The TSIMMIS Approach to Mediation: Data Models and Languages[1]

Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv,[2] Jeffrey Ullman, Vasilis Vassalos, 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. Introduction

In this first section we introduce the core concept of mediators, which the Tsimmis system implements. Section 2 discusses the components of Tsimmis, and Section 3 discusses the OEM data model used by Tsimmis. Section 4 introduces the mediator-specification language MSL, while in Section 5 we learn how it is used to generate mediators and other components automatically. Finally, in Section 6 we consider another language, LOREL, based on the OEM data model. It is the Tsimmis end-user's query language as well as the query language in the related LORE (Lightweight Object REpository) DBMS.

1.1. The Mediator Concept

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 in different ways by many different information sources. These sources are "heterogeneous" on many levels.

- Some may be relational, others not. Some may not be databases at all, but file systems, the Web, or legacy systems.
- 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.

[1] Work supported by USAF contract F33615–93–1–1339, NSF grant IRI–92–23405, and BSF grant 92–00360.

[2] Permanent address: Dept. of CS, Hebrew Univ., Jerusalem.

- 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.

- The information may not conform to a rigid schema in advance. Examples of "semi-structured" information include that found in SGML documents, repositories like ACeDB (Thierry-Mieg and Durbin [1992]) used in the Human Genome Project, and Lotus NOTES.

One way to integrate different information sources that deal with the same real-world entities is to create a *mediator*, which is a facility capable of answering queries about these entities. The mediator uses the raw sources (suitably interfaced by a *wrapper*), and/or other mediators to answer the queries, as suggested in Fig. 1.

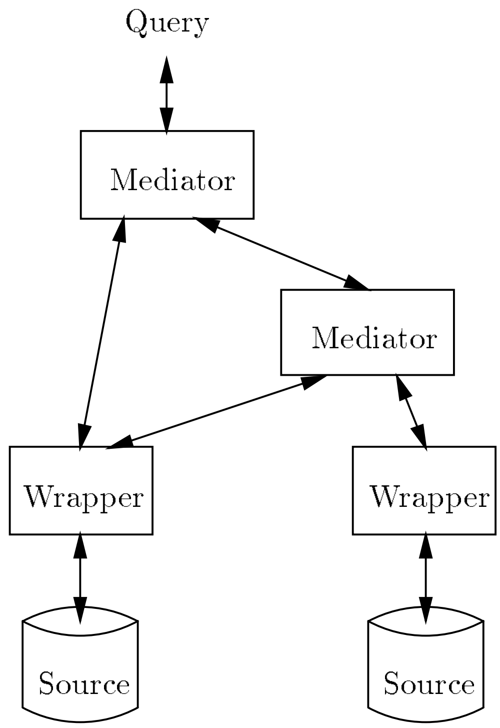

Fig. 1. A network of mediators and information sources.

1.2. 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.

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.

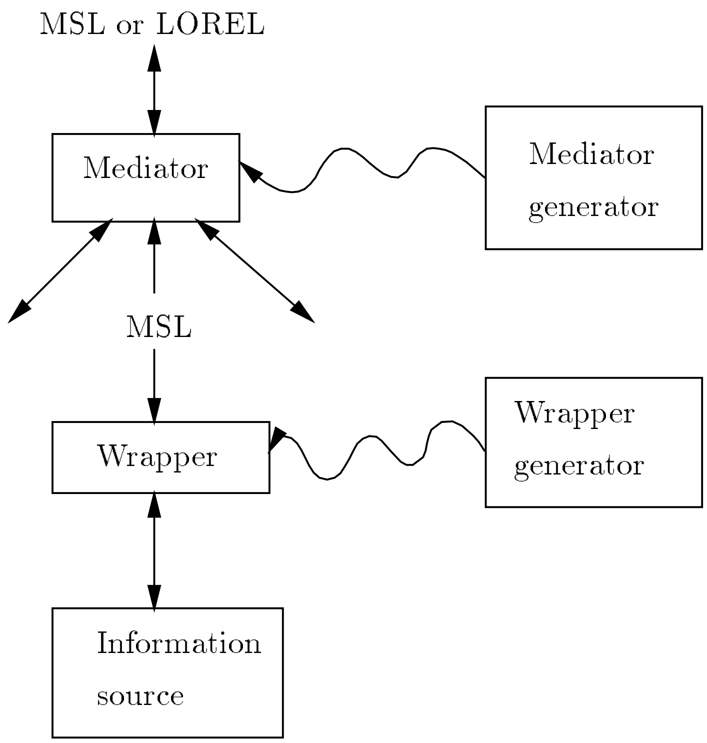

Fig. 2. The components of TSIMMIS.

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.

- *Mediators* are specified with a logic-based object-oriented language called *Mediator Specification Language (MSL)* that can be seen as a view definition language that is targeted to the OEM data model and the functionality needed for integrating heterogeneous sources.

- *Wrappers* (also called translators) are specified with the *Wrapper Specification Language (WSL)* that is an extension to MSL to allow for the description of source contents and querying capabilities. Wrappers allow user 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.

- A *common query language* links components. We are using MSL as both the query language and the specification language for mediators and as the query language for wrappers. The query language *LOREL* ("Lightweight Object REpository Language"), an extension of OQL (Cattell et al. [1994]) targeted to semistructured data, is oriented toward end-user queries and is also the query language of the LORE lightweight database system used for storing OEM objects locally. We discuss MSL in Section 4 and LOREL in Section 6.

- Wrapper- and mediator-generators. We are developing methodologies for generating classes of wrappers and mediators automatically from simple descriptions of their functions. We discuss mediators, wrappers 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 interfaces 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.

OID:	label	type	value

Fig. 3. An OEM object.

3.1. OEM Objects

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. 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 also need not be persistent.
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 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.

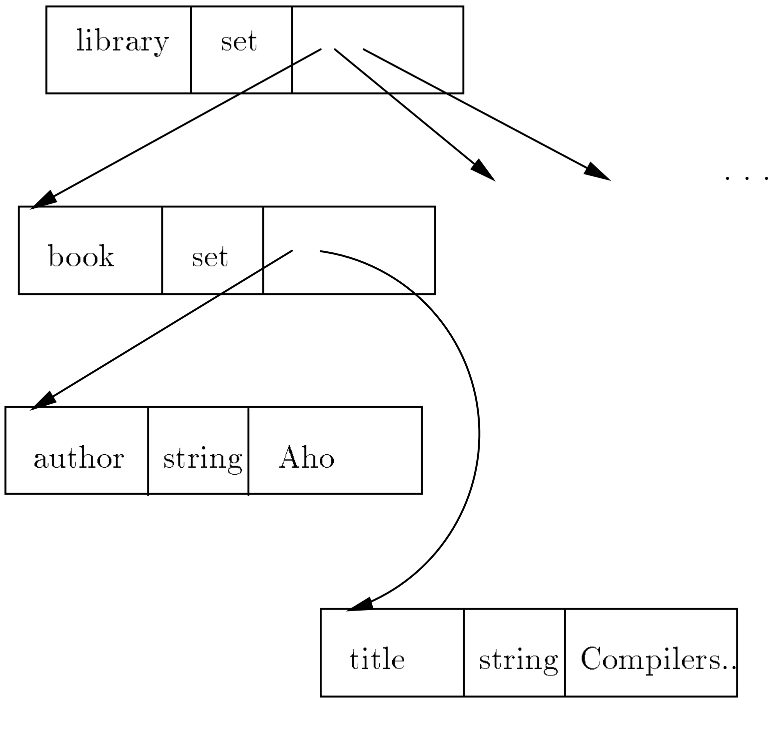

Fig. 4. A collection of OEM objects.

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 natural representation of complex structures, 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 Mediator Specification Language (MSL)

Our mediator-generator *MedMaker* provides a high level language, *MSL*, that allows the declarative specification of mediators. MSL is an object-oriented, logical query language targeted to the OEM data model. It contains features that facilitate the integration of heterogeneous systems. MSL borrows many concepts from logic-oriented languages such as datalog (Ullman [1988], Ullman [1989]), HiLog (Chen, Kifer and Warren [1993]) and F-Logic (Kifer and Lausen [1989]). However, a number of problems are avoided by using sets in a restricted way (variables may explicitly refer only to existing sets of objects). Indeed, in the absence of negative clauses, MSL can be viewed simply as a variant of datalog. The contribution is that unstructured as well as structured data can be queried, unlike datalog or OQL (Cattell et al., 1994).

A query consists of *rules*. Each rule consists of a head followed by a `:-` and a body. The head describes objects made available by the mediator, whereas the body describes conditions that must be satisfied by the source objects. In general, the heads and bodies are based on patterns of the form <*object-id label value*>. We may omit the object-id field when it is irrelevant. If it is missing from a body pattern it means that we do not care about the object-id appearing at the source. If it is missing from a head pattern it means that we have to invent an arbitrary, yet unique, object-id for the "generated" object.

A complete specification of MSL, including formal syntax and semantics is found in Papakonstantinou, Garcia-Molina and Ullman [1995]. 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.

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

```
<booktitle X> :-
    <library { <book {<title X> <author "Aho">}> }>@s1
```

Intuitively, in MSL triangular brackets associate labels with their values, while curly brackets group members of a set; this set would be the value of some object whose type is "set."

This query applies to a root object labeled `library`, which we suppose is available at a source called $s1$, which could be either a wrapper or another mediator. The object pattern (or patterns) that appear in the body of the query are matched against the object structure of the source $s1$ by looking for paths in the object structure that agree with the *nested structure* of the query. 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. The variable `X` binds to the value of the `title` subobjects of `book` objects that have an `author` subobject with value `'Aho'`. The query head indicates that every value that `X` binds is included in the result as value of an object labeled `booktitle`. Technically, these objects become subobjects of an object with label `answer` that is produced by the query. □

The rule in Example 3 is really a query. However, the same rule above could specify a trivial mediator: one that exports book titles of books by Aho that are found at source $s1$. We will further discuss mediator specification in Section 5.

5. Wrappers, 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 wrapper and mediator components automatically.

5.1. Wrappers

Wrappers provide access to heterogeneous information sources by converting application queries into source specific queries or commands. 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. Wrappers accept queries expressed in MSL. However, wrappers differ not only in the set of labels that they can deal with, but also in the set of supported queries involving those labels.

Unlike relational databases, where all SQL queries over the database schema can be answered, most information sources can answer a limited set of queries. 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.

Thus, a Tsimmis wrapper takes a query and decides first of all whether or not its underlying source can answer the query, i.e., whether the query is *directly supported*. If so, it turns the query into something that the underlying source can respond to. The wrapper then converts the answer into the appropriate OEM objects and returns this result. If the query is not directly supported, then the wrapper determines whether the query can be answered by applying local *filtering* (eg. selection) to a directly supported query. The wrapper then again converts the answer received from the source into the appropriate OEM object, applies the filter, and returns this result.

5.2. Wrapper Generation

We have built a template-based tool for generating wrappers (Vassalos [1996]). This wrapper-generator takes a set of templates of the form:

MSL template
// action //

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

Example 4: Suppose we wish to build a wrapper for a source that is a bibliographic search system (such as Folio at Stanford). We might define rules such as:

```
<books X> :-
    <library { X:<book {<title X> <author $AU>}> }>@s1
// sprintf(lookup-query, "find author %s", $AU) //
```

The generated wrapper examines a query and compares it to this and other patterns that are given in the wrapper specification file. If the query matches the pattern, with some string in place of the parameter `$AU`, then the associated action would be executed, with that string in place of the parameter. Thus, the query

```
<books B> :-
    <library { B:<book {<title X> <author "Aho">}> }>@s1
```

would generate a "native" query to the source asking for books authored by Aho. That is, the C function `sprintf` in the action above has the effect of assigning to the variable `lookup-query` the string `"find author Aho"`. This string is then passed to the source. □

We are currently exploring the use of *WSL*, a Wrapper Specification Language (Vassalos, [1996]), that is an extension of MSL with the functionality needed for source descriptions.

The architecture of a *wrapper implementation toolkit* for generating wrappers is described in Papakonstantinou, Gupta, Garcia, and Ullman [1995].

5.3. Extending Wrapper Rules

Since the query language is logic-based, 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 of what the system is capable of handling, 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. This process of enhancing the capability of a wrapper or mediator by logical deduction from its specification we have called *query normalization*.

Rajaraman, Sagiv, and Ullman [1995] 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 in general.

Example 5: 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.

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

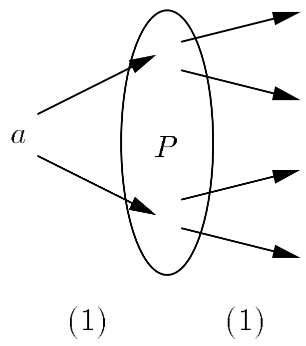

Fig. 5. Finding an individual's grandparents.

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. □

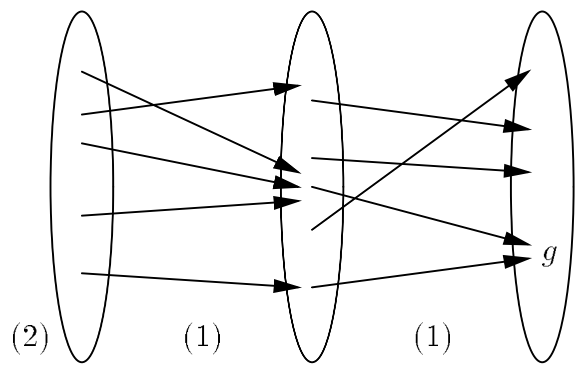

Fig. 6. Finding an individual's grandchildren.

5.4. Mediator Generation

Mediators are used for the integration of multiple heterogeneous information sources. Given a set of sources with wrappers that export OEM objects, we would like to build mediators to integrate and refine the information. In particular, we restrict our attention to mediators that provide integrated OEM "views" of the underlying information. At run time, when the mediator receives a request for information, MedMaker's *Mediator Specification Interpreter (MSI)* collects and integrates the necessary information from the sources, according to the specification. These sources can be either the wrapper of a raw information source or another mediator. The process is analogous to expanding a query against a conventional relational database view.

To describe a mediator in MSL, 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 bodies of the rules describe the object or objects at the source(s) that must exist for the defined object to appear in the mediator's view, and the conditions these objects must satisfy.

Our approach to mediator generation is very much like the use of MSL to define wrappers that we illustrated in Section 5.2. The principal difference is that since the sources for the mediator already "speak" OEM, there is no need for specialized actions to describe what must be done at the wrapper's source.

Example 6: We consider two sources that contain information about the staff of a Computer Science department. The first source is a relational database containing two tables with schemas

```
employee(name, title, reports_to)
student(name, year)
```

A wrapper, named `cs`, exports this information as a set of OEM objects.

A second source is a university "whois" facility that contains information about employees and students. This source contains (among other things) information about one's `dept` and `relation` to the university (student, employee, etc.). A wrapper `whois` provides access to this source.

Let us now consider a mediator, called `med`, that has access to wrappers `cs` and `whois` and exports a set of "cs_person" objects. Our goal in this example is that each "cs_person" object represents a person appearing in both sources. The subobjects of each "cs_person" object should represent combined information about this person. For example, since an object with information about John Doe is exported from both `cs` and `whois`, `med` combines this information and exports a "fused" object. The MSL specification of the `med` mediator is as follows:

```
Rules
<cs_person { <name N> <rel R> }> :-
    <person { <fullname N> <dept 'CS'> <relation R> }>@whois
    AND <R { <name N> }>@cs
```

Intuitively, we may think of the process of "creating" the virtual objects of the mediator as pattern matching. First, we match the patterns that appear in the body against the object structure of `cs` and `whois`, trying to bind the `variables` (represented by identifiers starting with a capital letter, such as `N`, `R`, etc.) to object components of `cs` and `whois`. Then we use the bindings to "construct" the objects specified in the head of the rule. The only difference with using MSL as a query language is that the objects specified by the query rule head are materialized at the client. MSL offers two important capabilities:

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

Example 7: Suppose we have a source that is the arcs of a graph, and we wish to query about paths in that graph. We can 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 nonrecursive MSL 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. □

- Second, the language allows the assignment of semantic object ids to objects as they are "imported" into the mediator. These semantic ids can then be used to specify how various objects are combined or merged into objects "exported" by the mediator. Object-id based set formation replaces the need for explicit grouping operators. MSL allows object ID's to be computed as logical terms, by a particular use of Skolem functions as F-logic (Kifer and Lausen [1989]). The following example illustrates how this capability might be used to advantage.

Example 8: 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, as part of the MSL specification of the mediator, that in the view exported by the mediator this student object is a member of the set that is the value of the course object with ID $course(c)$. □

6. The LOREL Query Language

LOREL (Lightweight Object REpository Language) is an OQL-based query language for the OEM model. It is the end-user query language for Tsimmis. It is also the query language for the LORE lightweight object repository, a related project at Stanford building a DBMS for the OEM data model.

A complete specification of LOREL, including formal syntax and semantics is found in Quass, Rajaraman, Sagiv, Ullman, and Widom [1995].[3] 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.

Example 9: "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
```

[3] There is a new generation of LOREL under development; see `http://db.stanford.edu/lore` for details.

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. □

6.1. 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 OQL or 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 the peculiar way in which SQL handles OR operations 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 10: 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,[4] 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. □

7. Related Work

Various projects on the integration of heterogeneous sources (see Ahmed et al. [1991], Kim et al. [1993], Batini et al. [1986], Hammer et al. [1993], Litwin et al. [1990], Thomas et al. [1990], Gupta [1989], Su et al. [1996] and Motro [1995]) focus on integrating relatively small numbers of structured databases. Most of the research on these projects focuses on resolving the semantic and schematic heterogeneities that arise upon integration. The solutions often rely on semantically rich data models that allow for easy representation of the semantic connections between the distributed data. Unlike these systems, TSIMMIS relies on a lightweight model that can easily adapt to different information representations. Furthermore, it relies on explicit view definition for specifying the integrated view. An interesting alternative is presented by SIMS (Arens et al. [1993]) and the Information Manifold (Levy et al. [1996]), where a reasoning phase is required for realizing which sources have the data of interest, unlike TSIMMIS where view expansion is all that is needed for finding what data each source must contribute.

Recently a new generation of projects has focused on the integration of sources that may not necessarily be structured databases. As we've discussed earlier, the integration system must be able to process the client queries by using only queries supported by the participating sources. HERMES (Subrahmanian et al. [1996]) solves this problem by a mediator specification language where some literals explicitly specify the parameterized calls that are sent to the sources. Unfortunately, this reduces the interface between the integration system and the sources to a limited set of explicitly listed parameterized calls. Unlike HERMES, TSIMMIS and Garlic (Papakonstantinou, Gupta and Haas [1996]) use source query capability descriptions that may describe infinite sets of queries. Unlike TSIMMIS, Garlic and Information Manifold consider the wrappers to be "thin" modules that support the directly specified queries. All capability extensions happen at the mediator level.

The importance of information-integration research is highlighted by the arrival of first-generation commercial products. These include IBM's DataJoiner (Gupta and Lin [1994]) and Microsoft's OLECOM (Blakeley [1996]).

8. Summary

We hope the reader has obtained a feel for the principal innovations in the Tsimmis system:

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

1. The OEM data model, an object-oriented model that uses object labels to represent both class information and attributes (instance variables) of objects.
2. The flexibility of OEM in representing objects of varying structure.
3. The idea of mediator- and wrapper-generation and the way these components can be expressed in the specification language MSL.
4. The idea of query "normalization," allowing the power of a mediator or wrapper to be increased by discovery of additional queries that its logical specification implies it can answer.
5. The query language LOREL for OEM objects and how its partial-match semantics matches the flexible structure of OEM objects.

References

R. Ahmed et al. [1991]. "The Pegasus Heterogeneous Multidatabase System," in *IEEE Computer*, 24:19–27, 1991.

Y. Arens, C. Y. Chee, C.-N. Hsu and C. A. Knoblock [1993]. "Retrieving and Integrating Data from Multiple Information Sources," in *Intl Journal of Intelligent and Cooperative Informations Systems* **2**:127–158, June 1993.

C. Batini, M. Lenzerini, and S. B. Navathe [1986]. " A Comparative Analysis of Methodologies for Database Schema Integration," in *ACM Computing Surveys*, 18:323–364, 1986.

J. Blakeley [1996]. "Data Access for the Masses through OLE DB," *ACM SIGMOD International Conf. on Management of Data*, pp. 161-172.

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 anonymous ftp as `pub/chawathe/1994/tsimmis-overview.ps` from `db.stanford.edu`.

W. Chen, M. Kifer, and D.S. Warren [1993]. "Hilog: a foundation for higher-order logic programming," in *Journal of Logic Programming*, 15:187–230, February 1993.

A. Gupta [1989]. *Integration of Information Systems: Bridging Heterogeneous Databases.* IEEE Press, 1989.

P. Gupta and E. Lin [1994]. "DataJoiner: a practical approach to multidatabase access," *Proc. PDIS Conf.*, page 264, 1994.

J. Hammer and D. McLeod [1993]. "An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems," in *Intl Journal of Intelligent and Cooperative information Systems*, 2:51–83, 1993.

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.

W. Kim et al. [1993]. "On Resolving Schematic Heterogeneity in Multidatabase Systems," in *Distributed And Parallel Databases*, 1:251–279, 1993.

A. Levy, A. Rajaraman, and J. Ordille [1996]. "Querying Heterogeneous Information Sources Using Source Descriptions," *Proc. VLDB Conf.*, 1996.

W. Litwin, L. Mark, and N. Roussopoulos [1990]. "Interoperability of Multiple Autonomous Databases," in *ACM Computing Surveys*, 22:267–293, 1990.

A. Motro [1995]. "Multiplex: A Formal Model for Multidatabases and its Implementation," Technical Report ISSE-TR-95-103, George Mason University, 1995.

Y. Papakonstantinou, A. Gupta, and L. Haas [1996]. "Capabilities-Based Query Rewriting in Mediator Systems," to appear in PDIS 96. Available by anonymous ftp as `pub/papakonstantinou/1995/cbr-extended.ps` from `db.stanford.edu`.

Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina [1996]. "Object fusion in mediator systems," *Proc. VLDB Conf.*, 1996. Available by anonymous ftp as `pub/papakonstantinou/1996/fusion.ps` from `db.stanford.edu`.

Papakonstantinou, Y., A. Gupta, H. Garcia-Molina, and J. D. Ullman [1995]. "A query translation scheme for rapid implementation of wrappers," *Proc. DOOD Conf.*, 1995. 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," *Proc. Intl. Conf. on Data Engineering*, March, 1995. Available by anonymous ftp as `pub/papakonstantinou/1994/object-exchange-heterogeneous-is.ps` from `db.stanford.edu`.

Y. Papakonstantinou, H. Garcia-Molina, J. D. Ullman [1995]. "MedMaker: A Mediation System Based on Declarative Specifications", *Proc. Intl. Conf. on Data Engineering*, March, 1996. Available by anonymous ftp as `pub/papakonstantinou/1995/medmaker.ps` from `db.stanford.edu`

Quass, D., A. Rajaraman, Y. Sagiv, J. D. Ullman, and J. Widom [1995]. "Querying semistructured heterogeneous information," *Proc. DOOD Conf.*, 1995. Available by anonymous ftp as `pub/quass/1994/querying-full.ps` from `db.stanford.edu`.

Rajaraman, A., Y. Sagiv, and J. D. Ullman [1995]. "Answering queries using templates with binding patterns," *Proc. PODS Conf.*, pages 105–112, 1995. Available by anonymous ftp as `pub/rajaraman/1994/limited-opsets.ps` from `db.stanford.edu`.

Su, S. Y. W. et al. [1996]. " NCL: a common language for achieving rule-based interoperability among heterogeneous systems," in *Journal of Intelligent Information Systems: Integrating Artificial Intelligence and Database Technologies* **6**:2–3, pp. 171-198, June 1996.

V. S. Subrahmanian et al. [1996]. "HERMES: A heterogeneous reasoning and mediator system," available at `http://www.cs.umd.edu/projects/hermes/overview/paper`.

J. Thierry-Mieg, and R. Durbin [1992]. "Syntactic definitions for the acedb data base manager," Technical Report MRC-LMB xx.92, MRC Laboratory for Molecular Biology, 1992.

G. Thomas et al. [1990] "Heterogeneous Distributed Database Systems for Production Use," in *ACM Computing Surveys*, 22:237–266, 1990.

J. D. Ullman [1988] *Principles of Database and Knowledge-Base Systems, Vol I: Classical Database Systems.* Computer Science Press, New York, NY, 1988

J. D. Ullman [1989] *Principles of Database and Knowledge-Base Systems, Vol II: The New Technologies.* Computer Science Press, New York, NY, 1989

Vassalos, V. [1996]. "Wrapper specification and query processing in the TSIMMIS project," unpublished memorandum.

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