A Method for Integrating Deductive Databases

Lihui Xu and Alexandra Poulovassilis

Department of Computer Science
King's College London
Strand, London WC2R 2LS
{lihui,alex}@dcs.kcl.ac.uk

Abstract. This paper presents an approach for integrating deductive databases. In our approach deductive databases are expressed in a functional database programming language. For integrating the extensional parts of deductive databases, we use a binary relational ER model with subtyping as the common data model and propose a semi-automatic method to perform the integration. For integrating the intentional parts of deductive databases, we formally define identity and containment relationships between derived functions and propose a systematic method for comparing their semantics and integrating them.

1 Introduction

This paper addresses the integration of deductive databases. Deductive databases couple a conventional database (the *extensional database*) with a knowledge base (the *intentional database*). The extensional database is described by a database schema expressed using a particular data model. Rules are commonly expressed as formulae in a logic language such as Datalog [20]. However, following recent work in functional database languages [7, 17, 8], we assume that rules are expressed as equations in a functional language. This language supports sets as first class objects and subsumes Datalog in the sense that any set of Datalog rules can be translated into a set of equations with the same semantics; in fact the language is more expressive than Datalog since set-manipulation functions such as nesting, counting and unnesting can be defined [8]. Integrity constraints are also more expressive than those of logic languages since cardinality and other aggregation constraints can be expressed [11].

In this paper we consider the integration of both the extensional and the intentional parts of a set of component deductive databases. The common data model (CDM) that we use for integrating the extensional parts is a binary relational ER model with subtyping. This model has fewer constructs than general ER models supporting higher-degree relationships, thus simplifying the integration process. There is no loss of expressiveness since an n-ary relationship can be represented as an entity type and n binary relationships.

The contribution of the paper is two-fold. Firstly, we propose a semi-automatic method for integrating the component schemas where the DBA needs only declare the relationships between schema constructs and need not specify how the global schema is derived. Secondly, we propose a systematic method of comparing the semantics of rules and integrating them accordingly.

The outline of the paper is as follows. In Section 2 we give an overview of the functional database programming language (DBPL) that we use to define the component and integrated databases, covering only aspects of the language that are prerequisites for this paper; more comprehensive accounts can be found in [17, 7, 8, 11, 4]. In Section 3 we present our deductive database integration method. In Section 4 we compare our method with other related methods; for reasons of space this comparison is brief and a fuller discussion may be found in [21]. Finally, in Section 5 we give our conclusions and indicate directions of further work.

2 The Language

2.1 Types

Our functional DBPL is strongly, statically typed and we use the notation e_1, ..., $e_n :: t$ to indicate that the expressions $e_1, \ldots, e_n$ all have type t. A number of primitive types are supported, such as *Bool*, *Str*, *Num* and *Chr*. Also supported are *function types*, of the form $t_1 \rightarrow t_2$ where t_1, t_2 are arbitrary types, and *structured types* such as polymorphic product, list and set types. In particular, $[t]$ is a list type and $\{t\}$ a set type for any type t, and $(t_1, \ldots, t_n)$ is an n-product type for $n > 1$ and any types $t_1, \ldots, t_n$, The user can declare enumerated types and new constants of such a type. For example, the following declarations declare two enumerated types and two constants of each type:

$$
\begin{array}{ll}
\_type & Person; \\
\_type & Male; \\
\_value & P1, P2 \ :: Person; \\
\_value & M1, M2 :: Male;
\end{array}
$$

Enumerated types play the role of entity types in our CDM and henceforth we use the terms 'enumerated type' and 'entity type' interchangeably. For every enumerated type, T, a system-defined 0-ary function *allT* returns the current extent of T in the form of a set. For example, after the above declarations $allPerson = \{P1, P2\}$ and $allMale = \{M1, M2\}$. We term the *allT* functions *type-extent functions*.

The user can also declare type synonyms, for example:

$$
\begin{array}{l}
\_type\ Name == Str; \\
\_type\ Date \ \ == (Num, Num, Num);
\end{array}
$$

Subtype relationships between entity types can be specified. For example, the declaration

$$\_type\ Male < Person;$$

causes the extent of *Person* to expand to include the extent of *Male*, so that we now have $allPerson = \{P1, P2, M1, M2\}$ and $allMale = \{M1, M2\}$. We refer the reader to [4] for further details of subtyping in our DBPL and of the type inference algorithm used.

2.2 Data functions

Factual information is stored within updatable, set-valued functions termed *data functions* which correspond to the extensional relations of a logic database. For example, the following data function stores persons and their names:

$$\_datafun\ name :: \{(Person, Name)\};$$

Since we have adopted a binary relational CDM, henceforth we assume that all data functions have a type which is of the form $\{(t_1, t_2)\}$ where each of t_1 and t_2 is either an entity type or a type synonym.

2.3 Intentional functions

These consist of one or more equations which are specified by the user. For example, the following commands specify two equations that define the higher-order $foldr$ function and one equation that defines the sum function in terms of $foldr$ (the notation $(x : xs)$ denotes a list with head x and tail xs):

$$\begin{aligned}
&_define\ foldr\ op\ e\ [\,] &&= e;\\
&_define\ foldr\ op\ e\ (x : xs) &&= op\ x\ (foldr\ op\ exs);\\
&_define\ sum\ xs &&= foldr\ (+)\ 0\ xs
\end{aligned}$$

2.4 Derived functions

Derived functions are a class of intentional functions that correspond to, and generalise, the intentional predicates of logic databases. They also generalise the derived functions of the functional data model [15] in that they are associative as opposed to uni-directional. In order to formally define derived functions, we need to introduce the notion of a *comprehension*, $\{h \mid Q_1;\ \ldots;\ Q_m\}$, where h is an expression, and Q_1 to Q_m are *qualifiers*. Each qualifier is either a boolean-valued expression, termed a *filter*, or it is a *generator* of form $p \leftarrow s$, where p is a pattern (i.e. an expression without functions) and s is a set-valued expression. Overall, a comprehension returns a set of type $\{t\}$, where t is the type of h.

Definition 1. A *derived function*, f, is one that is defined by an equation of one of the following two forms:

$$\begin{aligned}
&f = g_1 \cup \ldots \cup g_n;\\
&f = \{h \mid p_1 \leftarrow g_1;\ \ldots\ ; p_n \leftarrow g_n;\ Q\};
\end{aligned}$$

where $n \geq 1$, Q is a, possibly empty, sequence of filters, and each g_i is either the identifier f, or a data function, or another derived function. For the purposes of this paper we assume that derived functions have a type of the form $\{(t_1, \ldots, t_n)\}$ for any $n \geq 1$, where for $n = 1$ the type (t_1) is syntactically equivalent to the type t_1.

Derived functions may be recursively defined. For example, *ancs*, which returns the set of pairs (x, y) such that x is an ancestor of y:

$$\begin{aligned}
&_derived\ ancs\ = parents \cup ancs' \\
&_derived\ ancs' = \{(x, z) \mid (x, w_1) \leftarrow parents;\ (w_2, z) \leftarrow ancs;\ w_1 == w_2\}
\end{aligned}$$

We refer the reader to [10] for a more detailed account of derived functions in our language, including a discussion of evaluation and optimisation issues.

2.5 Integrity Constraints

An integrity constraint is a derived function that must always evaluate to the empty set e.g. $uniqueName$ ensures that a person has only one name:

$$\begin{aligned}
_constraint\ uniqueName = \{(x, n, m) \mid\ & (x, n) \leftarrow name;\ (y, m) \leftarrow name; \\
& x == y;\ n\ != \ m\}
\end{aligned}$$

Without loss of generality, we assume that no integrity constraint calls any other integrity constraint, including itself. This has no impact on the expressiveness of integrity constraints but simplifies the database integration process somewhat.

3 The Database Integration Strategy

In Section 3.1 we define the notion of a *component database* and its *stratification*, and discuss how constructs in the component databases are mapped to those of the integrated database. We present our database integration strategy in Sections 3.2 - 3.5.

3.1 Component Databases and Mappings

Following our discussion in Section 2, each component database, and also the integrated database, satisfies the following definition:

Definition 2. A *database* is a quintuple of sets, $\langle Schema, EDB, IDB, CDB, PDB\rangle$, where:

- *Schema* (the schema) consists of the type declarations and the subtype relationships between entity types;
- *EDB* (the extensional database) consists of the data functions and the type-extent functions;
- *IDB* (the intentional database) consists of the derived functions;
- *CDB* (the constraint database) consists of the integrity constraints;
- *PDB* (the procedural database) consists of all other intentional functions.

The members of the sets $Schema$, EDB, IDB, CDB and PDB are termed *database constructs*.

For the purposes of our database integration strategy we impose a *stratification* on the data functions, derived functions and integrity constraints of each component database, i.e. on the set $EDB \cup IDB \cup CDB$, which reflects the dependencies between function definitions. Since we have made the assumption that no integrity constraint calls any other integrity constraint, henceforth we assume that all stratifications are of the form

$$EDB, IDB^1, \ldots, IDB^s, CDB^1, \ldots, CDB^t$$

where $IDB^1 \cup \ldots \cup IDB^s$ is a partition of IDB, $CDB^1 \cup \ldots \cup CDB^t$ is a partition of CDB, and each CDB^i contains a single integrity constraint.

Given n component databases, $D_1, \ldots, D_n$, where $D_i = \langle Schema_i, EDB_i, IDB_i, CDB_i, PDB_i \rangle$ we integrate them into a single database:

$$D = \langle Schema, EDB, IDB, CDB, PDB \rangle$$

In doing so, we need to establish mappings between each D_i and D. The *schema-level mappings* define the correspondence between the database constructs of each D_i and those of D while the *instance-level mappings* define the actual values of the database constructs of D in terms of those of $D_1, \ldots, D_n$. From now on we use the notation $D_i.x$ to denote a database construct x of a component database D_i, and $D.x$ to denote a database construct x of the integrated database D.

Definition 3. The *schema-level mapping*, M_i, between D_i and D maps each database construct in D_i to one in D i.e. $M_i : D_i \longrightarrow D$. In particular M_i maps database constructs in $Schema_i$ to those in $Schema$, database constructs in EDB_i to those in EDB, and so forth.

Definition 4. For each type t in D_i there is an *instance-level mapping* $\tau_{D_i.t}$ which maps values of type t in D_i to values in D as follows:

(i) If $D_i.t$ is an entity type or type synonym, then $\tau_{D_i.t}$ maps values of type $D_i.t$ to values of type $M_i(D_i.t)$ according to some semantics specified by the DBA.
(ii) If $D_i.t$ is a primitive type, then $\tau_{D_i.t}$ maps values of type $D_i.t$ to themselves.
(iii) Given values $e_1 :: t_1, \ldots, e_k :: t_k$ in D_i, then $\tau_{D_i.(t_1,\ldots,t_k)}$ maps tuples $(e_1, \ldots, e_k)$ to tuples $(\tau_{D_i.t_1}(e_1), \ldots, \tau_{D_i.t_k}(e_k))$.
(iv) Given values $e_1, \ldots, e_k :: t$ in D_i, then $\tau_{D_i.[t]}$ maps lists $[e_1, \ldots, e_k]$ to lists $[\tau_{D_i.t}(e_1), \ldots, \tau_{D_i.t}(e_k)]$ and $\tau_{D_i.\{t\}}$ maps sets $\{e_1, \ldots, e_k\}$ to sets $\{\tau_{D_i.t}(e_1), \ldots, \tau_{D_i.t}(e_k)\}$.

3.2 Schema and EDB Integration

Given a database $D = \langle Schema, EDB, IDB, CDB, PDB \rangle$, we can represent its $Schema$ and EDB by means of a directed graph, G, such that each entity type or type synonym $T \in Schema$ is represented by a node labelled T, each subtype relationship $T_i < T_j$ is represented by an *inc-edge*, $T_i \Longrightarrow T_j$, from T_i to T_j and each data function $f :: \{(T_i, T_j)\}$ is represented by a *rel-edge*, $T_i \longrightarrow T_j$, labelled f from T_i to T_j. We call the subgraph of G induced by the rel-edges its *rel-subgraph* and the subgraph induced by the inc-edges its *inc-subgraph*.

From now we use G_i and G to represent the graphs of a component database D_i and the integrated database D, respectively. The graph representations of two example databases are given in Figures 1 and 2. $ICTRAIN$ stores information about inter-city passenger trains and $LOCTRAIN$ information about local passenger trains (local trains call at both cities and towns while inter-city trains only travel between cities).

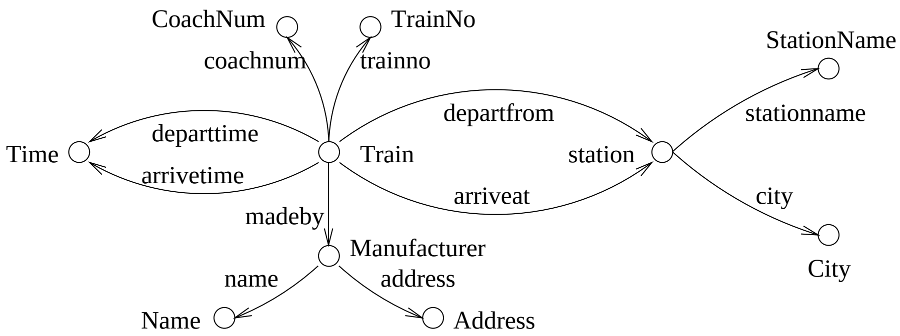

Fig. 1. Graph representation of $ICTRAIN$ database

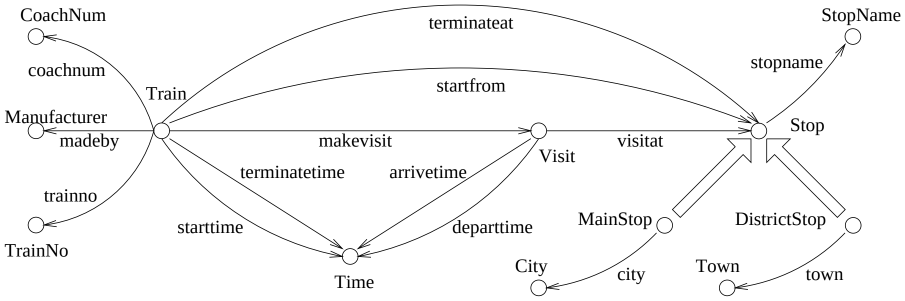

Fig. 2. Graph representation of $LOCTRAIN$ database

Our $Schema$ and EDB integration method relies on the above graph representation of the component schemas. In particular, when integrating $Schema_i$ and EDB_i for $i = 1 \ldots n$, we first integrate their nodes, then their inc-subgraphs, and finally their rel-subgraphs:

(a) Integrating Nodes. In order to integrate the nodes of $G_1, \ldots G_n$, the DBA has to provide correspondence statements between them. There are four kinds of correspondence statements:

Definition 5. Given two types $D_i.T_i$ in $Schema_i$ and $D_j.T_j$ in $Schema_j$, a correspondence statement between them can be one of the following: *$D_i.T_i$ is equivalent to $D_j.T_j$, $D_i.T_i$ is contained in $D_j.T_j$, $D_i.T_i$ is overlapping with $D_j.T_j$*, or *$D_i.T_i$ is disjoint with $D_j.T_j$*.

For example, in order to integrate the example databases $ICTRAIN$ and $LOCTRAIN$ into one database, $TRAIN$, the following correspondence statements may be specified:

- $ICTRAIN.Train$ is equivalent to $LOCTRAIN.Train$
- $ICTRAIN.CoachNum$ is equivalent to $LOCTRAIN.CoachNum$
- $ICTRAIN.TrainNo$ is equivalent to $LOCTRAIN.TrainNo$
- $ICTRAIN.Name$ is equivalent to $LOCTRAIN.Manufacturer$
- $ICTRAIN.Time$ is equivalent to $LOCTRAIN.Time$
- $ICTRAIN.StationName$ is contained in $LOCTRAIN.StopName$
- $ICTRAIN.Station$ is contained in $LOCTRAIN.Stop$
- $ICTRAIN.Station$ is equivalent to $LOCTRAIN.MainStop$
- $ICTRAIN.city$ is equivalent to $LOCTRAIN.city$

We note that the correspondence statement *is-equivalent-to* defines an equivalence relation. The entity types and type synonyms in the integrated $Schema$ are constructed from the equivalence classes of this relation; the names of these types are chosen by the DBA. If an equivalence class consists of the types $D_{i_1}.T_{i_1}, \ldots, D_{i_k}.T_{i_k}$ and its corresponding type in $Schema$ is $D.T_i$, then the schema-level mappings obtained from this correspondence are:

$$M_{i_j}(D_{i_j}.T_{i_j}) = D.T_i, \text{ for } j = 1 \ldots k$$

The instance-level mappings $\tau_{D_{i_j}.T_{i_j}}$ for $j = 1 \ldots k$ need to be specified by the DBA. The extent of $D.T_i$ is then $\bigcup_{j=1}^{k} \{\tau_{D_{i_j}.T_{i_j}}(x) \mid x \in D_{i_j}.allT_{i_j}\}$.

In some cases, for instance if two integrated entity types $D.T_i$ and $D.T_j$ are disjoint or overlapping, the DBA may decide to define a common supertype for them. If so, the DBA can create a supertype $D.T$ of $D.D_i$ and $D.T_j$ such that $D.allT = D.allT_i \cup D.allT_j$. Likewise, if $D.T_i$ and $D.T_j$ are overlapping, the DBA can create a subtype $D.T$ of $D.T_i$ and $D.T_j$ such that $D.allT = D.allT_i \cap D.allT_j$.

After node integration, the entity types and type synonyms in the integrated database $TRAIN$ are as shown in Figure 3.

(b) Integrating Inc-Subgraphs. For any two types $D.T_1$ and $D.T_2$ in $Schema$, if one of the following conditions is satisfied we add an inc-edge from $D.T_1$ to $D.T_2$ into the inc-subgraph of G i.e. add $D.T_1 < D.T_2$ to $Schema$:

(i) $D.T_2$ is an entity type created by the DBA and it is a supertype of $D.T_1$, or $D.T_1$ is an entity type created by the DBA and it is a subtype of $D.T_2$.

(ii) $D.T_1$ and $D.T_2$ are derived from two equivalence classes, C_1 and C_2, and there exist $D_i.T_{i_1} \in C_1$, $D_i.T_{i_2} \in C_2$, and an inc-edge $D_i.T_{i_1} \Longrightarrow D_i.T_{i_2}$.
(iii) $D.T_1$ and $D.T_2$ are derived from two equivalence classes, C_1 and C_2, and there exists $D_i.T_{i_1} \in C_1$ and $D_j.T_{j_2} \in C_2$ $(i \neq j)$ such that the DBA has specified the correspondence statement "$D_i.T_{i_1}$ is contained in $D_j.T_{j_2}$".

(c) Integrating Rel-Subgraphs. Integrating the rel-edges of the G_i also needs the DBA to provide correspondence statements between the data functions that form their labels.

Definition 6. Given data functions $D_i.f_i$ :: $\{(D_i.T_{i_1}, D_i.T_{i_2})\}$ in EDB_i and $D_j.f_j$ :: $\{(D_j.T_{j_1}, D_j.T_{j_2})\}$ in EDB_j, and a path of m data functions from $D_j.T_{k_1}$ to $D_j.T_{k_{m+1}}$ in EDB_j, $D_j.f_{k_1}$ :: $\{(D_j.T_{k_1}, D_j.T_{k_2})\}$, ..., $D_j.f_{k_m}$ :: $\{(D_j.T_{k_m}, D_j.T_{k_{m+1}})\}$, where $m \geq 1$, there are four kinds of correspondence statements:

(i) $D_i.f_i$ *is equivalent to* $D_j.f_j$,
(ii) $D_i.f_i$ *is contained in* $D_j.f_j$,
(iii) $D_i.f_i$ *is generalizable with* $D_j.f_j$,
(iv) $D_i.f_i$ *is mergeable with* the path $D_j.f_{k_1}$, ..., $D_j.f_{k_m}$.

Some prerequisites exist for the correspondences between data functions, which are as follows for each of the above cases respectively:

(i) $D_i.T_{i_1}$ is equivalent to $D_j.T_{j_1}$ and $D_i.T_{i_2}$ is equivalent to $D_j.T_{j_2}$.
(ii) $D_i.T_{i_1}$ is equivalent to or contained in $D_j.T_{j_1}$ and
$D_i.T_{i_2}$ is equivalent to or contained in $D_j.T_{j_2}$.
(iii) (a) $D_i.T_{i_1}$ is disjoint with $D_j.T_{j_1}$ and
$D_i.T_{i_2}$ is equivalent to or contained in $D_j.T_{j_2}$. Or
(b) $D_i.T_{i_1}$ is equivalent to or contained in $D_j.T_{j_1}$ and
$D_i.T_{i_2}$ is disjoint with $D_j.T_{j_2}$. Or
(c) $D_i.T_{i_1}$ is disjoint with $D_j.T_{j_1}$ and $D_i.T_{i_2}$ is disjoint with $D_j.T_{j_2}$.
(iv) $D_i.T_{i_1}$ is equivalent to or contained in $D_j.T_{k_1}$ and
$D_i.T_{i_2}$ is equivalent to or contained in $D_j.T_{k_{m+1}}$, or vice versa.

In particular, prerequisite (iii) requires that generalizable data functions store disjoint but similar information, while the prerequisite (iv) requires that if a data function is mergeable with a path of data functions then the information provided by the former and the latter are either equivalent or one contains the other.

For example, the following correspondence statements between the data functions of *ICTRAIN* and *LOCTRAIN* may be specified:

- *ICTRAIN.trainno* is equivalent to *LOCTRAIN.trainno*
- *ICTRAIN.coachnum* is equivalent to *LOCTRAIN.coachnum*
- *ICTRAIN.departtime* is equivalent to *LOCTRAIN.starttime*
- *ICTRAIN.arrivetime* is equivalent to *LOCTRAIN.terminatetime*
- *ICTRAIN.departfrom* is contained in *LOCTRAIN.startfrom*

- *ICTRAIN.arriveat* is contained in *LOCTRAIN.terminateat*
- *ICTRAIN.stationname* is contained in *LOCTRAIN.stopname*
- *LOCTRAIN.madeby* is mergeable with *ICTRAIN.madeby*, *ICTRAIN.name*

Definition 7. If two nodes $D.T_1$ and $D.T_2$ of G are constructed from equivalence classes C_1 and C_2 respectively, and a data function $D_i.f_i :: \{(D_i.T_{i_1}, D_i.T_{i_2})\}$ is a rel-edge of G_i such that $D_i.T_{i_1} \in C_1$ and $D_i.T_{i_2} \in C_2$, then we call the edge $D_i.f_i$ a *component edge* between $D.T_1$ and $D.T_2$.

The construction of the integrated rel-subgraph consists of the integration of equivalent edges, containing edges, generalizable edges and mergeable edges. The integration begins by integrating equivalent component edges. The relationship *is-equivalent-to* between component edges is an equivalence relation which divides the component edges between two nodes $D.T_1$ and $D.T_2$ into equivalence classes. If two equivalence classes F_i and F_j contain edges f_i and f_j respectively such that f_i is contained in f_j, then we integrate these two classes into one class. Two equivalence classes might be further generalized. For example, suppose two generalizable data functions $starttime :: \{(Coach, Time)\}$ and $departtime :: \{(Train, Time)\}$, where $Coach$ and $Train$ are disjoint, exist in two different component databases and types $Coach$, $Train$ and $Time$ have been integrated as $Coach$, $Train$ and $Time$ in the integrated database. If the DBA has created a supertype $Vehicle$ for $Coach$ and $Train$ in the integrated database, he might also want to integrate $starttime$ and $departtime$. He can do so by generalizing $starttime$ and $departtime$ as $leavetime :: \{(Vehicle, Time)\}$. The mergeability of single data function $D_i.f_i$ with a path of data functions $D_j.f_{k_1}$, ..., $D_j.f_{k_m}$ represents a *structural conflict* in the component databases. In this case, by constructing $m-1$ intermediate nodes, we transform $D_i.f_i$ into a path of k data functions $D_i.f'_{k_1}$, ..., $D_i.f'_{k_m}$ such that $D_i.f'_{k_r}$ and $D_j.f_{k_r}$ $r = 1 \ldots m$ are equivalent or one contains the other.

After performing the above four kinds of integration, the final rel-edge equivalence classes are obtained. The data functions of EDB, are constructed from these classes and their names are chosen by the DBA. If $F_i = \{D_{i_1}.f_{i_1}, \ldots, D_{i_k}.f_{i_k}\}$ is such a class and its corresponding integrated data function is $D.f_i$, then the schema-level mappings obtained from this correspondence are:

$$M_{i_j}(D_{i_j}.f_{i_j}) = D.f_i,\ for\ j = 1 \ldots k$$

The value of $D.f_i$ is $\bigcup_{j=1}^{k} \{\tau_{D_{i_j}.t_{i_j}}(x) | x \in D_{i_j}.f_{i_j}\}$ where $\{(D_{i_j}.t_{i_j})\}$ is the type of $D_{i_j}.f_{i_j}$.

To illustrate steps (a)-(c) above, Figure 3 gives the graph representation of the $Schema$ and EDB of the integrated database $TRAIN$.

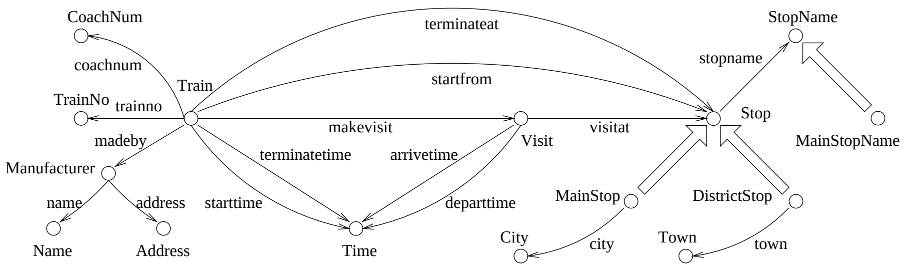

Fig. 3. Graph representation of $TRAIN$ database

3.3 IDB Integration

After integrating $Schema_1, \ldots, Schema_n$ and EDB_1, ..., EDB_n, the next step is to integrate IDB_1, ..., IDB_n.

We first transform function definitions into their integrated form before comparing their semantics. We recall that each IDB_i is partitioned into a series of strata, IDB_i^1, ..., $IDB_i^{s_i}$. We start by integrating the first set of strata IDB_1^1, ..., IDB_n^1. We then integrate the second set of strata, and so forth. Thus, when we are integrating the set of strata $IDB_1^j, \ldots, IDB_n^j$, we have already integrated $Schema_i$, EDB_i and $IDB_i^1, \ldots, IDB_i^{j-1}$ for $i = 1 \ldots n$. So transforming the definition of a derived function $D_i.f$ in IDB_i^j into its integrated form consists of translating the database constructs of the so far integrated database in its definition. In particular, each constant $c :: t$ is translated to $\tau_{D_i.t}(c)$, and each function g in $EDB_i \cup IDB_i$ is translated to $M_i(g)$ if $M_i(g) \in IDB$ or otherwise to g.

For two given derived functions, in order to know whether they can be integrated into one function we need compare their semantics. We define containment and identity relationships between two derived functions as follows:

Definition 8. For two derived functions $f, g :: t$, if for any element x in the semantic domain of t, $x \in f$ implies $x \in g$, then we say that g *contains* f or f *is contained by* g, which is expressed as $f \subseteq g$. If $f \subseteq g$ and $g \subseteq f$, then we say that f and g are *identical*, which is represented as $f \equiv g$.

Definition 9. For two sets of derived functions $F = \{f_1, \ldots, f_r\}$ and $G = \{g_1, \ldots, g_r\}$, we say that F and G are *mergeable* if

(i) F and G are two sets of mutual-recursive functions and either $f_m \equiv g_m$ or $f_m \subseteq g_m$, for $m = 1, \ldots, r$. Or
(ii) F and G are two singleton sets of non-recursive functions and $f_1 \equiv g_1$ or $f_1 \subseteq g_1$.

More generally, if k sets of derived functions $F_i = \{D_{i_1}.f_{i_1}, \ldots, D_{i_k}.f_{i_k}\}$ are mergeable, then then we will integrate them into one set of derived functions $F = \{D.f_1, \ldots, D.f_k\}$ such that:

$$M_{i_j}(D_{i_j}.f_{i_j}) = D.f_i, \; for \; j = 1 \ldots k$$

The value of $D.f_i$ is $\bigcup_{j=1}^{k} \{\tau_{D_{i_j}.t_{i_j}}(x) | x \in D_{i_j}.f_{i_j}\}$ where $\{(D_{i_j}.t_{i_j})\}$ is the type of $D_{i_j}.f_{i_j}$.

The overall algorithm for IDB integration is as follows, where n is the number of component databases, each IDB_i is partitioned into s_i strata and s is the number of strata the algorithm has constructed so far for D:

Algorithm Derived Function Integration:
$s := 0$;
for $j := 1$ **to** $max(s_1, \ldots, s_n)$ **do**
    for $i := 1$ **to** n **do**
        if $IDB_i^j \neq \emptyset$ **then**
            transform every function in IDB_i^j into its integrated form;
            $k := 1$;
            while $k \leq s$ **do**
                    if IDB_i^j *and* IDB^k *are mergeable* **then**
                        integrate IDB_i^j into IDB^k;
                        break;
                    end
                    $k := k + 1$;
            end;
            if $k > s$ **then**
                    $s := s + 1$;
                    $IDB^s := IDB_i^j$;
            end;
        end
    end
end.

When we integrate derived functions using the above algorithm, we must know whether the functions obey an identity or containment relationship. The following propositions provide a way to establish the relationships between derived functions by syntactically comparing their definitions, starting off with information about the identity or containment of data functions and recursively applying the propositions. Derived functions defined by a comprehension may need some syntactic transformation in order to facilitate the syntactical comparison, and a number of equivalences have been proposed in [9] which can utilized for this purpose. Formal proofs of the propositions are beyond the scope of this paper and will appear in a forthcoming technical report.

Proposition 10. Given two non-recursive derived functions f and g such that

$$f = \{e | p_1 \leftarrow f_1; \ldots; p_n \leftarrow f_n; Q\}$$
$$g = \{e | p_1 \leftarrow g_1; \ldots; p_n \leftarrow g_n; Q\}$$

if $f_i \equiv g_i$ for $i = 1 \ldots n$, then $f \equiv g$, otherwise if $f_i \subseteq g_i$ for $i = 1 \ldots n$, then $f \subseteq g$.

Proposition 11. Given two non-recursive derived functions f and g such that

$$f = f_1 \cup \ldots \cup f_m$$
$$g = g_1 \cup \ldots \cup g_n$$

if there is a mapping α from $\{f_1, \ldots, f_m\}$ to $\{g_1, \ldots, g_n\}$ such that for every $f_i \in \{f_1, \ldots, f_m\}$, $f_i \subseteq \alpha(f_i)$ then $f \subseteq g$. If both $f \subseteq g$ and $g \subseteq f$, then $f \equiv g$.

Proposition 12. Given two non-recursive derived functions f and g such that

$$f = \{e | p_1 \leftarrow f_1; \ldots; p_n \leftarrow f_n; Q'\}$$
$$g = \{e | p_1 \leftarrow f_1; \ldots; p_n \leftarrow f_n; Q\}$$

if for all instantiations of free variables, Q is true whenever Q' is true, then $f \subseteq g$.

Proposition 13. Given two sets of mutual recursive derived functions $\{f_1 = e_1, \ldots, f_r = e_r\}$ and $\{g_1 = e'_1, \ldots, g_r = e'_r\}$, if

$$(f_1 \equiv g_1) \& \ldots \& (f_r \equiv g_r) \Rightarrow (e_1 \equiv e'_1) \& \ldots \& (e_r \equiv e'_r)$$

then $f_i \equiv g_i$, for $i = 1 \ldots r$.

Proposition 14. Given two sets of mutual recursive derived functions $\{f_1 = e_1, \ldots, f_r = e_r\}$ and $\{g_1 = e'_1, \ldots, g_r = e'_r\}$, then if

$$(f_1 \subseteq g_1) \& \ldots \& (f_r \subseteq g_r) \Rightarrow (e_1 \subseteq e'_1) \& \ldots \& (e_r \subseteq e'_r)$$

and there is no derived function of the form $f' = \{h | p_1 \leftarrow g'_1; \ldots; p_n \leftarrow g'_n; Q\}$ in above two sets such that some g'_k is empty[1], then $f_i \subseteq g_i$, for $i = 1 \ldots r$.

To illustrate, we list the derived functions of $ICTRAIN$ and $LOCTRAIN$ in Appendices A and B respectively. The following relationships can be established between these derived functions, where $TR[f]$ denotes the translation of a derived function f into its integrated form:

- $TR[ICTRAIN.directreach] \equiv TR[LOCTRAIN.directreach1]$
- $TR[ICTRAIN.reachby1transfer] \subseteq TR[LOCTRAIN.onetransferreach]$
- $TR[ICTRAIN.easyreach] \subseteq TR[LOCTRAIN.easyreach]$
- $TR[ICTRAIN.idirectreach] \subseteq TR[LOCTRAIN.indirectreach]$
  $TR[ICTRAIN.reachable] \subseteq TR[LOCTRAIN.reachable]$

The resulting IDB is given in Appendix C.

[1] This condition guarantees that the proposition can be expressed as an *inclusive predicate* which can be proved by means of *fixed point induction* [14].

3.4 CDB Integration

The next integration step is to integrate $CDB_1, \ldots, CDB_n$ to give CDB. We recall that no integrity constraint calls any other integrity constraint, which means that each integrity constraint is defined in terms of data functions and derived functions only. Before integrating the integrity constraints in the component databases, we transform each of them into their integrated form by translating their definitions into the database constructs of $Schema$, EDB and IDB.

As with derived function integration, integrity constraint integration is also based on the semantics of the constraints. During the integration, for each component database D_i we compare the semantics of each constraint c_i in D_i with that of each integrity constraint c_j in each D_j, $j \neq i$. In both the following cases, c_i holds in the integrated database and we place it into a new stratum of the CDB: (i) if every D_j contains a constraint c_j such that c_i is contained by or is identical to c_j; (ii) if some D_j contain a constraint c_j such that c_i is contained by or is identical to c_j, but others do not, but if we evaluate c_i with respect to latter databases, either we always get the empty set or there are functions in the definition of c_i which are not defined in these databases. In all other cases we do not transfer c_i to D.

For example, when we compare the constraints in the component databases $ICTRAIN$ and $LOCTRAIN$, given in Appendices A and B, we have that:

- $TR[LOCTRAIN.onlytravelonce] \equiv TR[ICTRAIN.onetrip]$
- $TR[LOCTRAIN.traveltime] \subseteq TR[ICTRAIN.tripduration]$
- $TR[LOCTRAIN.rushhourtrain]$ is not applicable to $LOCTRAIN$.

Thus the integrity constraints obtained for the integrated database $TRAIN$ are as given in Appendix C.

3.5 PDB Integration

The final integration step consists of integrating $PDB_1, \ldots, PDB_n$ to give PDB. For this step, we do not attempt any automatic semantic integration of function definitions. We simply translate the function definitions of $PDB_1, \ldots, PDB_n$ into the database constructs of the so far integrated database and place the resulting definitions into the PDB.

4 Related Work

Since the 1980s a variety of approaches have been proposed for schema integration [6, 5, 12, 18]. Roughly, these approaches can be divided into two categories: *declarative statement* and *procedural statement* approaches. Our methodology for integrating the schema and extensional part of deductive databases falls into the category of declarative statement approaches.

Recently several papers addressing the integration of deductive databases have also appeared in literature [2, 19, 16]. Compared with these approaches, we propose a systematic method to compare the semantics of rules, and we integrate not only non-recursive derived functions but also recursive derived functions.

Some work has recently been done on the integration of integrity constraints [13, 1] and our approach is similar to both of these. All three approaches perform constraint integration in three steps: transformation, comparison of semantics and integration. However, in our approach we have provided a formal basis for comparing semantics and our integrity constraints are more expressive.

5 Conclusion

In this paper we have described a method for integrating deductive databases. For the extensional database integration, we use a binary relational ER model with subtyping as the common data model, and have proposed a semi-automatic method for integrating the component schemas where the DBA needs only declare the relationships between schema constructs and need not specify how the global schema is derived. For the intentional database integration, we integrate rules from different component databases according to their denotational semantics and have proposed a systematic method of comparing the semantics of rules; our method is useful especially in the case of recursive rule integration, which has not been dealt with by previous work. For further work we plan to investigate and formally define the notions of *correctness, completeness* and *minimality* [3] in our context, and to show that our method satisfies them, and to implement an interactive tool that supports our method.

A The EDB and CDB of the ICTRAIN database

```
_derived directreach = {(from,to)|(train1,station1)<-departfrom;
         (train2,station2)<-arriveat;(station3,from)<-stationname;
         (station4,to)<-stationname; train1==train2;
         station1==station3; station2==station4};
_derived reachby1transfer = {(from,to)|(from, to1)<-directreach;
                             (from2,to)<-directreach; to1==from2};
_derived easyreach = directreach U reachby1transfer;
_derived indirectreach = {(from,to)|(from, to1)<-directreach;
                          (from2,to)<-reachable; to1==from2};
_derived reachable = directreach U indirectreach;
_constraints onetrip = {(train1,dtime1,dtime2)|
         (train1,dtime1)<-departtime; (train2,dtime2)<-departtime;
         train1==train2; dtime1!=dtime2};
_constraints tripduration = {(train,dtime,atime)|
         (train,dtime)<-departtime; (train1,atime)<-arrivetime;
         train==train1; ((fst dtime - fst atime)>12)};
_constraints peaktimetrain = {(train,cnum)|
         (train,stime)<-starttime; (train1,cnum)<-coachnum;
         train==train1; (fst stime)<10; cnum<8};
```

B The EDB and CDB of the LOCTRAIN database

```
_derived callat = {(train,stname,dtime)|
         (train,visit1)<-makevisit; (visit2,stop2)<-visitat;
         (stop3,stname)<-stopname;(visit4,dtime)<-departtime;
         visit1==visit2; stop2==stop3; visit2==visit4};
_derived directreach1 = {(from,to)|(train1,stop1)<-startfrom;
         (train2,stop2)<-terminateat; (stop3,from)<-stopname;
         (stop4,to)<-stopname; train1==train2; stop1==stop3;
         stop2==stop4};
_derived directreach2 = {(from,to)|(train1,from,dtime1)<-callat;
         (train2,to,dtime2)<-callat; train1==train2;
         dtime1<dtime2};
_derived directreach3 = {(from,to)|(train1,stop1)<-startfrom;
         (train2,to,dtime2)<-callat; (stop3,from)<-stopname;
         train1==train2; stop1==stop3};
_derived directreach4 = {(from,to)|(train1,from,dtime1)<-callat;
         (train2,stop2)<-terminateat; (stop3,to)<-stopname;
         train1==train2; stop2==stop3};
_derived directreach  = directreach1 U directreach2
                        U directreach3 U directreach4;
_derived onetransferreach = {(from,to)|(from,to1)<-directreach;
                             (from2,to)<-directreach; to1==from2};
_derived easyreach = directreach U onetransferreach;
_derived indirectreach = {(from,to)|(from,to1)<-directreach;
                         (from2,to)<-reachable; to1==from2};
_derived reachable = directreach U indirectreach;
_constraints onlytravelonce = {(train1,stime1,stime2)|
         (train1,stime1)<-starttime; (train2,stime2)<-starttime;
         train1==train2; stime1!=stime2};
_constraints traveltime = {(train,stime,ttime)|
         (train,stime)<-starttime;(train1,ttime)<-terminatetime;
         train==train1; ((fst stime-fst ttime)>10)};
_constraints rushhourtrain = {(train,stname)|(train,stop1)<-stop;
         (stop2,stname)<-stopname;(train3,sttime)<-starttime;
         (stop1 in allDistrictStop); stop1==stop2; train==train3;
         (fst stime)<9}
```

C The EDB and CDB of the TRAIN database

```
_derived callat = {(train,stname,dtime)|(train,visit1)<-makevisit;
         (visit2,stn2)<-visitat; (stn3,stname)<-stationname;
         (visit4,dtime)<-departtime; visit1==visit2; stn2==stn3;
         visit2==visit4};
_derived directreach1 = {(from,to)|(train1,stn1)<-startfrom;
```

```
        (train2,stn2)<-terminateat; (stn3,from)<-stationname;
        (stn4,to)<-stationname; train1==train2; stn1==stn3;
        stn2==stn4};
_derived directreach2 = {(from,to)|(train1,from,dtime1)<-callat;
        (train2,to,dtime2)<-callat;train1==train2;dtime1<dtime2};
_derived directreach3 = {(from,to)|(train1,stn1)<-startfrom;
        (train2,to,dtime2)<-callat; (stn3,from)<-stationname;
        train1==train2; stn1==stn3};
_derived directreach4 = {(from,to)|(train1,from,dtime1)<-callat;
        (train2,stn2)<-terminateat; (stn3,to)<-stationname;
        train1==train2; stn2==stn3};
_derived directreach  = directreach1 U directreach2
                       U directreach3 U directreach4;
_derived onetransferreach = {(from,to)|(from,to1)<-directreach;
                            (from2,to)<-directreach; to1==from2};
_derived easyreach = directreach U onetransferreach;
_derived indirectreach = {(from,to)|(from,to1)<-directreach;
                         (from2,to)<-reachable; to1==from2};
_derived reachable = directreach U indirectreach;
_constraints onlytravelonce = {(train1,stime1,stime2)|
        (train1,stime1)<-starttime; (train2,stime2)<-starttime;
        train1==train2; stime1!=stime2};
_constraints traveltime = {(train,stime,ttime)|
        (train,stime)<-starttime;(train1,ttime)<-terminatetime;
        train==train1;((fst stime - fst ttime) > 12)};
_constraints rushhourtrain = {(train,stname)|(train,stop1)<-stop;
        (stop2,stname)<-stopname; (train3,sttime)<-starttime;
        (stop1 in allDistrictStop);stop1==stop2;
        train==train3; (fst stime)<9}
```

References

1. R. Alzahrani et al., "Integrity Merging in an Object-Oriented Federated Database Environment", in Proc. 13th British National Conference on Databases (BNCOD-13), Manchester, 1995. Springer-Verlag LNCS 940.
2. C. Baral, S. Kraus and J. Minker., "Combining Multiple Knowledge Bases", IEEE Trans. on Knowledge and Data Engineering, Vol.3, No.2, June 1991.
3. C. Batini, M. Lenzerini and S. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration", ACM Computing Surveys, Vol.18, No.4, Dec. 1986.
4. S. Courtenage and A. Poulovassilis, "Combining Inheritance and Parametric Polymorphism in a Functional Database Language", in Proc. 13th British National Conference on Databases (BNCOD-13), Manchester, 1995. Springer-Verlag LNCS 940.
5. J. Koh and A. Chen, "Integration of Heterogeneous Object Schemas," Proc. of Entity-Relationship Approach-ER'93, Arlington, Texas, USA, Dec. 1993.

6. A. Motro, "Superviews: Virtual Integration of Multiple Databases," IEEE Transactions on Software Engineering, Vol.13, No.7, July 1987.
7. A. Poulovassilis and C. Small, "A Functional Programming Approach to Deductive Databases", in Proc. 17th International Conference on Very Large Databases (VLDB), Barcelona, 1991.
8. A. Poulovassilis and C. Small, "A Domain-theoretic Approach to Integrating Functional and Logic Database Languages", in Proc. 19th International Conference on Very Large Databases (VLDB), Dublin, 1993.
9. A. Poulovassilis and C. Small, "Investigation of Algebraic Query Optimisation for Database Programming Languages", in Proc. International Conference on 20th Very Large Databases, Santiago, Chile, 1994.
10. A. Poulovassilis and C. Small, "Optimisation of Derived Functions in Database Programming Languages", Technical Report 96/14, Dept. of Computer Science, King's College London, 1996.
11. S. Reddi, "Integrity Constraint Enforcement in the Functional Database Language PFL", in Proc. 11th British National Conference on Databases (BNCOD-11), Keele, 1993. Springer-Verlag LNCS 696.
12. M. Reddy et al., "A Methodology for Integration of Heterogeneous Databases", IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.6, Dec. 1994.
13. M. Reddy, B. Prasad and A. Gupta, "Formulating global integrity constraints during derivation of global schema", Knowledge & Data Engineering, 16, 1995.
14. D.A. Schmidt, *Denotational Semantics*, Allyn and Bacon, 1986.
15. D.W. Shipman, "The functional data model and the data language DAPLEX", ACM Trans. on Database Systems, Vol.6, No.1, March 1981.
16. L. Sirounian and W. Grosky, "A Knowledge Model For Unifying Deductive and Non-Deductive Heterogeneous Databases", IEEE Trans. on Knowledge and Data Engineering, Vol.7, No.1, Feb. 1995.
17. C. Small and A. Poulovassilis, "An Overview of PFL", in Proc. 3rd International Workshop on Database Programming Languages, Nafplion, August 1991.
18. S. Spaccapietra and C. Parent, "View Integration: A Step Forward in Solving Structural Conflicts," IEEE Transactions on Knowledge and Data Engineering, Vol.6, No.2, April 1994.
19. V. Subrahmanian, "Amalgamating Knowledge bases", ACM Trans. on Database Systems, Vol.9, No.2, June 1994.
20. J.D. Ullman, *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.
21. L. Xu and A. Poulovassilis, "A Method for Integrating Deductive Databases," Technical Report 97/02, Dept. of Computer Science, King's College London, 1997.