On Referring Expressions in Information Systems derived from Conceptual Models

Alexander Borgida[†], David Toman[‡] and Grant Weddell[‡]

[†]Department of Computer Science, Rutgers University, New Brunswick, USA
`borgida@cs.rutgers.edu`

[‡]Cheriton School of Computer Science, University of Waterloo, Canada
`{david,gweddell}@uwaterloo.ca`

Abstract. We apply recent work on referring expression types to the issue of identification in conceptual modelling. In particular, we consider how such types yield a separation of concerns in a setting where an Information System based on a conceptual schema is to be mapped to a relational schema plus SQL queries. We start from a simple object-centered representation (as in semantic data models), where naming is not an issue because everything is self-identified (possibly using surrogates).We then allow the analyst to attach to every class a preferred "referring expression type", and to specify uniqueness constraints in the form of generalized functional dependencies. We show (1) how a number of well-formedness conditions concerning an assignment of referring expressions can be efficiently diagnosed, and (2) how a concrete relational schema and SQL queries over this schema are derived from a combination of the conceptual schema and queries over it, once identification issues have been separately resolved as above.

1 Introduction

The Entity Relationship notation, and its many extensions, were designed with the explicit purpose of helping to derive relational database schemata from the conceptual model. One feature of the relational model, namely that attribute fillers must be values such as strings and integers, means that relationships between entities need to be represented as relationships between their "names" (primary keys), and therefore all entities need to have primary keys. This resulted in (E)ER modelers having to *pay premature attention to naming issues*. For example:

(a) One cannot create tables representing relationships before one has decided on how entities are going to be externally named.
(b) One needs to distinguish right from the beginning "weak entitiy sets", like `ROOM`, with attributes `room-num` and `capacity`, which are insufficient to act as an external key, from regular entity sets like `BUILDING`, with attribute `address` that can identify it. This is necessary even though no deep ontological factors distinguish `ROOM` and `BUILDING`.

(c) If entity set `PERSON` is identified by `ssn`, and it has subclass `FAMOUS-PERSON`, then the latter must inherit its identifier from the superclass. This, despite the fact that we might prefer as identifier the attribute `name` for `FAMOUS-PERSON`, or maybe even `star-name` (like Bono or The Edge), which is only applicable to `FAMOUS-PERSON`.

(d) Certain entity sets are introduced by *generalization* as the union of very heterogeneous sub-classes; for example, `LEGAL-ENTITY` is the generalization of `PERSON` and `COMPANY`, for the purpose acting as participant in relationships such as ownership. In such cases, one is forced to immediately create an artificial attribute, (e.g., `legal-entity-number`), which replaces the natural keys of `PERSON` (e.g., `ssn`) and `COMPANY` (e.g., `corp-name` and `city`). Since `legal-entity-number` is meaningless to end users, programmers must always remember to perform joins so as to include the usual keys of the subclasses, depending on the individuals returned.

Note that when using an object-centered modelling notation supporting object identity, such as most semantic data models since Taxis [4], problems (a) and (b) do not arise at the beginning because relationships are stated between objects themselves. Our essential starting point is that one can therefore postpone the naming issue to a separate pass. Unfortunately, problems like (c) and (d) persist. We propose a multi-part approach to address these. The following outlines the remainder of the paper and how our results provide a basis for this approach:[1]

(1) After this enumeration, we introduce by example a simple conceptual modelling notation, $\mathcal{C}$, that has the common features of so-called "attribute-based" semantic models, such as those surveyed in Table 12 of [3]. The data model is based on the familiar object-centered view of the world consisting of individual objects, with attributes that can have as values either other objects or atomic values, such as SQL datatypes. The objects are grouped into classes, satisfying a variety of constraints, such as subclass hierarchies, disjointness and coverage relating to other classes, and a general form of functional dependency. As we hope to illustrate below, $\mathcal{C}$ can serve as a lingua franca for standard conceptual models such as EER, UML class diagrams, DL-Lite ontologies, and so on. The important point is that $\mathcal{C}$ does this without the need to decide external referring expressions for objects. In Section 2, we provide a slight syntactic variant, $\mathcal{C}_{\mathrm{AR}}$, of $\mathcal{C}$, which gives it more of a relational flavor by making "internal" object identifiers visible as "abstract" attribute values.

(2) As with all object-centered conceptual models, one can use arbitrary SQL-like queries over $\mathcal{C}$, and especially $\mathcal{C}_{\mathrm{AR}}$ schemas, where variables range over extents of classes, in order to express data access requirements. In Section 2, we also introduce SQL^{path}, a core of SQL in which it is possible to employ "dotted path notation" (e.g., `x.manager.salary`) to avoid explicit foreign key joins, and hence make queries shorter.[2]

[1] The outline is followed by a sequence of examples that illustrate intuitively the entire process. The remainder of the paper is a more formal development of the ideas.

[2] This and other earlier language features of $\mathcal{C}$ were already available in Taxis [4] and GEM [7].

(3) In a separate, orthogonal pass, modellers specify (*i*) functional dependency-like constraints that include keys, and (*ii*) *preferred naming schemes* for each class in a *referring expression type language*, introduced in Section 3. The notation makes it possible to address situations like (b), (c) and (d) above.

(4) Given a $\mathcal{C}_{\mathrm{AR}}$ schema, a referring expression type assignment and a set of SQL^{path} queries, algorithms can be given to perform several key tasks:

– In Section 3, we show how to verify that the referring expressions comprising the naming schemes in **(3)** above do indeed uniquely identify objects in tables and queries.
– In Section 4, we how show how the referring expression types in **(3)** guide the replacement of abstract attributes in a $\mathcal{C}$ schema with sequences of concrete attributes, thus obtaining a concrete relational schema.
– In Section 4, we also show how to translate any SQL^{path} query into *a provably equivalent* regular SQL query over the concrete relational schema, thereby eliminating all path expressions as well as non-printable object ids that might have been returned by the query.

We illustrate the above using a situation where legal entities, which are either persons or companies, can own vehicles (one owner per vehicle), while persons can drive vehicles. A conceptual schema expressed in $\mathcal{C}$ might be given as follows:[3]

```
class PERSON (ssn: INT, name: STRING, isa LEGAL-ENTITY,
  disjoint with VEHICLE)
class COMPANY (corp-name: STRING, city: STRING, isa LEGAL-ENTITY)
class LEGAL-ENTITY (covered by PERSON, COMPANY)
class VEHICLE (vin: INT, make: STRING, owned-by: LEGAL-ENTITY)
class CAN-DRIVE (driver: PERSON, driven: VEHICLE)
```

Given this schema, a modeller will then be able to express the following queries in SQL^{path}:

– "*The name of anyone who can drive a vehicle made by Ford*":

```
select d.driver.name from CAN-DRIVE d
where d.driven.make = 'Ford'
```

– "*The owners of GM vehicles*":

```
select v.owned-by from VEHICLE where v.make = 'GM'
```

Note that as it stands, the second query does not specify how the (heterogeneous) owners, which share no common concrete attributes that can identify them, will be described in the final answer.

Concurrently with writing queries, modellers can address external naming preferences. Note that this might require adding (or discovering) functional dependencies from which one can derive key/uniqueness information. The naming process might start by stating that `ssn` is a key for `PERSON`, and that the combination of attributes `(corp-name,city)` is a key for `COMPANY`; and then associating referring expression types "`ssn=?`" to class `PERSON`, and "`(corp-name=?,`

[3] We explain the correspondence to a $\mathcal{C}_{\mathrm{AR}}$ schema in the next section.

`city=?)`" to class `COMPANY`, making these key values the references. (Note that `PERSON` might have had other keys.) A referring expression type for `LEGAL-ENTITY` objects might be given as follows:

$$\texttt{PERSON} \rightarrow \texttt{ssn=?; COMPANY} \rightarrow \texttt{(corp-name=?, city=?).}$$

Were it possible for an object to be both a person and a company, the use of ";" expresses a *preference* for using `ssn` attribute values for identifying the object.

Once it has been verified that this assignment of referring expression types is *well-formed* in the sense that it resolves all identification issues (see Section 3), it then becomes possible to automatically map the schema originally given by modellers to a concrete relational schema with additional primary key attributes and with "object pointer" attributes replaced by (sequences of) concrete attributes. In turn, SQL^{path} queries are similarly translated to executable SQL queries over this schema (see Section 4). To illustrate, the following are (parts of) the concrete tables that would be produced for `PERSON`, `LEGAL-ENTITY` and `VEHICLE`:

```
table PERSON (ssn INT, name STRING, primary key (ssn),...)
table LEGAL-ENTITY (disc enum{'PERSON','COMPANY'}, ssn INT,
  corp-name STRING, city STRING, primary key (disc, ssn, corp-name, city))
table VEHICLE (vin, make, owner-disc, owner-ssn,owner-co-name,owner-city,
  foreign key (owner-disc, owner-ssn, owner-corp-name, owner-city)
  references LEGAL-ENTITY (disc, ssn, corp-name, city) )
```

Note how identification for `LEGAL-ENTITY` objects is ultimately resolved: four attributes are added, with attribute `disc` acting as a discriminant in variant records. (We assume inapplicable attributes are always initialized with default non-null values.) To illustrate how SQL^{path} queries are mapped, consider the second example query above. It maps to the following executable query:

```
select v.disc, v.ssn, v.corp-name, v.city from VEHICLE v
where v.make = 'GM'.
```

2 Abstract Relational Databases

We now introduce $\mathcal{C}_{\mathrm{AR}}$, a minor variant of the modeling language used in Section 1 examples. Essentially, it makes object identifiers *programmer visible*, in order to bring data declaration and manipulation syntax closer to SQL, which is more familiar to application programmers.

Definition 1 ($\mathcal{C}_{\mathrm{AR}}$: A more relational but still abstract view of $\mathcal{C}$) Let TAB, AT, and CD be sets of table names, attribute names, and *concrete domains* (data types), respectively, and let `OID` be an *abstract domain* of *identifiers/surrogates*, disjoint from all concrete domains. A $\mathcal{C}_{\mathrm{AR}}$ schema Σ is a set of *abstract table declarations* of the form

$$\texttt{table } T \texttt{ (self OID, } A_1\ D_1 \texttt{, ... , } A_k\ D_k \texttt{, } \varphi_1 \texttt{, ... , } \varphi_\ell \texttt{)}$$

where $T \in \mathsf{TAB}$, `self` $\in \mathsf{AT}$ is the *primary key* of T (a *distinguished attribute* identifying the aggregation $(A_1, \ldots, A_k \in \mathsf{AT})$); $D_i \in \mathsf{CD} \cup \{\texttt{OID}\}$, and φ_j are *constraints* attached to the abstract table T (see below). □

To illustrate, we begin translating class PERSON into $\mathcal{C}_{\rm AR}$ as follows:

```
            table PERSON( self OID, ssn INT, name STRING, ...
```

Note the occurrence of attribute `self` — the user visible object identifier. There are five kinds of constraints relevant to identification issues, and which we use in our examples:[4]

1. (*foreign keys*) `foreign key` (A) `references` T
2. (*specialization*) `isa` T
3. (*cover constraints*) `covered by` $\{T_1, \ldots, T_m\}$
4. (*disjointness constraints*) `disjoint with` T
5. (*path functional dependencies*) `pathfd` $\mathsf{Pf}_1, \ldots, \mathsf{Pf}_n \rightarrow \mathsf{Pf}$

Continuing with the translation of class PERSON to $\mathcal{C}_{\rm AR}$, we add three constraints:

```
    ... isa LEGAL-ENTITY, disjoint with COMPANY, pathfd ssn → self ).
```

The first two constraints assert that `self` values of `PERSON` tuples are a subset of `self` values of `LEGAL-ENTITY` tuples, and are disjoint from `self` values of `COMPANY` tuples; the third asserts that any pair of `PERSON` tuples agreeing on `ssn` also agree on `self` (i.e., `ssn` is a key for `PERSON`). In fact, `pathfd`s are more general and powerful: one can declare

```
          table OFFICE( self OID, office-num INT, located-in OID,
            foreign key (located-in) references BUILDING,
            pathfd office-numb, located-in.address → self )
```

once having specified that buildings have addresses. This says that (*i*) each value of `located-in` must appear as the `self` value of a `BUILDING` tuple, and (*ii*) the office number and the address of the office's building form a key for offices. The latter addresses the issue (b) of weak entity identification, using the power of *attribute paths*, `located-in.address` in this case.

We call attributes ranging over $D_i \in \mathsf{CD}$ *concrete*, since their values are atomic, such as the INTegers, and the remaining attributes *abstract*. Also, without loss of generality, we assume that every attribute A_i, other than `self`, is included in the declaration of at most one abstract table, and write $\mathsf{Home}(A_i)$ to refer to this table. If A_i is abstract, we assume there is a foreign key constraint for A_i to some abstract table, referred to as $\mathsf{Dom}(A_i)$. Thus, $\mathsf{Home}(\texttt{ssn}) = \texttt{PERSON}$ and $\mathsf{Dom}(\texttt{located-in}) = \texttt{BUILDING}$ in the above. Finally, an instance I of an abstract table T is a (finite) set of $k+1$ tuples in which `self` ranges over `OID` and is the *primary key* of T, and A_i range over D_i.

For a table T in schema Σ, we write $\Sigma \models \varphi \in T$ to denote the fact that a particular constraint φ for T (possibly not explicitly stated) *logically follows* from the constraints in schema Σ. For example, the above declaration of `PERSON` logically entails that "(`pathfd  ssn, name` $\rightarrow$ `self`)" also holds for `PERSON`. And if $\{T_1, T_2\}$ cover T, then so does $\{T_1, T_2, S\}$ for any S.

[4] To adhere to SQL'99 syntax, a formulation using a general `assertion` would be needed in most cases.

With respect to a particular abstract table T', a `foreign key` constraint requires each A value of a T' tuple to also occur as a `self` value for some T tuple. (Note that A must therefore be abstract.)

The meaning of `isa`, `cover` and `disjoint` constraints is obvious, relating the `self` values of various tables.

Path functional dependencies are a strict generalization of keys and functional dependencies common in classical relational modelling, and considerably extend how identification issues can be resolved. In particular, such constraints allow *attribute paths* in place of attributes, as illustrated in the case of `OFFICE` above. Paths are defined as follows:

Definition 2 (Attribute Paths) An *attribute path* Pf is an expression of the form $A_1.A_2.\ldots.A_k$, where $A_i \in \mathsf{AT}$ and $k \geq 1$, and is *well defined for an abstract table* T if $k = 1$ and A_1 is `self`, or if $\mathsf{Home}(A_1) = T'$, $\Sigma \models (\texttt{isa}\ T') \in T$ and either $k = 1$ or $\Sigma \models (\texttt{foreign key}\ A_1\ \texttt{references}\ T'') \in T'$ and the attribute path $A_2.\cdots.A_k$ is well defined for T''. □

The value of an attribute path corresponds to *navigating* the attributes starting from (the key attribute `self` of) a particular tuple $t \in T$. In relational terms the navigation corresponds to a sequence of *foreign key joins*.

The meaning of $\texttt{pathfd}\ \mathsf{Pf}_1, \ldots, \mathsf{Pf}_n \rightarrow \mathsf{Pf}$ is then that two tuples in a T' instance that agree on the values of all Pf_i must also agree on the value of Pf.

The problem of deciding when $\Sigma \models \varphi \in T$ holds can be reduced to reasoning about logical consequence in the *description logic* $\mathcal{DLFD}$ [5], which is decidable. If no `cover` constraints occur in Σ, it also becomes possible to reduce such questions to reasoning about logical consequence in the description logic $\mathcal{CFDI}_{nc}^{\forall-}$ [6], which is decidable in PTIME. (Further details, however, are beyond the scope of this paper.)

We introduce next a core relational algebra fragment of SQL that incorporates attribute paths:

Definition 3 (SQL^{path}) The following grammar gives the syntax for (an idealized) SQL-like query language over instances of $\mathcal{C}$ schema:

$$\begin{array}{rll} Q ::= & T\ x & (\textit{table reference}) \\ \mid & \texttt{select}\ x_1.\,\mathsf{Pf}_1, \ldots, x_k.\,\mathsf{Pf}_k\ Q & (\textit{projection}) \\ \mid & \texttt{from}\ Q, Q & (\textit{product}) \\ \mid & Q\ \texttt{where}\ x.\,\mathsf{Pf}_1 = y.\,\mathsf{Pf}_2 & (\textit{selection}) \\ \mid & Q\ \texttt{union}\ Q & (\textit{union}) \\ \mid & Q\ \texttt{minus}\ Q & (\textit{set difference}) \end{array}$$

As in SQL, we require that all variables are appropriately *bound* with a "$T\ x$" clause, and denote T by $\mathsf{Bound}(x)$; that Pf is well defined for $\mathsf{Bound}(x)$ for any term "$x.\,\mathsf{Pf}$"; that variables in subqueries of the `from` clause are disjoint; and that the subqueries in the `union` and `minus` operations are *union compatible*. We also assume the standard SQL-like interpretation of the above syntax. □

In summary, SQL^{path} deviates from standard SQL in three ways:

1. In addition to *standard atomic datatypes*, we have introduced an abstract domain `OID` (this then allows abstract attributes, and the ability to refer directly to abstract identifiers with expressions of the form "$x.A$").

2. We allow the use of attribute paths in place of single attributes in `where` conditions.
3. We allow “$T\ x$” as a query (where SQL would require “$\texttt{select} * \texttt{from}\ T\ x$”).

We also allow obvious syntactic sugar to make examples more readable, such as using conjunction in the `where` clauses, and multi-arity `from` clauses instead of the nested use of `from`.

Less obviously, terms of the form “$x_1.\mathsf{Pf}.A$” occurring in `select` and `where` clauses can ultimately be replaced by terms of the form “$x_2.A$” by repeatedly applying the following rewrite rules (including a symmetric version of the first rule for right-hand-sides of a selection condition):

“$Q\ \texttt{where}\ x_1.A.\mathsf{Pf}_1 = x_2.\mathsf{Pf}_2$” $\rightsquigarrow$
“$(\texttt{from}\ Q, T\ x_3\ \texttt{where}\ x_1.A = x_3.\texttt{self})\ \texttt{where}\ x_3.\mathsf{Pf}_1 = x_2.\mathsf{Pf}_2$”

and

“$\texttt{select}\ \ldots, x_1.A.\mathsf{Pf}_1, \ldots\ Q$” $\rightsquigarrow$
“$\texttt{select}\ \ldots, x_2.\mathsf{Pf}_1, \ldots\ ((\texttt{from}\ T\ x_2, Q)\ \texttt{where}\ x_1.A = x_2.\texttt{self})$”,

where, in all cases, $\Sigma \models (\texttt{foreign key}\ (A)\ \texttt{references}\ T) \in \mathsf{Bound}(x_1)$ and Pf_1 is well defined for T. Additional rewrite rules can ensure, for terms of the form “$x.A$”, that home tables of A-values correspond to the bound tables for variables (i.e., are not inherited by `isa` constraints):

“$Q\ \texttt{where}\ x_1.A_1 = x_2.A_2$” $\rightsquigarrow$
“$(\texttt{from}\ Q, \mathsf{Home}(A_1)\ x_3\ \texttt{where}\ x_1.\texttt{self} = x_3.\texttt{self})\ \texttt{where}\ x_3.A_1 = x_2.A_2$”,

where $\mathsf{Home}(A_1) \neq \mathsf{Bound}(x_1)$. For example, applying such rewritings to our introductory query

```
select d.driver.name from CAN-DRIVE d
where d.driven.make = 'Ford'
```

would ultimately produce the query

```
select p.name from CAN-DRIVE d, PERSON p, VEHICLE v
where v.make = 'Ford' and d.driven = v.self and d.driver = p.self.        (1)
```

Note that this requires, e.g., confirming that

$\Sigma \models (\texttt{foreign key (driven) references VEHICLE}) \in \texttt{CAN-DRIVE}$

and that `make` is an attribute of `VEHICLE`.

3 Managing Identity

By introducing the purely abstract domain `OID`, $\mathcal{C}_{\mathrm{AR}}$ frees the user from any need to address identification issues when formulating queries. This enables our main contribution: a separation of concerns in which identification issues can be addressed concurrently with the formulation of data access requirements. But unlike various *object models*, the values of attributes over this domain, including the primary key attribute `self`, are *purely abstract* and *are not storable* in concrete table instances.

We now show how these issues can be resolved by using a *referring expression type language* proposed in [2]. Intuitively, a type in this language defines a space of first-order formulas free in one variable.[5] The objective is for each formula to be true for *exactly one* object in `OID` in every abstract schema instance that satisfies all schema constraints. The language is given in the following:

Definition 4 (Referring Expression Types and Assignments)
A *referring expression type* Rt is an instance of a recursive *pattern* language given by the grammar:

$$Rt \quad ::= \quad \mathsf{Pf} = ? \;\mid\; Rt, Rt \;\mid\; G \to Rt, \;\mid\; Rt; Rt$$

where Pf is an attribute path ending in a concrete attribute, and where $G = \{T_1, \ldots, T_\ell\}$ is a set of table names called a *guard*. Let Σ be a $\mathcal{C}_{\mathrm{AR}}$ schema. We write $\mathrm{R{\scriptstyle E}}(Rt)$ to refer to a set of *referring expressions* ϕ_i induced by a given referring expression type Rt relative to Σ as follows:

$$\begin{aligned}
\mathrm{R{\scriptstyle E}}(\mathsf{Pf} = ?) &= \{x.\,\mathsf{Pf} = a \mid a \text{ a constant}\}\\
\mathrm{R{\scriptstyle E}}(Rt_1, Rt_2) &= \{\phi_1 \wedge \phi_2 \mid \phi_i \in \mathrm{R{\scriptstyle E}}(Rt_i)\}\\
\mathrm{R{\scriptstyle E}}(\{T_1, \ldots, T_k\} \to Rt) &= \{\textstyle\bigwedge_{i=1}^{k}(\exists y_1, \ldots, y_l.T_i(x, y_1, \ldots, y_l)) \wedge \phi \mid \phi \in \mathrm{R{\scriptstyle E}}(Rt)\}\\
\mathrm{R{\scriptstyle E}}(Rt_1; Rt_2) &= \mathrm{R{\scriptstyle E}}(Rt_1) \cup \{\phi \in \mathrm{R{\scriptstyle E}}(Rt_2) \mid \neg\exists\psi \in \mathrm{R{\scriptstyle E}}(Rt_1).(\phi \equiv \psi)\}
\end{aligned}$$

Given $T \in \mathsf{Tables}(\Sigma)$, we say that Rt is *weakly identifying* for T if, for all instances I of Σ,

$$\begin{aligned}
I \models \forall x_1, x_2.(&\exists y_1, \ldots, y_l.T(x, y_1, \ldots, y_l) \wedge \phi(x/x_1)) \wedge\\
&(\exists y_1, \ldots, y_l.T(x, y_1, \ldots, y_l) \wedge \phi(x/x_2)) \to x_1 = x_2,
\end{aligned}$$

holds for all $\phi \in \mathrm{R{\scriptstyle E}}(Rt)$, and that Rt is *strongly identifying for* T if in addition

$$I \models \neg\exists x.(\phi_1 \wedge \phi_2)$$

holds for all syntactically distinct $\phi_1, \phi_2 \in \mathrm{R{\scriptstyle E}}(Rt)$.

A *referring type assignment* for Σ is a mapping RTA from $\mathsf{Tables}(\Sigma)$ to referring expression types. □

For example, RTA might assign either "`ssn=?`" or "`name=?`" as the referring expression type for `PERSON`. The former would quality as strongly identifying, however the latter would not (since, e.g., two people can have the same name).

In [2], it is also shown that any Rt can be converted to a normal form with the following structure:

$$G_1 \to (\mathsf{Pf}_{1,1} = ?, \ldots, \mathsf{Pf}_{1,k_1} = ?); \ldots; G_k \to (\mathsf{Pf}_{k,1} = ?, \ldots, \mathsf{Pf}_{k,k_k} = ?).$$

We call each subexpression "$G_i \to (\mathsf{Pf}_{i,1} = ?, \ldots, \mathsf{Pf}_{i,k_i} = ?)$" of this normal form separated by ";" a *component* of Rt. For the remainder of the paper, we assume $\mathsf{RTA}(T)$ is already in this form, for any $T \in \Sigma$, and also that each guard G_i contains at most one table name.[6] To improve readability, we omit empty guards, and write T as shorthand for guard $\{T\}$. Finally, we write $\mathsf{Fix}(Rt, T)$ to denote a normal form Rt with T added to any empty guard. Thus, $\mathsf{Fix}(Rt, T)$

[5] Each formula can be viewed as a SQL^{path}query computing a table with a single "x" column.

[6] Allowing guards to have more than one table name is a straightforward extension.

will have the form

$$T_1 \to (\mathsf{Pf}_{1,1} = ?, \ldots, \mathsf{Pf}_{1,k_1} = ?); \ldots ; T_k \to (\mathsf{Pf}_{k,1} = ?, \ldots, \mathsf{Pf}_{k,k_k} = ?). \qquad (2)$$

The next definition deals with such situations in which, e.g., $\mathsf{RTA}(\texttt{PERSON})$ is given by "`PERSON` $\to$ `ssn=?; name=?`". Here, "`name=?`" will be ignored since the guard of the first component has precedence and will "catch" any `PERSON`.

Definition 5 (Non-redundant Referring Types) Let Σ be a $\mathcal{C}_{\mathrm{AR}}$ schema, RTA a referring type assignment, $T \in \mathsf{Tables}(\Sigma)$, and assume $\mathsf{Fix}(\mathsf{RTA}(T), T)$ has the form (2). We say that the jth component "$T_j \to (\mathsf{Pf}_{j,1} = ?, \ldots, \mathsf{Pf}_{j,k_j} = ?)$" is *redundant with respect to* T if it satisfies any of the following conditions:

$$\begin{array}{l}(a)\ \Sigma \models (\texttt{covered by } \{T_1, \ldots, T_{j-1}\}) \in T_j,\\ (b)\ \Sigma \models (\texttt{covered by } \{T_1, \ldots, T_{j-1}\}) \in T, \text{ or}\\ (c)\ \Sigma \models (\texttt{disjoint with } T_j) \in T.\end{array}$$

Given an arbitrary Rt in normal form, we write $\mathsf{Prune}(Rt, T)$ to denote the referring expression $\mathsf{Fix}(Rt, T)$ from which all components redundant with respect to T have been removed. $\Box$

It is easy to see that each of the three cases identify a component that can be safely removed from a referring expression type assignment for a table T: the first two since any qualifying object will have a referring expression induced by earlier components, and the latter since the guard will never apply for any T object.

The following example illustrates another potential problem with a given RTA: that not all possible referring type assignments can support synthesizing arbitrary concrete SQL queries:

Example 6 Consider the SQL^{path} query

$$\texttt{select } x.\texttt{self} \texttt{ from } T_1\ x,\ T_2\ y \texttt{ where } x.\texttt{self} = y.\texttt{self} \qquad (3)$$

over a $\mathcal{C}_{\mathrm{AR}}$ schema Σ in which T_i is declared as follows:

$$\texttt{create } T_i \texttt{ (self OID, } A_i \texttt{ STRING, pathfd } A_i \to \texttt{self).} \qquad (4)$$

When $\mathsf{RTA}(T_i)$ is given by "$A_i = ?$", the ability to compare the `OID` values is lost since the differing referring expressions associated with T_1 and T_2 do not provide a way to determine if the same object belongs to both tables. The problem is solved, e.g., by instead defining $\mathsf{RTA}(T_2)$ as "$T_1 \to A_1 = ?; A_2 = ?$" since T_2 objects are then *identified* by A_1 values when also in T_1. $\Box$

All such mapping issues are avoided when a referring expression type assignment is *identity resolving*, which can be defined as follows:

Definition 7 (Identity Resolving Type Assignments)
Let Σ be a $\mathcal{C}_{\mathrm{AR}}$ schema and RTA a referring type assignment for Σ. Given a linear order $\mathcal{O} = (T_{i_1}, \ldots, T_{i_k})$ on the set $\mathsf{Tables}(\Sigma)$, define $\mathcal{O}(\mathsf{RTA})$ as the following referring expression type:

$$\mathsf{Fix}(\mathsf{RTA}(T_{i_1}), T_{i_1}); \ldots ; \mathsf{Fix}(\mathsf{RTA}(T_{i_k}), T_{i_k}).$$

We say that RTA is *identity resolving* if there is some linear order $\mathcal{O}$ such that the following conditions hold for each $T \in \mathsf{Tables}(\Sigma)$:

1. $\mathsf{Fix}(\mathsf{RTA}(T), T) = \mathsf{Prune}(\mathcal{O}(\mathsf{RTA}), T)$,

2. $\Sigma \models (\texttt{covered by } \{T_1, ..., T_n\}) \in T$, where $\{T_1, ..., T_n\}$ are all tables occurring in the guards in $\mathsf{Fix}(\mathsf{RTA}(T), T)$, and
3. for each component $T_j \rightarrow (\mathsf{Pf}_{j,1} = ?, \ldots, \mathsf{Pf}_{j,k_j} = ?)$ of $\mathsf{Fix}(\mathsf{RTA}(T), T)$, the following also holds: (i) $\mathsf{Pf}_{j,i}$ is well defined for T_j, for $1 \leq i \leq k_j$, and (ensuring *strong identification*) (ii) $\Sigma \models (\texttt{pathfd } \mathsf{Pf}_{j,1}, \ldots, \mathsf{Pf}_{j,k_j} \rightarrow \texttt{self}) \in T_j$.

We write $\mathsf{Order}(\mathsf{RTA})$ for a fixed choice for such an order when one exists. □

Given an RTA, the existence of $\mathcal{O}$ can tested by checking for cycles in a graph with nodes labeled by table names and directed edges connecting tables that appear in consecutive guards of a referring type assigned by RTA. The linear order is then any topological sort of the (acyclic) graph. The remaining conditions can also be checked by appeal to the description logics $\mathcal{DLFD}$ [5], and $\mathcal{CFDI}_{nc}^{\forall -}$ [6] (see previous section).

Example 8 Consider the SQL^{path} query and $\mathcal{C}_{\mathrm{AR}}$ schema Σ given by (3) and (4) in Example 6 above, and also assume $\mathsf{RTA}(T_1)$ and $\mathsf{RTA}(T_2)$ are given respectively by "$A_1 = ?$" and "$T_1 \rightarrow A_1 = ?; A_2 = ?$". Then $\mathsf{RTA}(T_2)$ implies that T_1 must precede T_2 in $\mathsf{Order}(\mathsf{RTA})$. Indeed, the linear order $\mathcal{O} = (T_1, T_2)$ satisfies all conditions required for RTA to be identity resolving. In contrast, if $\mathsf{RTA}(T_1)$ is instead given by "$T_2 \rightarrow A_2 = ?; A_1 = ?$", then no such linear order $\mathcal{O}$ exists and RTA is not identity resolving. This can be blamed on an inherent ambiguity on how objects belonging to both tables should be referenced.

More generally, entity sets/classes are often assumed to be disjoint, unless they participate in an `isa` hierarchy. In such cases, one should be free to chose the identifying Rt independently. For example, consider where $\mathsf{RTA}(T_i)$ is given by "$A_i = ?$", and where the constraint "`disjoint with` T_2" is added to T_1. RTA is now identity resolving in this case since all conditions hold for $\mathcal{O} = (T_1, T_2)$ (or for $\mathcal{O} = (T_2, T_1)$).

Finally, consider the other end of the spectrum, in which it becomes possible to include a global identifier, say `uri`, over some concrete domain. In our T_i setting, this can be achieved by adding to Σ the table

```
table UNIVERSE ( self OID, uri STRING, pathfd uri → self ),
```

and by adding constraint "`isa UNIVERSE`" to T_1 and T_2. If "`uri` $= ?$" is the referring expression type for all abstract tables, then RTA is also identity resolving. □

An identity resolving referring type assignment yields a natural way to *coerce* referring expression types to more general types. This is based on the observation that, for a linear order $(T_{i_1}, \ldots, T_{i_k})$, all referring expression types that are formed as sub-sequences of components of RTA can be simply extended with additional components as long as the result is still a sub-sequence of Rt.

Definition 9 (Coercion) Let Σ be a $\mathcal{C}_{\mathrm{AR}}$ schema, RTA an identity resolving referring type assignment for Σ, and Rt_1 and Rt_2 two referring expressions with component orders conforming to $\mathsf{Order}(\mathsf{RTA})$. We say that Rt_1 is a *referring supertype of* Rt_2 if all components of Rt_2 are also components of Rt_1, and write $Rt_1 \triangledown Rt_2$ to denote the *least common referring supertype* of both Rt_1 and Rt_2, that is, a referring expression type with components given by the union of the components of Rt_1 and Rt_2 and that are ordered by $\mathsf{Order}(\mathsf{RTA})$. □

Example 10 Consider the SQL^{path} query

$$(\texttt{select } x.\texttt{self } T_1\ x) \texttt{ union } (\texttt{select } x.\texttt{self } T_2\ x) \qquad (5)$$

over schema (4) in Example 6 above, and also assume $\mathsf{RTA}(T_1)$ and $\mathsf{RTA}(T_2)$ are given respectively by "$A_1 = ?$" and "$T_1 \to A_1 = ?; A_2 = ?$". As Example 8 shows, RTA is an identity resolving type assignment. However, the union operation requires a coercion to a common referring expression type for T_1 and T_2. In particular, since $\mathsf{Order}(\mathsf{RTA}) = (T_1, T_2)$, $\mathsf{RTA}(T_1) \triangledown \mathsf{RTA}(T_2)$ defines this as "$T_1 \to A_1 = ?; A_2 = ?$" (matching $\mathsf{RTA}(T_2)$). Thus, an *encoding* of referring expressions given by "$A_1 = ?$" in a *concrete* version of T_1 must be extended to an encoding of "$T_1 \to A_1 = ?; A_2 = ?$" before computing the `union` operation. In the next section, we present a simple encoding that enables such coercion. □

4 Concrete Relational Databases and SQL^{path}

For a given $\mathcal{C}_{\mathrm{AR}}$ schema Σ, an identity resolving referring type assignment can serve as a basis to encoding elements of `OID` with *sequences of values* for concrete attributes that can serve as surrogate keys for the values. We now present such an encoding, Rep, and show how it leads, in turn, to a concrete relational database for Σ, and finally to SQL queries over this schema that implement SQL^{path} queries. This "closes the loop" on our overall objective for a separation of concerns.

On Concrete Representation of Referring Expressions

Definition 11 (Rep) Let T and Rt be an abstract table and referring expression type, where $\mathsf{Fix}(Rt, T)$ is given by

$$T_1 \to (\mathsf{Pf}_{1,1} = ?, \ldots, \mathsf{Pf}_{1,k_1} = ?); \ldots; T_k \to (\mathsf{Pf}_{k,1} = ?, \ldots, \mathsf{Pf}_{k,k_k} = ?),$$

and let $D_{i,j}$ be the underlying concrete domain for the final attribute in each $\mathsf{Pf}_{i,j}$. Also let $\mathsf{Nm}(\mathsf{Pf})$, where $\mathsf{Pf} = A_1. \cdots .A_\ell$, denote a new attribute name "A_1-...-A_ℓ". We write $\mathsf{Rep}(Rt, T)$ to denote the sequence of concrete attributes

$$(\texttt{disc enum}\{`T_1\text{'}, \ldots, `T_k\text{'}\}, \mathsf{Nm}(\mathsf{Pf}_{1,1})\ D_{1,1}, \ldots, \mathsf{Nm}(\mathsf{Pf}_{k,k_k})\ D_{k,k_k}).$$

If Rt consists of a single component, then attribute `disc` is excluded. □

Note that Rep uses an auxiliary Nm function to invent new attributes names simply by replacing "dots" by "dashes".[7] The following example now illustrates how Rep can be used to encode abstract values occurring in abstract tables:

Example 12 Consider the SQL^{path} query (5) over schema (4) above (see Examples 10 and 6), and assume $\mathsf{RTA}(T_1)$ and $\mathsf{RTA}(T_2)$ are given respectively by "$A_1 = ?$" and by "$T_1 \to A_1 = ?; A_2 = ?$". Then $\mathsf{Rep}(\mathsf{RTA}(T_1))$ and $\mathsf{Rep}(\mathsf{RTA}(T_2))$ are given respectively by "(A_1 `STRING`)" and by

$$\text{"}(\texttt{disc enum}\{`T_1\text{'}, `T_2\text{'}\}, A_1 \texttt{ STRING}, A_2 \texttt{ STRING})\text{"}.$$

Now consider where: T_1 has object e_1 with A_1 = `'abc'`, T_2 has object e_2 with A_2 = `'bcd'`, and both T_1 and T_2 have object e_3 with A_1 = `'cde'` and A_2 =

[7] Other options for both Nm and Rep are clearly possible, e.g., based on introducing *variant* record types.

`'def'`. Referring expressions for each e_i would then be encoded as follows:[8]

e_1 : (`'abc'`),
e_2 : (`'`T_2`'`, $\langle \mathit{defaultSTRINGvalue} \rangle$, `'bcd'`) and
e_3 : (`'`T_1`'`, `'cde'`, `'def'`).

To compute the union operator, coercion for concrete representations of referring expressions is necessary. In this case, value sequences encoding references to e_i will need to be augmented with additional values to conform to $\mathsf{Rep}(\mathsf{RTA}(T_1) \triangledown \mathsf{RTA}(T_2))$, which matches $\mathsf{Rep}(\mathsf{RTA}(T_2))$ (see Example 10). Thus, the encoding of the referring expression for e_1 is extended to

$$(\texttt{'}T_1\texttt{'},\ \texttt{'abc'},\ \langle \mathit{defaultSTRINGvalue} \rangle)$$

prior to evaluating `union`. □

As the example illustrates, extending our coercion operator for referring expression types to their concrete representations is straightforward. In particular, to coerce a $\mathsf{Rep}(Rt_1, T_1)$ tuple to a $\mathsf{Rep}(Rt_2, T_2)$ tuple, where Rt_2 is a *referring supertype* of Rt_1, it suffices to create the $\mathsf{Rep}(Rt_2, T_2)$ tuple by using the values from the $\mathsf{Rep}(Rt_1, T_1)$ tuple for the concrete attributes corresponding to common components, and to assign default values to the remaining columns. We denote this function by $\mathsf{Coerce}_{(Rt_1,T_1)}^{(Rt_2,T_2)}$. To convert a $\mathsf{Rep}(Rt_2, T_2)$ tuple back to a $\mathsf{Rep}(Rt_1, T_1)$ tuple, we first check if the value of the `disc` attribute corresponds to a guard for some component of Rt_1 and, if so, we project the former to attributes in $\mathsf{Rep}(Rt_1, T_1)$; the conversion is undefined otherwise. We denote this function by $\mathsf{Restrict}_{(Rt_2,T_2)}^{(Rt_1,T_1)}$. Note that both functions can be expressed using SQL query constructs, e.g., by using constant expressions in a `select` clause in the case of the Coerce function.

To simplify notation, we extend the Coerce and Restrict functions to tuples by applying the functions component-wise (assuming values of concrete attributes map to themselves), and omit mention of T_i to improve readability.

The main consequence of a referring expression type assignment RTA that is identity resolving and of Rep, our suggestion for a concrete encoding of referring expressions, is that we now have a way to compare possibly different representations of an `OID` value:

Lemma 13 Let RTA be an identity resolving type assignment for Σ and $\mathcal{R}$ a set of referring expression types such that $\{\mathsf{RTA}(T) \mid T \in \mathsf{Tables}(\Sigma)\} \subseteq \mathcal{R}$ and such that $\mathcal{R}$ is closed under $\triangledown$. Then, for every $e_1, e_2 \in \texttt{OID}$, if $\mathsf{Rep}(Rt_1, T_1)$ tuple t_1 and $\mathsf{Rep}(Rt_2, T_2)$ tuple t_2 are concrete representations of referring expressions to e_1 and e_2, respectively, then $e_1 = e_2$ if and only if

$$\mathsf{Coerce}_{Rt_1}^{Rt_1 \triangledown Rt_2}(t_1) = \mathsf{Coerce}_{Rt_2}^{Rt_1 \triangledown Rt_2}(t_2).$$

□

The Coerce and Restrict functions are naturally extended to *abstract attributes* to produce a list of column names of the representation, to *abstract tuples* that may contain several abstract identifiers (we assume that concrete attributes are

[8] We assume a non-null default value exists for each concrete domain.

represented by themselves), and, in turn, to abstract database instances and query answers.

On Concrete Relational Schemata
With Rep, it is straightforward to define a *concrete relational schema* corresponding to a given $\mathcal{C}_{\mathrm{AR}}$ schema (such as "`table PERSON (`$\cdots$`)`" given in our introduction):

Definition 14 (Mapping Abstract $\mathcal{C}_{\mathrm{AR}}$ Tables to Concrete Relations) Let $T \in \Sigma$ be an abstract table and RTA an identity resolving referring type assignment for Σ. The *concrete relational table schema* $\mathsf{Tab}(T)$ for T is obtained by appending attributes and constraints to an initially empty sequence s, in "`table` T `(`s`)`", in the order they occur in T according to the following:

1. For attribute "`self OID`", append columns $\mathsf{Rep}(\mathsf{RTA}(T), T)$ together with a primary key constraint consisting of these columns.
2. For a concrete attribute "A D", append same if not already included.
3. For an abstract attribute "A `OID`", append columns $\mathsf{Rep}(\mathsf{RTA}(\mathsf{Dom}(A), T)$ after renaming each column by prefixing with "A`-`". Also add a foreign key constraint from these columns to $\mathsf{Tab}((\mathsf{Dom}(A)))$.
4. For a constraint φ, add a general assertion constraint if φ is not the first occurrence of a foreign key constraint for some attribute.[9] $\square$

In addition, whenever multiple foreign keys are declared on T for *the same* attribute A, to enforce database integrity we need either to add general assertion constraints that verify the presence of the appropriate value in the other referenced tables. Alternatively, one may add additional columns (using Rep repeatedly) and then enforce integrity locally.

Along similar lines, Rep can also be extended, with respect to a referring type assignment RTA, to *instances* of abstract tables and to bindings of abstract values to variables in queries. We write $\mathsf{Rep}_{\mathsf{RTA}}$ to denote this extension.

On Query Translation
To summarize, relational operations on the concrete representation of referring expressions, in particular equality comparisons, requires that *compatibility* issues between referring expression types that can potentially refer to the same value in `OID` must be addressed. In our setting, Lemma 13 ensures this for Rep in the case of an identity resolving type assignment. Hence, to translate a $\mathrm{SQL}^{path}Q$ to concrete SQL, we can apply rewritings to Q to ensure all terms have the form "$x.A$", for some abstract or concrete attribute A (see Section 2 for more details), and then apply Map, a recursive procedure:

Definition 15 (Query Compilation) Let Q be a SQL^{path} query over the abstract schema Σ, and RTA an identity resolving type assignment for Σ. We define Map, a function that maps Q to concrete SQL by induction on the structure of

[9] An assertion can be replaced by a foreign key constraint when φ is an `isa` constraint to an abstract table T' for which $\mathsf{RTA}(T') = \mathsf{RTA}(T)$.

Q, as follows:

$$
\begin{aligned}
\mathsf{Map}(T\ x) &\mapsto \mathsf{Tab}(T)\ x \\
\mathsf{Map}(\texttt{select}\ x_1.A_1,\ldots,x_k.A_k\ Q) &\mapsto \texttt{select}\ \mathsf{Rep}_{\mathsf{RTA}}(x_1.A_1),\ldots,\mathsf{Rep}_{\mathsf{RTA}}(x_k.A_k)\ \mathsf{Map}(Q) \\
\mathsf{Map}(\texttt{from}\ Q_1,Q_2) &\mapsto \texttt{from}\ \mathsf{Map}(Q_1),\mathsf{Map}(Q_2) \\
\mathsf{Map}(Q\ \texttt{where}\ x_1.A_1 = x_2.A_2) &\mapsto Q\ \texttt{where} \\
\mathsf{Coerce}^{Rt(x_1.A_1)\triangledown\, Rt(x_2.A_2)}_{Rt(x_1.A_1)}(\mathsf{Rep}_{\mathsf{RTA}}(x_1.A_1)) &= \mathsf{Coerce}^{Rt(x_1.A_1)\triangledown\, Rt(x_2.A_2)}_{Rt(x_1.A_1)}(\mathsf{Rep}_{\mathsf{RTA}}(x_2.A_2)) \\
\mathsf{Map}(Q_1\ \texttt{union}\ Q_2) &\mapsto \mathsf{Coerce}^{Rt(Q_1\triangledown Q_2)}_{Rt(Q_1)}(\mathsf{Map}(Q_1)) \\
&\qquad\qquad\qquad \texttt{union}\ \mathsf{Coerce}^{Rt(Q_1\triangledown Q_2)}_{Rt(Q_2)}(\mathsf{Map}(Q_2)) \\
\mathsf{Map}(Q_1\ \texttt{minus}\ Q_2) &\mapsto \mathsf{Restrict}^{Rt(Q_1)}_{Rt(Q_1\triangledown Q_2)}(\mathsf{Coerce}^{Rt(Q_1\triangledown Q_2)}_{Rt(Q_1)}(\mathsf{Map}(Q_1)) \\
&\qquad\qquad\qquad \texttt{minus}\ \mathsf{Coerce}^{Rt(Q_1\triangledown Q_2)}_{Rt(Q_2)}(\mathsf{Map}(Q_2)))
\end{aligned}
$$

Note that $Rt(\cdot)$ denotes the referring types assigned to variables in answer tuples (or, by mild abuse of notation, to all components of a tuple), and also that *equality comparisons* on $\mathsf{Rep}(\cdot)$ are performed component-wise when needed. (The type assignments originate from RTA and "$T\ x$" operators, and are preserved through the query save for `union` operators that convert variables to least common referring supertypes with respect to the corresponding referring types in Q_1 and Q_2.) □

Observe that the definition of Map is purely syntactic and produces a concrete SQL query for which the following, our main result, applies:

Theorem 16 Let Σ be a $\mathcal{C}_{\mathrm{AR}}$ schema and let RTA an identity resolving type assignment for Σ. For any SQL^{path} query Q over Σ and every database instance I of Σ:

$$\mathsf{Rep}_{\mathsf{RTA}}(Q(I)) = (\mathsf{Map}(Q))(\mathsf{Rep}_{\mathsf{RTA}}(I)).$$ □

Example 17 Applying Map to the SQL^{path} query (1) from Section 2 then yields the following in SQL when RTA(`PERSON`) and RTA(`VEHICLE`) are respectively given by "`ssn=?`" and "`vin=?`":

```
select p.name from CAN-DRIVE d, PERSON p, VEHICLE v
where v.make = 'Ford' and d.driven-vin = v.vin and d.driver-ssn = p.ssn.
```
□

5 Summary and Future Work

This paper was motivated by two problems that seem to inhere in relational DBMS: (*i*) the need to prematurely commit to an "external key" (printable values) in designing relational schemas; and (*ii*) the need to choose a *single and simple way* to refer to all entities/tuples in a class/table rather than allow for variations.

To help with this, we proposed a simple semantic data model $\mathcal{C}$, where naming is not an issue because objects have identity; and a simple extension of SQL, SQL^{path}, which allows implicit foreign key joins in the form of "path expressions" such as `v.owner.name`.[10] In the hope of making SQL programmers more

[10] We emphasize that neither of these are novel ideas: they have been present in database semantic models since Taxis [4], and GEM [7].

comfortable, we turned $\mathcal{C}$ into $\mathcal{C}_{\text{AR}}$, a more relational-like version, where object surrogates are visible as columns in tables (but will not be stored).

Orthogonally to schema and query specification, analysts can specify uniqueness constraints in the form of path functional dependencies, and especially describe complex *preferred naming schemes* for each class/table in the abstract schema. The language for preferred naming schemes allows us to solve the problems raised in the beginning, such as having different naming schemes for subclasses than for superclasses, and not having to invent new names for generalizations.

We emphasize that we view the separation of concerns between naming and schema/query body specification to be a central contribution of this work.

To support this, we provided ways to verify that the naming schemes are indeed unique, based on the dependencies specified, and algorithms for converting the abstract schema and queries into ordinary SQL table declarations and queries, where object (identifiers) are no longer visible.

There are a number of interesting problems that remain to be investigated. One issue is how to help relieve analysts from the burden of having to write complex referring expressions for *every* class in the schema. One could start with default rules: a single key `k` for class `C` results in type expression `k=?`, which is inherited to all subclasses of `C` that do not specify keys. More interesting is the case of subclasses, like `FamousPerson` of `Person`, which do specify something different. The theory of default inheritance in AI would suggest that the most specific rule apply to each class. Using ideas from [1], one could then automatically generate referring expression types combining the two, e.g., the type "`FAMOUS-PERSON` $\rightarrow$ `starName ; ssn`" for `PERSON`. This approach would also have the advantage of propagating local changes and additions — an important software engineering property. Other directions for work include a richer language for referring expression types, alternatives to the concrete representation, (including support for alternative mappings of class hierarchies to relational tables, which are mentioned in many textbooks).

References

1. Alexander Borgida. Modeling class hierarchies with contradictions. In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988.*, pages 434–443. ACM Press, 1988.
2. Alexander Borgida, David Toman, and Grant Weddell. On referring expressions in query answering over first order knowledge bases. In *Principles of Knowledge Representation and Reasoning*, 2016. (in press).
3. Richard Hull and Roger King. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv.*, 19(3):201–260, 1987.
4. John Mylopoulos, Philip A. Bernstein, and Harry K. T. Wong. A language facility for designing database-intensive applications. *ACM Trans. Database Syst.*, 5(2):185–207, 1980.
5. David Toman and Grant Weddell. On Attributes, Roles, and Dependencies in Description Logics and the Ackermann Case of the Decision Problem. In *Description Logics 2001*, pages 76–85. CEUR-WS vol.49, 2001.

6. David Toman and Grant E. Weddell. On adding inverse features to the description logic $\mathcal{CFD}^{\forall}_{nc}$. In *PRICAI 2014: Trends in Artificial Intelligence - 13th Pacific Rim International Conference on Artificial Intelligence, Gold Coast, QLD, Australia*, pages 587–599, 2014.
7. Carlo Zaniolo. The database language GEM. In David J. DeWitt and Georges Gardarin, editors, *SIGMOD'83, Proceedings of Annual Meeting, San Jose, California, May 23-26, 1983.*, pages 207–218. ACM Press, 1983.