The TSIMMIS Approach to Mediation: Data Models and Languages (Extended Abstract)

Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass,
Anand Rajaraman, Yehoshua Sagiv, Jeffrey Ullman, Jennifer Widom
Stanford University, Stanford CA 94305 USA

ABSTRACT

TSIMMIS — The Stanford-IBM Manager of Multiple Information Sources — is a system for integrating information. It offers a data model and a common query language that are designed to support the combining of information from many different sources. It also offers tools for generating automatically the components that are needed to build systems for integrating information. In this paper we shall discuss the principal architectural features and their rationale.

1. Mediators

The mediator architecture (Wiederhold [1992]) is one of several that have been proposed to deal with the problem of *integration of heterogeneous information.* Even as simple a concept as the employees of a single corporation may be represented by many different databases. These databases are "heterogeneous" on many levels.

- Some may be relational, others not. Some may not be databases at all, but spreadsheet files or raw text files.
- The types of data may vary; a salary could be stored as an integer or a character string.
- The underlying units may vary; salaries could be stored on a per-hour or per-month basis, for example.
- The underlying concepts may differ in subtle ways. A payroll database may not regard a retiree as an "employee," while the benefits department does. Conversely, the payroll department may include consultants in the concept "employee" while the benefits department does not.

One way to integrate different information sources that deal with employees is to create a *mediator*, which is a facility capable of answering queries about employees. The mediator uses the raw sources such as personnel records to answer the queries. The mediator may also use other mediators as intermediaries, as suggested in Fig. 1.

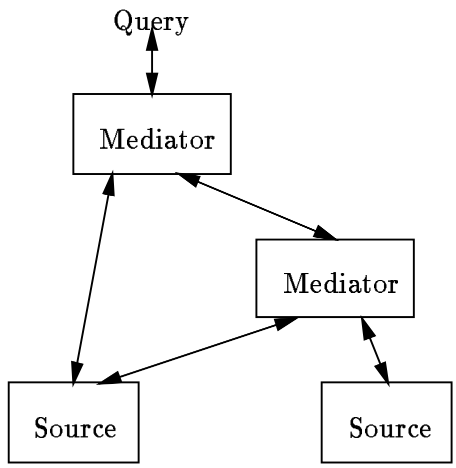

Fig. 1. A network of mediators and information sources.

1.1. Mediator Requirements

Our goal is to make it easy to build mediator networks as suggested by Fig. 1. We see several requirements of a mediator architecture.

Y. Sagiv's permanent address: Dept. of CS, Hebrew Univ., Jerusalem. Work supported by USAF contract F33615–93-1-1339, NSF grant IRI–92–23405, BSF grant 92–00360, the Anderson faculty scholar fund, and equipment grants of DEC and IBM.

1. There must be a common data model that is more flexible than the models commonly used for database management systems. A mediator model must support:
   a) A rich collection of structures, including nested structures as is found in the type system of typical modern programming languages.
   b) Graceful handling of missing information or related information of widely differing structures.
   c) *Meta-information*, that is, information about the structures themselves and about the meanings of the terms used in the data.
2. There must be a common query language to allow
   a) New mediators to join old ones for augmented functionality and
   b) New sources to provide input to an existing mediator.
3. There must be tools to make the creation of new mediators and mediator systems easier than would be the case if everything were built from scratch.

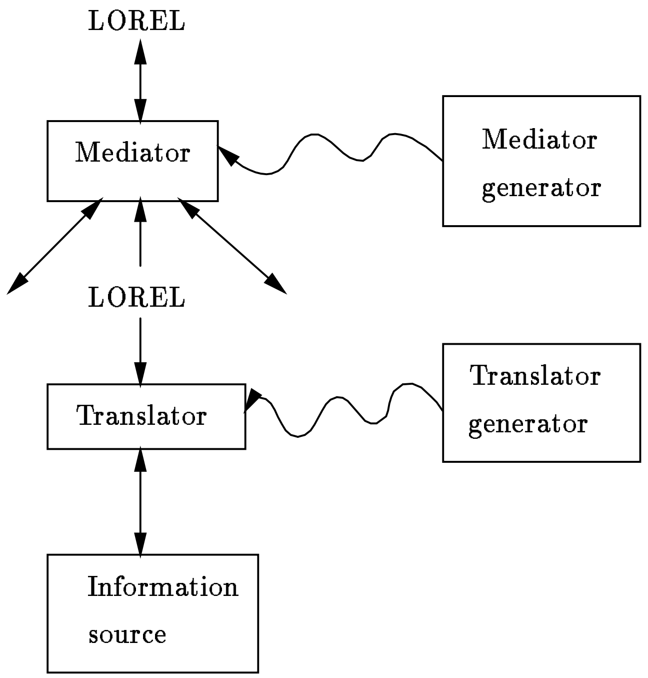

Fig. 2. The components of TSIMMIS.

2. The TSIMMIS Mediation System

The principal components of TSIMMIS are suggested in Fig. 2. We use:

- A *"lightweight" object model* called OEM (Object-Exchange Model) serves to convey information among components. It is "lightweight" because it does not require strong typing of its objects and is flexible in other ways that address desideratum (1) above. We shall say more about OEM in Section 3.
- A *common query language* links components. This language *LOREL* ("Lightweight Object REpository Language") is roughly "SQL with dots" to allow following substructures in OEM objects. We discuss LOREL in Section 4.
- *Translators* ("wrappers") allow LOREL queries to be converted into source-specific queries. We do not assume sources are databases, and it is an important goal of the project to cope with radically different information formats in a uniform way.
- *Mediators* talk LOREL both to ask and answer queries.
- Translator- and mediator-generators. We are in the process of developing methodologies for generating classes of translators and mediators automatically from simple descriptions of their functions. We discuss mediators, translators and their automatic generation in Section 5.

For a general discussion of TSIMMIS, including components such as constraint management and user interfaces not described here, see Chawathe et al. [1994].

3. The Object-Exchange Model

The Object-Exchange Model (OEM) is described in Papakonstantinou, Garcia, and Widom [1995]. Data represented in OEM is *self-describing*, in the sense that it can be parsed without reference to an external schema. This capability simplifies the intefaces among TSIMMIS components. A second feature of OEM is flexibility in data organization, both in the structures that can be described and in the tolerance of alternative forms for "similar" objects, differences in terminology, and differences in the kinds of information obtainable from "similar" sources. It is possible to see OEM as "object-oriented," in the sense that the fundamental OEM concept is an "object." However, the type system of OEM is quite elementary. Objects have a single data component, which is either an atomic value (e.g., an integer) or a set of objects. With these primitives, it is possible to simulate all the structures that are found in more conventional object-oriented type systems. For example, record structures are simulated by sets of objects, each of whose labels names a field of the structure. More significantly, objects are not organized into classes. A query will address all objects whose structure matches the conditions of the query.

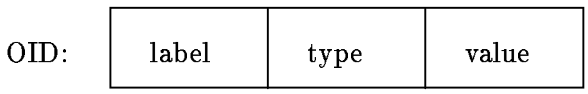

Fig. 3. An OEM object.

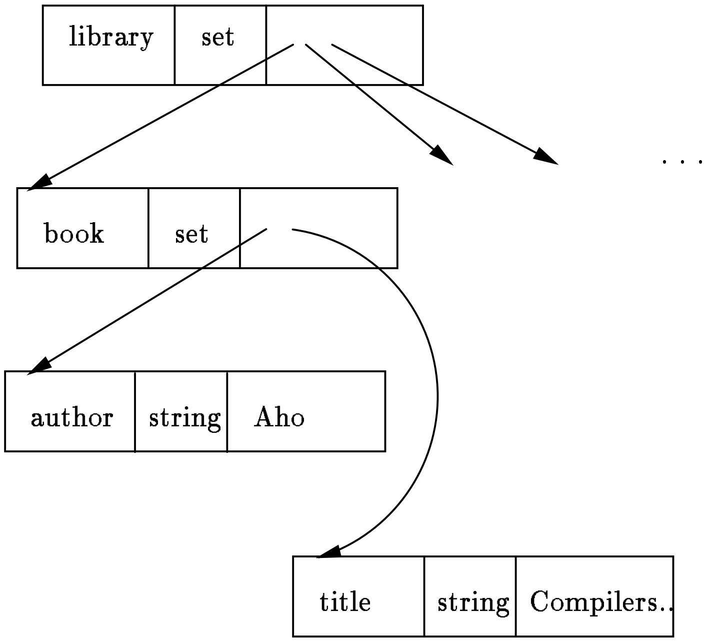

Fig. 4. A collection of OEM objects.

3.1. OEM Objects

As suggested in Fig. 3, OEM objects have four components:

1. *Object ID.* It may be constructed by the mediators to be an expression describing where the object came from. It may also be a pointer to an object in the workspace used to answer the query. Unlike object-oriented database systems, OEM object-ID's may be local to a query. They need not be persistent, and they are optional rather than required.

2. *Label* tells what the object represents (roughly, its class). However, there is no "schema." Objects with a given label are not required to have a particular set of subobjects. Labels are expected to have human-understandable definitions that may be retrieved easily by the user. Thus, labels carry all the information there is about objects, which is why we refer to OEM as "self-describing."

3. *Type* of its value, either *set* or an atomic type like *string*.

4. *Value*, either an atomic value or a set of object references.

Example 1: Figure 4 shows a collection of OEM objects. At the top is a *root* object whose label is `library`. Its value is a set of objects, so its type is *set*. Among the set of objects that form the value of the library, we see one, labeled `book`. There is no reason to suppose that all objects in the library have label `book`. There may be objects labeled `report`, or `video`, for example. In the future we may find objects with labels of a kind we cannot now imagine.

The `book` object has a value whose type is also *set*. However, unlike the `library` object, the "set" here is used to simulate a record structure. We expect to find as the value of the `book` object a set of subobjects with different labels, each representing one of the "fields" of the "structure." We have shown a subobject labeled `author` and a subobject labeled `title`, each with a string-type atomic value. Yet there could be other subobjects, for example an object labeled `postscript` whose atomic value is a postscript image of the book. There could be other `author`-labeled subobjects, if that was appropriate. There is no "schema" for books, and we would expect in practice that a library composed of information from a variety of sources would have books with many different sets of "fields" in their structure. □

3.2. OEM as a Logical Data Model

While it is natural to think of OEM as an object-oriented model, and it indeed provides some of the advantages, such as the variety of types, found in object-oriented models, it is also useful to see OEM as a form of first-order logic. In this logic, labels are predicates, and they relate object identifiers to other object identifiers or to atomic values, in the same way that the objects themselves do.

Example 2: The root `library` object in Example 1 may be thought of as a predicate $library(B)$. Its value is the set of object ID's for all the books and other objects that are members of the value of the `library` object.

Similarly, corresponding to the label `book` will be a predicate $book(B, X)$ whose value is a set of pairs (b, x). Here, b is the object ID of a `book` object, and x is the object ID of an object in the set that is the value of the object with ID b. For instance, if $b1$ is the object ID of the `book` object shown in Fig. 4, and $a1$ and $t1$ are the object ID's of the `author` and `title` subobjects shown in that figure, then $book(b1, a1)$ and $book(b1, t1)$ are true. Put another way, $(b1, a1)$ and $(b1, t1)$ are members of the relation that is the value of predicate *book*.

We might also expect from Fig. 4 that there is a predicate $title(T, S)$ whose value is a set of pairs (t, s). Here, T is the object ID of an object with label `title`, and s is a string, the title that appears as the value of the object t. However, it is important to remember that OEM does not enforce a schema. Thus, it is permissible if the *title* predicate also contains pairs (t, s) corresponding to `title` objects t whose value is a set. In that case, s would not be a string but an object ID for one of the objects that is a member of the set that is the value of object t. □

4. The LOREL Query Language

LOREL (Lightweight Object REpository Language) is intended to serve as a language for carrying queries from one component to another, as was suggested by Fig. 2. The style of LOREL is common for languages that try to act as very-high-level query languages in an object-oriented framework. We may think of it superficially as "SQL with dots," in a tradition that extends from GEM (Zaniolo [1983]) through more recent object-oriented query languages such as OSQL (Fishman et al. [1989]), XSQL (Kifer, Kim, and Sagiv [1992]), or OQL (Cattell et al. [1994]).

A complete specification of LOREL, including formal syntax and semantics is found in Quass, Rajaraman, Sagiv, Ullman, and Widom [1994]. Here, we shall give only an example that suggests the flavor of the language. It is based on the object structure suggested by Fig. 4, where there was only one root object, the one labeled `library`.

Example 3: "Find the books of which Aho is an author."

```
select library.book.title
where library.book.author = "Aho"
```

This query applies to the root object labeled `library`. If there were more than one root object available, we could specify the root or roots in a from-clause

```
from library
```

that could appear between the select- and where-clauses as in SQL.

We answer this query by looking for paths in the object structure that follow the *path expressions* of the query, which are the sequences of labels connected by dots. For example, in Fig. 4 there is one path that has the sequence of labels `library`, `book`, `author`, although in practice there would be many. We can also find a path labeled `library`, `book`, `title` in Fig. 4, and because the common prefixes `library` and `library.book` match the same objects, these two paths together form an *object assignment* that matches the path expressions of the query. Fortunately, the condition of the where-clause, that the author be "Aho," are satisfied by the object assignment. Thus, the `title` object at the end of the path corresponding to the select-clause becomes part of the answer. Technically, a copy of this `title` object becomes a subobject of an object with label `answer` that is produced by the query. □

4.1. The Importance of Partial Object-Assignments

It is interesting to explore one of the major design decisions of LOREL that differs from the corresponding approach in SQL. We believe the LOREL approach is correct for languages that serve as information integrators, although the SQL approach may be fine for conventional query languages.

The reader may be aware of a bug/feature of

SQL dealing with the way OR is handled in where-clauses. Suppose we have three unary relations R, S, and T, and we wish to compute $R \cap (S \cup T)$. Supposing each of these relations have a single attribute A, we might expect the following SQL query to do the trick.

```
select R.A
from R, S, T
where R.A = S.A or R.A = T.A;
```

Unfortunately, if T is empty, the result is empty, even if there are elements in $R \cap S$. The reason is that SQL semantics requires a *total assignment* of tuples to the three relations R, S, and T mentioned in the from-clause. If T (or S) is empty, we cannot find such a total assignment, and thus there is no way to produce an answer.

We would expect intuitively that a *partial assignment* of tuples would be adequate. For example, if T is empty, but R and S each contain the tuple (0), we would like to assign this tuple to both R and S, assign nothing to T, and find that the where-condition is satisfied since $R.A = S.A$. We could then produce $R.A$, that is, (0), as one of the answer tuples.

The distinction is not important in SQL, because there is unlikely to be an empty relation in a conventional relational database. However, the analogous situation in mediated databases is much more likely.

Example 4: Here is a LOREL query about the library suggested by Fig. 4. "Find all books written by Aho or having subject compilers."

```
select library.book.title
where library.book.author = "Aho" or
    library.book.subject = "compilers"
```

If a book, such as the one suggested by Fig. 4 does not have a `subject` subobject,[1] then there is no way to create an object assignment that matches all three path expressions in the above query. Thus, the book suggested in Fig. 4 would not be an answer, even though it intuitively matches the query.

Fortunately, LOREL requires only a partial match. If we match the path expressions `library.book.author` and `library.book.title` to paths in Fig. 4 exactly as we did in Example 3, that is enough to satisfy with where-condition. Thus, we produce the title `"Compilers..."` as one of the answers to the above LOREL query. □

[1] Do not confuse the *title*, which begins with "Compilers," with a *subject*, which does not exist in the figure.

5. Translators, Mediators, and Their Generators

A central goal of TSIMMIS is to make it easy to produce mediated systems. One support for this goal is the common data model and language which are designed to make it easy to integrate information from related but not-quite-the-same information sources. Another support is tools for generating translator and mediator components automatically. We are just beginning to develop these tools and experience in their use.

5.1. Translators

We do not assume that information sources have SQL or similar capability. Sources may be text files, data organized into tables, spreadsheet files, or most any other format. A source may also be a query or command system, such as a bibliographic search system. Any translator accepts queries expressed in LOREL. However, translators differ not only in the set of labels that they can deal with, but in the set of queries involving those labels that they will accept.

We may be used to a query language like SQL, where in any database, any SQL query can be answered as long as all relations it refers to are in the database. That property does not hold for many sources that are not relational databases. For example, a bibliographic search system may "understand" concepts like author, title, and publication date. The system may respond to queries asking for the titles written by a certain author, or the publication dates of those books, yet be unable or unwilling to respond to a query asking for all titles published in 1989, for example.

As a consequence, we see a translator as a system that takes queries and decides first of all whether or not its underlying source can answer the query. If so, it turns the query into something that the underlying source can respond to. The translator then converts the answer into the appropriate OEM objects and returns this result.

5.2. Translator Generation

There exists at the present a rule-based tool for generating translators (Papakonstantinou [1994]).

This translator-generator takes a set of rules of the form:

regular-expression pattern => action

It also takes user-written functions that are needed to connect the translator to the source and execute queries on that source.

Example 5: Suppose we have a `bib` (a UNIX bibliographic format) file as a source. We might define rules such as:

```
    select library.book.title
    where library.book.author = $1
=>
    Code that searches for an .AU line with $1
    and returns the associated .TI line
```

The generated translator would examine a query and compare it to this and other patterns that were given in the file from which the translator was generated. If the query matched the pattern, with some string in place of the parameter `$1`, then the associated action would be executed, with that string in place of the parameter. Thus, the query

```
select library.book.title
where library.book.author = "Aho"
```

would cause the `bib` file to be examined for `.AU` lines containing the substring `"Aho"`. □

A more ambitious system for generating translators is described in Papakonstantinou, Gupta, Garcia, and Ullman [1995].

5.3. Extending Translation Rules

If the query language is logic-based, or can be interpreted in first-order logic as can most of the features of LOREL, then there is the opportunity to answer certain queries that match no pattern. A query can be deduced to be logically equivalent to a query pattern with a rule or to the combination of several queries whose patterns appear in rules. As a simple example, if a query pattern has a where-condition that is the AND of two subconditions, then a query with the order of those subconditions reversed can also be answered.

Rajaraman, Sagiv, and Ullman [1994], examines the problem of deducing when a given query is equivalent to a sequence of queries that match the patterns of rules. An example will suggest what can be achieved.

Example 6: Suppose the information source is a genealogy in some form, and the only two query patterns it can answer are:

1. Given any individual C, find C's parents.
2. Find all the individuals who have parents specified by the information source.

There are several reasons why the source might be limited in this way. For example, it could be a relational database with a child-parent relation and an index only on the child component.

However, given these two query patterns, we can also handle the query "Find the grandparents of individual a" by:

i) Find the set P of parents of a, using query (1).

ii) For each individual p in set P, use query (1) to find the parents of p.

iii) The answer is the union of all the individuals found in step (*ii*).

This sequence of queries is suggested in Fig. 5.

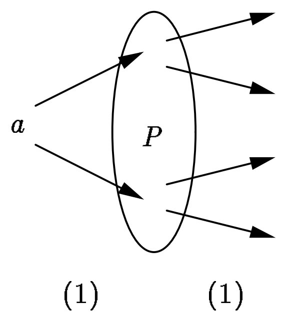

Fig. 5. Finding an individual's grandparents.

Now, consider the query: "Find the grandchildren of individual g." We need query (2) as well as (1).

i) Use (2) to find all individuals.

ii) Use (1) to find the parents of the individuals found in (*i*).

iii) Use (1) to find the parents of the individuals found in (*ii*).

iv) Find those individuals whose grandparents are found in (*iii*) to be individual g.

The sequence of queries is suggested by Fig. 6. It is a very expensive strategy but is the best that can be done given the queries that the source can answer. □

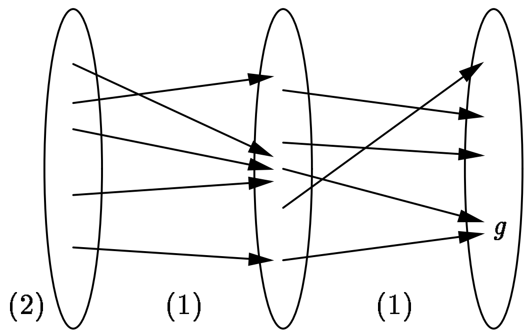

Fig. 6. Finding an individual's grandchildren.

5.4. Mediators

Mediators are in some ways similar to translators. They accept queries that refer to labels in some set of labels known to the mediator, and they may not accept all possible queries about those labels. The query is translated into one or more queries that are executed by the source or sources below. These sources can be either the translator of a raw information source, or another mediator. Thus, mediator generation can be similar to translator generation, with the principal exception that the action associated with a pattern may consist of a sequence of queries, and the source at which each is executed must be specified.

However, there are certain kinds of mediators that seem especially important, and it is possible to develop special-purpose, simple tools for generating mediators of these kinds. For example,

- A *union mediator* takes queries and passes them unchanged to two or more other sources. It combines the answers of each into one.
- A *join mediator* creates a view of two other sources (possibly other mediators). The view contains an object for each pair of objects, one from each of the two sources, that agree in certain values.

Example 7: A medical advisor could combine information from various drug companies about dosages (summarized in a union mediator) with patient records giving age and weight of a patient. This mediator system is suggested by Fig. 7. The dosage mediator is expected to answer a query in which a patient and a drug are specified and the proper dosage is requested. This operation is effectively a join of the dosage and patient information, with equality on the age and weight.

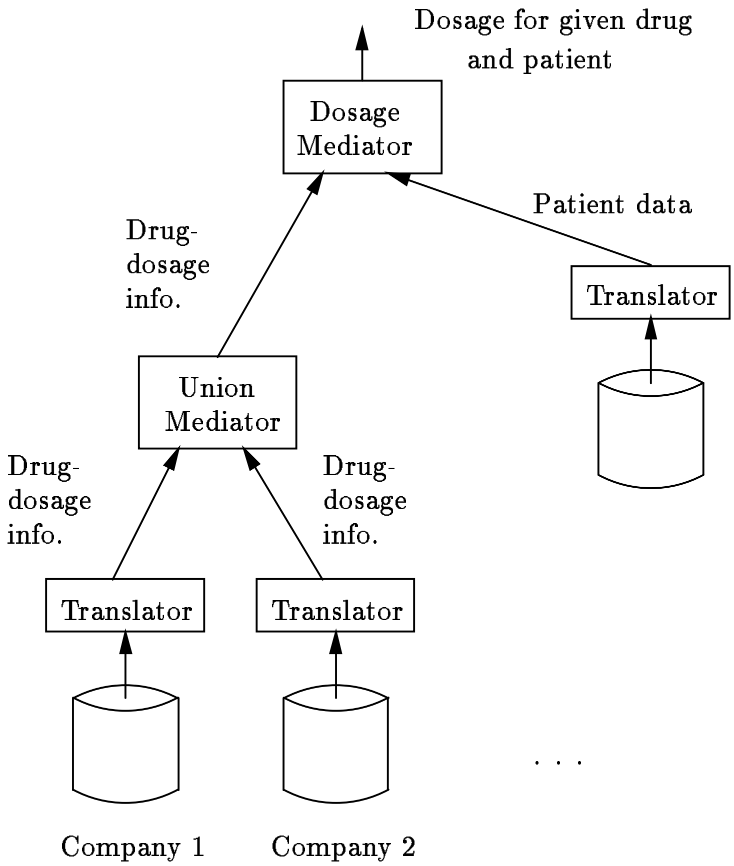

Fig. 7. A mediator system for dosage advice.

Such a query might be answered as follows. The mediator first asks the patient information about the age and weight of the patient.[2] Given the age and weight information, the dosage mediator then asks the union mediator for drug companies about the appropriate dosage of the given drug for the patient's age and weight. This query is passed to each of the drug companies' databases, and presumably only the company that manufactures the drug is able to respond. However the union mediator returns whatever information it obtains from below, perhaps modifying the label to indicate the source, especially if two companies give contradictory recommendations. □

5.5. Mediator Generation

We are in the process of developing a logical lan-

[2] In this simple example we show a single database of patient information, but in practice this "database" would be formed by a mediator, perhaps a union mediator, that got its information from many databases at many different clinics and hospitals.

guage for describing mediators. To describe a mediator, one gives logical rules that define the OEM objects that the mediator makes available in a "view." Queries to the mediator refer to objects in this view. The action portion of the rules describes an object or objects at the source(s) that must exist for the defined object to appear in the mediator's view. The language under development offers two important capabilities:

- The language allows recursive definition of view objects. The following example suggests why that capability might be important.

Example 8: Suppose we have a source that is the arcs of a graph, and we wish to query about paths in that graph. We cannot express "path" in LOREL. However, we could define a mediator that supports a view that is the set of path objects justified by the graph. We could then ask the mediator a simple LOREL query about whether there is a path from node a to node b. The mediator would issue to the source as many queries as necessary to answer the query. Note that this approach allows the problem of computing transitive closures efficiently to be solved once, when the mediator is generated, rather than each time a query is asked. □

- The language allows object ID's to be computed as logical terms, as in F-logic (Kifer and Lausen [1989]). The following example illustrates how this capability might be used to advantage.

Example 9: One use of this capability is to give objects a description of how they were derived from sources. Another important use of this feature is in describing views in which source objects are grouped into sets. For example, suppose we have a source of "enrollment" objects, that is, student-course pairs. We could create a view with objects labeled `course` and with a value equal to the set of students enrolled in that course. The rules describing this view might define for each course c an "object identifier" $course(c)$. Then, for each student-course pair (s, c) in the enrollment source, we would create a view-object with label `student` and value equal to s. We then require that in the view this object is a member of the set that is the value of the course view-object with ID $course(c)$. □

References

Cattell, R. G. G. et al. [1994]. *The Object Database Standard: ODMG–93*, Morgan-Kaufmann, San Mateo.

Chawathe S., H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom [1994]. "The TSIMMIS project: integration of heterogeneous information sources," IPSJ Conference, Tokyo, 1994. Available by anon. ftp as `pub/chawathe/1994/tsimmis-overview.ps` from `db.stanford.edu`.

Fishman, D. et al. [1989]. "Overview of the Iris DBMS," in *Object-Oriented Concepts, Languages, and Applications* (W. Kim and F. H. Lochovsky, eds.), pp. 219–250, Addison-Wesley, Reading, Mass.

Kifer, M., W. Kim, and Y. Sagiv [1992]. "Querying object-oriented databases," *ACM SIGMOD International Conf. on Management of Data*, pp. 393–402.

Kifer, M. and G. Lausen [1989]. "F-logic: a higher-order logic for reasoning about objects, inheritance, and schemes," *ACM SIGMOD International Conf. on Management of Data*, pp. 143–146.

Papakonstantinou, Y. [1994]. Unpublished memorandum.

Papakonstantinou, Y., A. Gupta, H. Garcia-Molina, and J. D. Ullman [1995]. "A query translation scheme for rapid implementation of wrappers," unpublished memorandum, Dept. of CS, Stanford. Available by anonymous ftp as `pub/papakonstantinou/1995/ transgen.ps` from `db.stanford.edu`.

Papakonstantinou Y., H. Garcia-Molina, and J. Widom [1995]. "Object exchange across heterogeneous information sources," Intl. Conf. on Data Engineering, Taipei, March, 1995. Available by anonymous ftp as `pub/papakonstantinou/1994/ object-exchange-heterogeneous-is.ps` from `db.stanford.edu`.

Quass, D., A. Rajaraman, Y. Sagiv, J. D. Ullman, and J. Widom [1994]. "Querying semistructured heterogeneous information," unpublished memorandum, CSD, Stanford Univ., Nov., 1994. Available by anonymous ftp as `pub/quass/1994/querying-full.ps` from db.stanford.edu.

Rajaraman, A., Y. Sagiv, and J. D. Ullman [1994]. "Answering queries using templates with binding patterns," to appear in PODS, May, 1995. Available by anonymous ftp as pub/rajaraman/1994/limited-opsets.ps from db.stanford.edu.

Wiederhold, G. [1992]. "Mediators in the architecture of future information systems," *IEEE Computer* **25**:3, pp. 38–49.

Zaniolo, C. [1983]. "The database language GEM," *Proc. ACM SIGMOD Intl. Conf. on Management of Data.*