler cases.

5.1 Mapping Faults

We illustrate the case when a mapping is invoked with an argument for which it has no defined result, and it is programmed to make the adjustment.

5.1.1 Creating Objects

Let's consider $JobSphere_1$ and $JobSphere_2$, in which jobs are represented as character strings and as persistent objects. In general, automatic object creation depends on being able to do all the necessary initialization. The correspondence here might simply be by name: the string in $JobSphere_1$ is the name of the object in $JobSphere_2$. When the mapping encounters a new string in $JobSphere_1$, it could automatically create a job in $JobSphere_2$ having that name:

```
CREATE FUNCTION MapName2Job(Char n) → Job AS
BEGIN
  VAR j;
  j := SELECT Job jj WHERE Name(jj)=n;
  IF IsNull(j) THEN
    j := CREATE Job;
    Name(j) := n;
  RETURN j;
END;
```

Note the risk of relying on properties such as names as the basis for a mapping. If users can change the names of job objects, they may become unreliable for mappings.

5.1.2 Creating Types

The schema mismatch examples can be handled similarly. Suppose the target sphere is $JobSphere_3$, which maintains jobs as types, e.g., as subtypes of Employee. The target domain for the mapping is the JobType type group described earlier. The appropriate action on encountering a new job name is to create a new type having that name:

```
CREATE FUNCTION MapName2JobType(Char n) → JobType AS
BEGIN
  VAR j;
  j := SELECT JobType jj WHERE Name(jj)=n;
  IF IsNull(j) THEN
    j := CREATE JobType;
    Name(j) := n;
  RETURN j;
END;
```

Note that the object being created is a type.

Without type groups, this mapping would have to be expressed in terms of Type rather than JobType. In that case it would really provide a mapping between any type and its name, whether or not the type corresponds to a job. It would return spurious results when invoked with the name of a type which is not a job type. The problem cannot be solved simply by limiting to subtypes of Employee, since there may be other subtypes such as Male, Female, Retired, Exempt, Temporary, PartTime, etc. which don't correspond to jobs.

5.1.3 Creating Functions

Schema mismatch involving functions is also similar. For the mapping $\mu^S_{1,2}$ between $StockSphere_1$ and $StockSphere_2$, we define the target domain in $StockSphere_2$ as the function group *AC-Funcs*. Whenever this mapping encounters a new company in $StockSphere_1$, it should create a new function in the group. This can be done automatically if all the initialization information is known. All functions in the group have the same signature, hence the argument and result types are known. The only thing missing is a name for the function. We will assume that to be provided by an arbitrary *MakeName* function, which might engage in a user dialog to get a name, or it might simply concatenate some predefined prefix or suffix.

If we don't assume any algorithmic correspondence between companies and functions, such as one based on naming patterns, then the correspondence has to be maintained as stored assertions:

```
CREATE FUNCTION Co2FuncData (Company c) → ACFuncs f AS STORED;
```

With that, we can define the actual mapping function as

```
CREATE FUNCTION MapCompany2ACFunc (Company c) → ACFuncs AS
BEGIN
  VAR f;
  f := Co2FuncData(c);
  IF IsNull(f) THEN
    fn := MakeName(c);
    f := CREATE ACFuncs fn(Reading,Date) → Price AS STORED;
    Co2FuncData(c) := f;
  RETURN f;
END;
```

Note the programmatic creation of functions within this procedure, with the function name provided in a variable.

5.1.4 Completing the Mappings

We have described how a mapping fault in $\mu^D_{i,j}$ might cause an adjustment to the mapping and the target domain d_j upon encountering an unfamiliar element of a source domain d_i. We haven't addressed the question of further adjustments in other domains d_k. How do we know how to map something from d_k into the new element of d_j? For example, when we created a new JobType in $JobSphere_3$, how do we know how to map things from other spheres into that new type?

We could wait until $\mu^D_{k,j}$ faults on an unfamiliar argument, and then see if it should map to the recently created element in d_j. But that raises a new question: how did we know to create a new element in d_j when $\mu^D_{i,j}$ faulted? Maybe we should have mapped to a pre-existing element in d_j.

This is an aspect of the identity problem, trying to determine whether things in different domains are "the same thing". Such identity problems are being investigated separately, and are not addressed in this paper.

5.2 Deletion

All we do here is describe the problem.

How do we know if deletion happens?

When do jobs disappear in the first sphere? Under what conditions does the disappearance of a job name in the first sphere require deletion of the corresponding job in the second? There could be a pileup of superfluous objects when the corresponding literals disappear. (This could be more serious in schema mismatch, when the superfluous objects might be types or functions.)

Under what conditions does deleting a job in the second sphere imply things should be changed in the first? What sort of change? Do people lose jobs? If we delete a job in the second sphere, and its name still occurs in the first, then the job might get re-created all over again in the second.

The solutions require local administrators to establish policy, which needs to be expressible in the database programming language.

6 Integrated Use

6.1 Integrator Functions

The localized functions $f_i : t_i \rightarrow d_i$ (Section 2.3) typically occur in distinct spheres s_i whose autonomies need to be respected, i.e., new types, functions, or other objects cannot be created in these spheres for the purpose of integration. Integration is thus done in a new *integrating sphere* s^*, which may or may not include the s_i as sub-spheres.

In terms of the salary example, the t_i would be sets of employees in different countries, and the d_i correspond to the different currencies. The domain mappings $\mu^D_{i,j}$ are currency conversions. Salaries can be seen in a uniform way by mapping them into a common currency d^* in s^*, which may or may not be one of the d_i. The localized functions f_i might not have the same name, e.g., "Salary", "Sal", "Wages", "Pay", "Earnings", or equivalents in other languages.

An integrating sphere might be *in the style of* one of the underlying spheres s_k, meaning that the forms and representations of functions, types, and objects in s^* are like those in s_k. In that case we sometimes denote the integrating sphere as s^{*k} to give us a hint of its style.

The integrator function $f^* : t^* \rightarrow d^*$ would be defined in the integrating sphere s^*, with

$$
\begin{aligned}
&t^* = t_1 \cup \ldots \cup t_n, \\
&f^*(x) ::= \textit{if}\; t_i(x) \;\textit{then}\; \mu^D(d_i, d^*, f_i(x)).
\end{aligned}
$$

$f^*(x)$ chooses and executes the appropriate localized function f_i, then maps its result from the domain d_i to the domain d^*. For now we make the simplifying assumption that $t_i \cap t_j = \phi$ for $i \neq j$, so that we don't have to worry about f_i and f_j yielding possibly different results for the same argument (to be relaxed later).

We define the type group $T = \{t_1, \ldots, t_n\}$ and the function group $F = \{f_1, \ldots, f_n\}$.

The steps in evaluating $f^*(x)$:

1. Determine the relevant type t_i of the argument x.
2. Pick the corresponding localized function f_i.
3. Evaluate $y_i = f_i(x)$.
4. Identify the source domain d_i.
5. Compute the final result by applying the domain group mapping: $f^*(x) = \mu^D(d_i, d^*, y_i)$.

Type groups (Appendix A.3) are useful for step 1. We are not interested in all the possible types to which x might belong, but only one of the localized types t_i in the group T. We can do step 1 as $t_i = Classify(x, T)$.

Step 2 requires choosing the localized function f_i defined on the argument type t_i. Function groups are useful here since we don't want just any function that happens to be defined on t_i. Step 2 consists of picking the f_i in F which is defined on t_i. This mechanism is at the heart of the integrator function, and will be discussed at length in subsequent sections. (The astute reader may observe a close resemblance to the resolution of overloaded functions.) Whatever the mechanism, we can characterize it as a "binding" function $\beta(f^*, t_i)$ which returns the corresponding $f_i \in F$.

$y_i = f_i(x)$ is readily evaluated for step 3. Step 4 is handled by the arbitrary function we defined in Section 2.4: $d_i = \delta(f_i, x)$. Step 5 is a straightforward application of the domain group mapping.

Combining all these steps, we can express the behavior of $f^*(x)$ as

$$f \leftarrow \beta(f^*, Classify(x, T)),$$
$$f^*(x) = \mu^D(\delta(f, x), d^*, f(x)).$$

The binding step, i.e., picking the localized function f_i in F corresponding to the localized type t_i, can be done

- Implicitly via overloading.
- Programmatically, e.g., via case statements or conditionals.
- Via explicit stored mappings, using function groups.

6.1.1 Binding Via Overloading

Integrators strongly resemble overloaded functions. In both cases, invocation of a function requires choosing from a set of other functions to be executed. Simple resolution (binding) of overloaded functions is based on the types of the arguments, which is just what we want here. In fact, this is a very simple form of overload resolution, since it is only based on single types, and there is no inheritance through intermediate types.

Let's first illustrate overloading applied to simple integration when there are no mismatches. Assume we have disjoint sets of employees AEmployee...ZEmployee, perhaps in different sectors of the company, and there is a *Salary* function on each, all returning salaries in US dollars. These existing Salary functions constitute a group of localized functions $f_A \dots f_Z$.

In order to access the salary of all employees, all we have to do is define a supertype spanning all the employees [DH], with a *Salary* function defined on it (Figure 4):

```
CREATE TYPE Employee SUPERTYPE OF AEmployee ... ZEmployee;
CREATE FUNCTION Salary(Employee) → Number AS 0;
```

```
              Employee
                 | Salary
                 |
          ----------------------
          |                    |
      AEmployee     ...    ZEmployee
           Salary               Salary
```

Figure 4:

The *Salary* function on Employee is defined to provide a default value, in case the Employee supertype is instantiable. Via late binding and overload resolution [Ly], an invocation of *Salary(x)* will bind to the *Salary* function defined on whichever subtype x belongs to. (We are still assuming these types are disjoint.)

If the salary functions don't all have the same name, overload resolution could be extended with a simple aliasing mechanism to allow them to behave as though they had the same name. Equivalently, the function group F could be explicitly defined as the set of functions to which $f^*(x)$ could be resolved.

Overloading can also be exploited in several cases when there are domain mismatches, i.e., different currencies:

- If localized functions can be installed to do the currency conversion. This reduces to the previous case.
- Currency can be automatically determined, either because it is explicitly returned by the localized functions or because it can be deduced from the type or country of the employee.

Assume now that the various *Salary* functions return results in local currencies, and we want to see them all in US dollars. If there is a *Country* function defined for each employee (possibly defined in terms of the subtype to which he belongs), then we might define a global salary function as

```
CREATE FUNCTION GSalary(Employee e) → Number AS
RETURN MapCurrency(Country(e),'US',Salary(e));
```

Here *Salary(e)* is again bound to the appropriate localized function by overloading. Its result is then converted by *MapCurrency*, based on the country of the employee. Note that *Country* is serving as the δ function for identifying the domain.

6.1.2 Binding By Cases

When overloading cannot be exploited, equivalent functionality can be explicitly specified in conditionals, case statements, or rules.

Suppose we didn't have appropriate name correspondences, and aliasing is not supported (Figure 5).

```
                Employee
                   | Salary
                   |
            ----------------------
            |                    |
        AEmployee     ...    ZEmployee
             ASalary              ZSalary
```

Figure 5:

Then the *Salary* function for all employees could be written as

```
CREATE FUNCTION Salary(Employee e) → Number AS
BEGIN
  RETURN
  IF AEmployee(e) THEN MapCurrency(Country(e),'US',ASalary(e));
  ELSE IF BEmployee(e) THEN MapCurrency(Country(e),'US',BSalary(e));
  ⋮
  ELSE IF ZEmployee(e) THEN MapCurrency(Country(e),'US',ZSalary(e));
  ELSE 0; /* or other default or error action */
END;
```

Note that compile-time type checking needs to observe the conditionals. For example, *ASalary* is applied to the variable e, which has only been declared as Employee, not AEmployee. Verifying that e will in fact be bound to an instance of AEmployee requires awareness of the logic flow. An alternative would be to defer to run-time type checking.

6.1.3 Binding Via Stored Mappings

The logic of overloading can also be simulated via stored mappings. Suppose we defined the type group and function group

$$EmpType = \{AEmployee, \ldots, ZEmployee\}$$
$$SalFunc = \{ASalary, \ldots, ZSalary\}$$

as follows:

```
CREATE TYPEGROUP EmpType;
ADD TYPE EmpType TO TypeNamed(AEmployee),...,TypeNamed(ZEmployee);
```

```
CREATE FUNCTIONGROUP SalFunc;
ADD TYPE SalFunc TO FuncNamed(ASalary),...,FuncNamed(ZSalary);
```

We can establish the mapping between EmpType and SalFunc in stored data:

```
CREATE FUNCTION ET2SF (EmpType) → SalFunc AS STORED;
ET2SF(TypeNamed('AEmployee')) := FuncNamed('ASalary');
⋮
ET2SF(TypeNamed('ZEmployee')) := FuncNamed('ZSalary');
```

Using the *Classify* function to determine the EmpType of an employee, we can define the *Salary* function on Employee as

```
CREATE FUNCTION Salary(Employee e) → Number AS
RETURN MapCurrency(Country(e),'US',ET2SF(Classify(e,EmpType))(e));
```

Let's do that again with the functions unnested, to see the logic:

```
CREATE FUNCTION Salary(Employee e) → Number AS
BEGIN
  VAR t,f,x,c;
  t := Classify(e,EmpType); /* an EmpType */
  f := ET2SF(t); /* a salary function */
  x := f(e); /* a salary in local currency */
  c := Country(e); /* the local currency */
  RETURN MapCurrency(c,'US',x); /* convert to US currency */
END;
```

Note the application of function variables.

6.2 Generalized Integration

Integrator functions have so far been relatively simple, mimicking the behavior of the underlying localized functions by simply picking one of them and converting its result to another domain.

Sometimes they need to be more elaborate, e.g., to reconcile mismatch within the argument type, to compensate for missing localized functions, to reconcile the results of several localized functions, to provide auxiliary information, or to incorporate usage-dependent domain mappings.

For a more general example, suppose we wanted to know the starting salaries of jobs in a multi-national corporation, where each sector in the corporation may use different concepts and representations of jobs, as well as different currencies. The sectors correspond to the underlying spheres s_i, and the corporation is the integrating sphere s^*.

Corresponding to each domain group $D = \{d_1, \ldots, d_n\}$ there is an *integrating domain* d^* in the sphere s^*. It is the target domain for the mappings $\mu_{i,*}^D$ to be used in integration. If the integrating domain is "in the style of" one of the d_k, e.g., it uses the same sort of representation, we will write the domain as d^{*_k}. In this case, we would typically expect $\mu_{k,*}^D$ to be the identity mapping, i.e., $\mu^D(d_k, d^{*_k}, x) = x$.

Let's say that each sphere s_i has a type (domain) Job_i, which together constitute the group JobGroup. Job_1 in sphere s_1 might be a set of job names occurring as primary keys; Job_2 in s_2 might be a set of persistent job objects. The integrating domain Job^* might be chosen to be in the form of persistent job objects; if we reuse the objects in Job_2, then the integrating domain might be written Job^{*_2}. There may be more job objects in the latter, corresponding to jobs existing in s_1 but not in s_2. There may or may not be a direct correspondence between jobs in s^* and the jobs in any s_i, i.e., the mapping may be very complex and arbitrary, as discussed earlier.

To get starting salaries in s^* we need a function

$$StartSal^*\colon Job^* \rightarrow d^*.$$

d^* is the common currency chosen for the corporation, e.g., US dollars.

For a given job in Job^*, the $StartSal^*$ function has to:

1. Get the starting salary from each sector that has such a job.
2. Convert that to the common currency.
3. Do something about results from multiple sectors.

Step 1 is difficult if there is not a good inverse mapping from the integrated type Job^* and the type Job_i in each sphere s_i. We will assume such a mapping exists.

Step 1 essentially amounts to having a function $StartSal_{i*}$ defined for each subtype Job_{i*}. A simple case is when sphere s_i has a $StartSal_i$ function, and there is a simple mapping μ_{*i}^J from Job^* to Job_i. It's simplest when the mapping is the identity mapping; then $StartSal_{i*}(x) = StartSal_i(x)$. This simple case corresponds to our first integration example, involving salaries of employees. We essentially assumed that the Employee type was already integrated, and that a salary function was available for each subtype.

If a $StartSal_i$ function is not available, $StartSal_{i*}$ has to be provided in some other way in s^*, either explicitly or implicitly. The mechanism might be to simply rename (alias) an existing function in s_i, or to provide a function which either supplies a default value or makes use of other functions available in s_i. Such functions might be explicitly defined in s^*, or they might be implicitly specified in the definition of $StartSal^*$ itself.

6.2.1 Missing Localized Functions

Sometimes the appropriate localized functions simply don't exist. Then it becomes necessary to invent them in the integrating sphere, either to provide default values or to reconstruct results from other information known in the localized spheres. This can either be done directly within the integrator function, or by simulating separate localized functions in the integrating sphere.

For example, there might not be a *Salary* function for ZEmployees, but rather the two functions *BasePay* and *OvertimePay* which could be added together to give a total salary. One approach would be to define a *Salary* function for ZEmployees (in the integrating sphere, if autonomy needs to be respected) as the sum of the other two. Then integration could proceed as before. Alternatively, it could be incorporated into the integrator function:

```
CREATE FUNCTION Salary(Employee e) → Number AS
BEGIN
  RETURN
  IF AEmployee(e) THEN MapCurrency(Country(e),'US',ASalary(e));
  ELSE IF BEmployee(e) THEN MapCurrency(Country(e),'US',BSalary(e));
  ⋮
  ELSE IF ZEmployee(e) THEN MapCurrency(Country(e),'US',BasePay(e)+OvertimePay(e));
  ELSE 0; /* or other default or error action */
END;
```

6.2.2 Integration of Overlapping Spheres

When the underlying populations are not disjoint ($t_i \cap t_j \neq \phi$), the results of several localized functions may need to be reconciled. This might arise, for example, when the same movie or restaurant is rated in several databases, or various sources of income for a given person are reported in different databases. The integrator function might be programmed to take various possible actions, such as

- Simply report all the results, identifying the sources.
- Select a "best" one, by some criterion.
- Merge the results, perhaps by summing or averaging, as appropriate.

While the resolution of this situation generally requires some arbitrary computation to be specified, two basic requirements can be identified:

- Aggregation of multiple results from the localized functions.
- Operations on such aggregates.

Suppose, for example, that an employee could belong to more than one EmpType, earning a salary in each. (We won't say how that is detected across multiple databases. Since we are not addressing identifier problems in this paper, we simply assume an employee has the same identifier everywhere.)

If we wished to simply report all such salaries, together with the type from which each salary comes, we might define a query over the possible types:

```
CREATE FUNCTION Salary(Employee e) → Set of <EmpType, Number> AS
SELECT t, MapCurrency(Country(e),'US',ET2SF(t)(e))
FOR EACH EmpType t WHERE t(e);
```

The query in that function iterates over the types t in EmpType (not over the instances of those types). If the employee e is an instance of t, the result will include a tuple $< n, s >$, where n is the name of the particular EmpType and s is the corresponding salary converted to US dollars.

Note the use of a type group in the signature and in the for-each clause.

If we wished to simply report the sum of salaries:

```
CREATE FUNCTION Salary(Employee e) → Number AS
RETURN BagSum(SELECT MapCurrency(Country(e),'US',ET2SF(t)(e))
              FOR EACH EmpType t WHERE t(e));
```

6.2.3 Auxiliary Results

Auxiliary information might indicate the underlying source of the information, e.g., the currency from which it was converted. Or it might provide some indication of the reliability of the conversion; mapping from letter grades back to numeric might be accompanied by a flag identifying it as an estimate. Auxiliary information was illustrated in the previous section, when the particular EmpType was returned along with the salary.

6.2.4 Usage-Dependent Domain Mappings

Sometimes mappings cannot be separated from the integrator functions. The mapping might depend on auxiliary rules involving the use of the source domain. Thus an 85 might be a B for undergraduate courses, but an A for graduate courses. Similarly, the mapping for jobs might depend on other attributes of the job-holder, such as length of time in job, or educational level. Then the mapping needs to know the job-holder (or information about him), not just the job being mapped. In such cases the domain mappings would be incorporated into the integrator functions.

6.3 Integrators For Schema Mismatch

6.3.1 With Types

In general, we can't integrate $StockSphere_3$ with the other two without knowing which types correspond to jobs. It might occasionally turn out that those are the only (user-defined) types in the schema, or the only types that employees might possess, or the only subtypes of Employee. Usually, however, we can't do the integration without type groups.

Once we have the JobType type group, we can assume the three localized functions

$$Job_1 : Emp_1 \rightarrow JobName,$$

$$Job_2 : Emp_2 \rightarrow Job,$$

$$Job_3 : Emp_3 \rightarrow JobType,$$
$$Job_3(e) ::= Classify(e, Jobtype).$$

in the three spheres (if not present, they could be added as part of the integration). Now we are back to the previous case of defining an integrator over three localized functions.

Creating types in the integrating sphere is potentially more complex than described in Section 5.1.2. Each created type is itself potentially an integrating domain [Section 6.2] with a complex definition.

Integrating the job spheres in the style of $JobSphere_3$ would require making each job a type. Introducing a new job such as "Designer" into $JobSphere^{*3}$ means introducing a definition of Designer itself as a derived type, being the image of designer jobs in all the spheres. Providing a rule to generate such definitions automatically could be difficult.

Other uses of types in the integrated sphere are similarly complicated by the fact that they are essentially derived types (Appendix A.2), with corresponding implications for type checking and queries.

6.3.2 With Functions

The relationship between $StockSphere_1$ and $StockSphere_2$ is expressed in the equivalence

$$Activity(c,r,d) = \mu^S_{1,2}(c)(r,d).$$

That is, the function obtained from the mapping $\mu^S_{1,2}(c)$ [Section 5.1.3] is in turn applied to (r,d) to yield the same result as $Activity(c,r,d)$. For example,

$$\mu^S_{1,2}(HP) = HPActivity;$$
$$Activity(HP,r,d) = HPActivity(r,d).$$

As in [KL], we might want to

1. Integrate in a sphere $StockSphere^{*1}$ in which all stocks are presented in the style of $StockSphere_1$, i.e., as arguments to a single $Activity^*$ function.

2. Integrate in a sphere $StockSphere^{*_2}$ in which all stocks are presented in the style of $StockSphere_2$, i.e., via an individual $XActivity^*$ function for each company X.

The existing functions in the underlying spheres will be labelled $Activity_1$ and $XActivity_2$.

For $StockSphere^{*_1}$, the integrator function has the form

$$Activity^*: Company^* \times Reading \times Date \rightarrow Price.$$

For simplicity, we assume that Reading, Date, and Price are uniform across the spheres. In this case, we are integrating mismatched arguments rather than results, and the general mechanisms of image domains and associated mappings as described in Section 6.2 apply here. We will need subtypes Company_1 and Company_2 corresponding to the stocks existing in the two spheres. As before, we start with the simplifying assumption that they are disjoint.

A key step is to recognize that the localized function $Activity_2$ does not exist. We can introduce it explicitly as a distinct function, or incorporate into the definition of $Activity^*$ as follows:

$$\begin{aligned}&Activity^*(c, r, d) ::= \\ &\quad \textit{if } Company_1(c) \textit{ then } Activity_1(c, r, d) \\ &\quad \textit{else } \mu^S_{*,2}(c)(r, d).\end{aligned}$$

The mapping $\mu^S_{*,2}$ is largely the same as $\mu^S_{1,2}$.

For $StockSphere^{*_2}$, integration is actually accomplished by a group of functions $ACFuncs^*$ corresponding to the group of functions $ACFuncs$ in $StockSphere_2$. There may be more functions in $ACFuncs^*$ than in $ACFuncs$. There is an integrator function for each company in either sphere, having a form such as

$$\begin{aligned}&HPActivity^*(r, d) ::= \\ &\quad \textit{if } Company_1(HP) \textit{ then } Activity_1(HP, r, d) \\ &\quad \textit{else } \mu^S_{*,2}(HP)(r, d).\end{aligned}$$

Again, note that creating new functions in the integrating sphere is potentially more complex than described in Section 5.1.3. Each created function is itself an integrating function with a complex definition of the form just shown.

6.4 Updating Integrator Functions

Update is a major aspect of the domain mismatch problem. In the integrated view, we not only want to see the salaries of employees, we may want to update them as well.

Update in a functional object model [F1, F2, Ly] is modeled as an assignment

$$f(x) := y$$

causing subsequent invocations of $f(x)$ to return y. A localized function could be directly updated by

$$f_i(x) := y_i$$

in which y_i is an element of d_i. This might be a direct update of the salary of an individual in his local currency.

One might wish to do such updates through an integrator function, e.g., express a salary update in US dollars and have it converted to the local currency. This might occur in a global update giving everyone in the company a 10% increase.

With $y^* \in d^*$, the update of an integrator function

$$f^*(x) := y^*$$

should be performed as

$$\textit{if } t_i(x) \textit{ then } f_i(x) := \mu^D(d^*, d_i, y^*),$$

i.e.,

1. Determine the type t_i of x (recall our disjointness assumption).
2. Apply the corresponding mapping $\mu^D_{*,i}$ (e.g., a currency conversion) to y^*, yielding an element y_i of d_i.
3. Find the corresponding localized function f_i.
4. Do the update.

Update through integrator functions is much the same as the view update problem. It can only be done automatically in limited cases, i.e., when the integrator function and the domain mapping are both simply invertible. Otherwise, as described in [KL], it is necessary to explicitly define the update algorithm to be associated with the integrator function, which could be done via "update entry points" (a proposed extension):

```
CREATE FUNCTION Salary(Employee e) → Number s AS
  RETURN MapCurrency(Country(e),'US',ET2SF(Classify(e,EmpType))(e));
ENTRY(:=):
  ET2SF(Classify(e,EmpType))(e) := MapCurrency('US',Country(e),s);
```

As before, we can explode this without nested functions to see the logic:

```
CREATE FUNCTION Salary(Employee e) → Number s AS
  BEGIN
    VAR t,f,x,c;
    t := Classify(e,EmpType); /* an EmpType */
    f := ET2SF(t); /* a salary function */
    x := f(e); /* a salary in local currency */
    c := Country(e); /* the local currency */
    RETURN MapCurrency(c,'US',x); /* convert to US currency */
  END;
ENTRY(:=):
  BEGIN
    VAR t,f,x,c;
    t := Classify(e,EmpType); /* an EmpType */
    f := ET2SF(t); /* a salary function */
    c := Country(e); /* the local currency */
    x := MapCurrency('US',c,s);/*convert to local currency*/
    f(e) := x; /* the update */
  END;
```

Note the function variable in the update.

7 Conclusions

Domain mismatch and schema mismatch are complex problems. They can best be understood by structuring the environment in terms of domain groups corresponding to conceptual territories, with different domains occurring in different spheres. Integration then occurs using integrating domains in an integrating sphere. The domain mismatch problem separates into two parts, the definition and maintenance of domain mappings, and the definition and update of integrator functions. Schema mismatch can in many cases be reduced to the domain mismatch problem by treating type groups and function groups as domains in themselves. The problems can generally be decomposed into a mapping aspect (corresponding to the mappings in [KL]) and an integrating aspect (corresponding to the rules in [KL]).

Although such analysis and decomposition is helpful, the solutions generally require sophisticated language capabilities. The role of a database programming language is to permit the solutions to be expressed and maintained with the database, rather than in application code.

Thus behavior specification is an essential contribution of object-orientation to the solution of the mismatch problem. Subtypes and supertypes are another essential feature for reconciling disparate domains. Overloaded operators are also useful. Object identity, on the other hand, seems to add problems: maintaining domain mappings can require explicit creation or deletion of persistent objects.

In a context which includes type systems, persistent objects, and non-trivial correspondences between domains, desirable language facilities include:

- Arbitrary computational power: conditionals, iteration, and probably even recursion (though we haven't actively looked for examples requiring recursion), as well as aggregate types and operations.
- Type and function groups, including disjointness and covering specifications.
- Uniform treatment of system and user objects:
  - Uniform syntax/semantics for creating user- and system-type objects.
  - Variables and expressions allowed wherever system objects can occur, i.e., functions and types. This would include their occurrence in declarations and queries.
  - DDL within procedures, e.g., dynamic creation of types and functions, with parameterized arguments.
  - User subtypes of system types (e.g., type and function groups).
  - Creation of user-defined supertypes as well as subtypes.
- Update entry points.
- Extended overloading, via aliasing and compatible result types.
- Derived types.
- Subtypes of literals, including
  - Dimensioned types (units).
  - Enumerated types.
  - Length-constrained types.

OSQL and IPL are still evolving, and we have not completed our analysis of the extent to which they currently support these requirements. This will be continued in the Pegasus project.

8 Acknowledgments

Thanks go to Jurgen Annevelink for developing IPL, Ravi Krishnamurthy for identifying key language requirements, and to Abbas Rafii and the other Pegasus project teammates for discussion of integration concepts and examples. Jurgen and Ravi also provided valuable feedback on prior drafts.

9 References

[AD] R. Ahmed, P. DeSmedt, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, M.-C. Shan, "Pegasus: A System for Seamless Integration of Heterogeneous Information Sources", Proc. IEEE COMPCON, March 1991, San Francisco, Calif.

[AR] Rafi Ahmed and Abbas Rafii, "Relational Schema Mapping and Query Translation in Pegasus", Workshop on Multidatabases and Semantic Interoperability, Nov. 2-4, 1990, Tulsa OK. Also HPL-DTD-90-4, Hewlett-Packard Laboratories, Oct. 9, 1990.

[An] Jurgen Annevelink, "Database Programming Languages: A Functional Approach", HPL-DTD-90-12, Hewlett-Packard Laboratories, Nov. 30, 1990.

[BL] C. Batini, M. Lenzerini, and S.B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration", ACM Computing Surveys 18(4), Dec. 1986.

[DH] Umeshwar Dayal and Hai-Yann Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System", IEE Trans. Software Engrg, Vol SE-10 No 6, Nov. 1984.

[F1] D.H. Fishman et al, "Iris: An Object-Oriented Database Management System", ACM Transactions on Office Information Systems, 5(1), January 1987. Also in *Readings in Object-Oriented Database Systems,* Zdonik and Maier, editors, Morgan Kaufmann, San Mateo, California, 1989.

[F2] D.H. Fishman, et al, "Overview of the Iris DBMS", *Object-Oriented Concepts, Databases, and Applications,* Kim and Lochovsky, editors, Addison-Wesley, 1989.

[Ke] William Kent, "The Many Forms of a Single Fact", Proc. IEEE COMPCON, Feb. 27-Mar. 3, 1989, San Francisco. Also HPL-SAL-88-8, Hewlett-Packard Laboratories, Oct. 21, 1988.

[KL] Ravi Krishnamurthy, Witold Litwin and William Kent, "Language Features for Interoperability of Databases with Schematic Discrepancies", HPL-DTD-90-14, Hewlett-Packard Laboratories, Dec. 17, 1990. [Submitted for publication.]

[Ly] Peter Lyngbaek, "OSQL: A Language for Object Databases", HPL-DTD-91-14, Hewlett-Packard Laboratories, Dec. 17, 1990.

[PP] Pegasus Project, "Pegasus: An Interoperable Heterogeneous Information Management System", [Submitted for publication.]

A Appendix: Type Groups and Function Groups

A.1 Types

We model a type t as a predicate function $t(x)$ which is true if and only if x is an instance of t. More generally, for any predicate and its corresponding extension, we let p denote the set and $p(x)$ the corresponding predicate expression, such that $p(x) \Leftrightarrow x \in p$. Thus $[p(x) = q(x) \vee r(x)] \Leftrightarrow [p = q \cup r]$. This notation applies to types and groups.

A.2 Derived Types

Like any function, predicate functions which serve as types can have their values established by assertion (stored data) or by derivation rules. The type of an object can be asserted when it is created (e.g., as a person) or later during its lifetime (e.g., when it becomes an employee).

Special derivation rules apply to types via subtype relationships: every employee is a person.

Derived types (a proposed extension to OSQL) could be defined in much the same way as derived functions:

```
CREATE TYPE Senior SUBTYPE OF Person x AS
Age(x) > 65;
```

Much like view maintenance, derived types can be supported in a backward-chaining or forward-chaining fashion. Backward chaining means the derivation is evaluated whenever the type is referenced (e.g., in queries), which could be inefficient for complex derivations. Forward chaining means that every addition or deletion has to be detected (e.g., whenever a person is created or destroyed, or changes age) and propagated into the extension of the type. This would require some sort of monitor or trigger facility.

A.3 Type Groups

A *type group* is a type whose instances are types, i.e., a subtype of Type in the Iris model. It is an auxiliary concept being proposed as an extension to OSQL/IPL.

Type groups are useful in the signatures of functions which have types as their arguments or results, when they are constrained to accept or return restricted sets of types. For example, when jobs such as Engineer and Programmer are types, we may want to define a mapping which returns one of these job types, but not any other type. We also might want a query to range over just this set of types.

TypeGroup (the type whose instances are type groups) would itself be a type group, and so would Type. A type group such as JobType might be definable in OSQL by either of the following:

```
CREATE TYPEGROUP JobType;
```

```
CREATE TYPE JobType SUBTYPE OF Type;
```

The following three could then be equivalent:

```
CREATE TYPE Engineer;
ADD TYPE JobType TO Engineer;

x := CREATE JobType;
Name(x) := 'Engineer';

CREATE JobType 'Engineer';
```

Classify(x,T) classifies an object x with respect to the types in a group T, returning the types in T of which x is an instance:

$$Classify\!: Object \times TypeGroup \rightarrow Type,$$
$$Classify(x, T) = \{t \mid T(t) \wedge t(x)\},$$

or, in OSQL,

```
CREATE FUNCTION Classify(Object x, TypeGroup g) → Set of Type AS
SELECT Type t WHERE g(t) AND t(x);
```

Notice how types are being applied as variable functions (predicates) via g and t.

If the types in T are disjoint, then *Classify(x,T)* is single-valued. For example, $Classify(x, JobType)$ returns the instance(s) of JobType of which x is an instance.

Type groups have several potential uses:

- They can themselves serve as domains when dealing with schema mismatch.
- Specification that the types in a type group are disjoint, or that they cover or partition some other type. For example, JobType usually partitions Employee.

Type groups are closely related to parameterized types. A "parameterized type" is typically not really a type itself, but a mapping into a set of types. A type group corresponds to the set into which this mapping maps.

A.4 Covered Types

A *covered type* (also a proposed OSQL extension) is the union of types in a type group:

```
CREATE TYPE t COVERED BY T;
```

The type Employee could be defined as being covered by the group JobType.

Properties:

- Every instance t_i in the group T is a subtype of t.
- Any type added to the group T becomes a subtype of t.
- The group T *covers* t, i.e., every instance of t is an instance of at least one $t_i \in T$. It follows that t has no "immediate" instances, i.e., it is an abstract type, not instantiable. Objects cannot be created as instances of t, or made instances of t (unless simultaneously made an instance of some t_i). A type t_i cannot be removed from an object unless the object retains or is immediately given another type $t_j \in T$.
- If the group T is disjoint, then it *partitions* t, i.e., every instance of t is an instance of exactly one $t_i \in T$.

Enumerating the instances of a type t covered by a group T is logically equivalent to the OSQL-like query

```
SELECT UNIQUE x FROM Object x, Type ti WHERE T(ti) AND ti(x);
```

A covered type is effectively a derived type, i.e., one having a membership condition derived by some rule. Enumerating the instances of a covered type could be as difficult as enumerating the instances of a derived type, especially when the types in the covering group are themselves derived.

Type groups provide a useful way to organize the subtypes of a type. A type can be covered or partitioned by several independent type groups, corresponding to different ways of subdividing the type. For example, employees might also be partitioned by the two subtypes Male and Female, which might constitute a Gender type group. Employees may also be divided into subtypes A,B,C,...by organizational criteria, e.g., by division or country. These subtypes would constitute yet another type group. (It would be interesting to explore combinations of type groups, yielding groups of intersections of types, e.g., MaleEngineers.)

A.5 Function Groups

A *function group* is analogous to a type group, being a type whose instances are functions, hence a subtype of Function. Useful type groups typically have the same or similar signatures, related via type groups.

If types are modeled as predicate functions, then type groups turn out to be a special case of function groups.