MedMaker: A Mediation System Based on Declarative Specifications*

Yannis Papakonstantinou, Hector Garcia-Molina, Jeffrey Ullman
Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA

Abstract

Mediators are used for integration of heterogeneous information sources. In this paper we present a system for declaratively specifying mediators. It is targeted for integration of sources with unstructured or semi-structured data and/or sources with changing schemas. In the paper we illustrate the main features of the Mediator Specification Language (MSL)*, show how they facilitate integration, and describe the implementation of the system that interprets the MSL specifications.*

1 Introduction

Many applications require integrated access to heterogeneous information, stored at sources with different data models and access mechanisms [LMR90, Gup89, C+94, A+91]. The TSIMMIS data-integration system provides integrated access via an architecture (see Figure 1.1) that is common in many projects: *Wrappers* [C+94, FK93] (also called *translators* [PGMW95]) convert data from each source into a common model, as illustrated in Figure 1.1. The wrappers also provide a common query language for extracting information. Applications can access data directly through wrappers, but they may also go through *mediators* [PGMW95, Wie92]. A mediator combines, integrates, or refines data from wrappers, providing applications with a "cleaner" view. For example, a mediator for Computer Science publications could provide access to a set of bibliographic sources that contain relevant materials. Users accessing the mediator would see a single collection of materials, with, for example, duplicates removed and inconsistencies resolved (e.g., all authors names would be in the format last name, first name).

Our focus on this paper is on integration of sources that do not have a well defined static schema. This class of sources includes databases that have an often changing schema, as well as information sources that contain unstructured or semistructured data. There are many applications that use such data. A typical example is electronic mail where objects have some well defined "fields" such as the destination and source addresses, but there are others that vary from one mailer to another. Furthermore, fields are constantly being added or modified. The same situation arises with medical records, bibliographic information, knowledge representation frames[G+92], and many others.

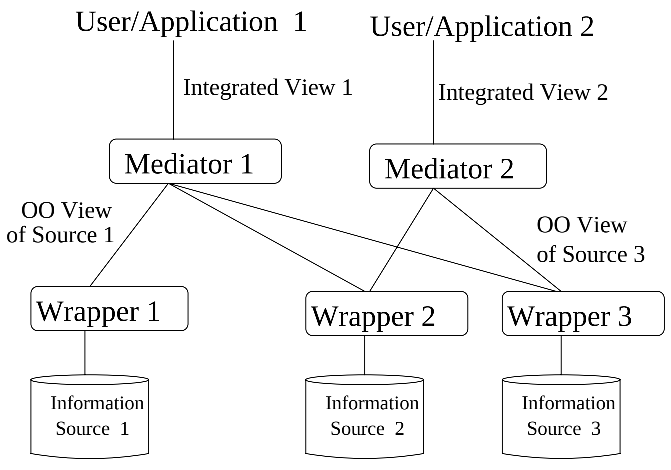


Figure 1.1: The TSIMMIS architecture for integration

1.1 The OEM Model

Most applications that have to deal with unpredictable, unstructured information use a *self-describing* model [MR87], where each data item has an associated descriptive label. In [PGMW95] we have defined a self-describing data model, called the Object Exchange Model (OEM), that captures the essential features of the models used in practice. It also generalizes them to allow arbitrary nesting and to include object identity.

To illustrate the OEM model, consider the following objects (one object per line):

```
<&1, person, set, {&11, &12, &13, &14}>
     <&11, name, string, 'George Jones'>
     <&12, department, string, 'CS'>
     <&13, relation, string, 'employee'>
     <&14,affiliations, set, {&141, &142}>
          <&141, affiliation, string, 'AI'>
          <&142, affiliation, string, 'DB'>
```

Each OEM object consists of an *object-id* (e.g., `&12`), a *label* that explains its meaning (e.g., `department`), a *type* (e.g., `string`), and a *value* of the specified type (e.g., `'CS'`). Object-ids can be of different types, but for now, think of them as arbitrary strings that are used to link objects to their subobjects. (For more details, see [PGM].) Labels are strings that are meaningful to the application or end user. Labels may have different meanings at different sources. Indeed, it will be the job of mediators to resolve these conflicts. Values may be either of an atomic type (e.g., `'George Jones'` is of type string), or be a set of subobjects (e.g., the value of the "affiliations" object is {`&141`, `&142`}).

Some OEM objects (e.g., the object identified by `&1`) are *top-level* objects, and we write them with the leftmost indentation. For performance reasons clients query object structures starting, by default, from the top-level objects. For example, a simple query may ask for top-level "person" objects that have a "department" subobject with value `'CS'`. Nevertheless, the client is not restricted to query the object structure starting from top-level objects, as will be explained in Section 2.

The OEM model forces no regularity on data. For example, a second `person` object may or may not have subobjects with the same labels as the person shown above. The fact that there is no schema, or each object has its own schema if you will, makes it possible to represent heterogeneous, changing information. It also facilitates the representation of information integrated from multiple heterogeneous sources, that typically have different schemas.

1.2 The Mediator Specification Language (MSL)

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. The significant programming effort involved in the hard-coded development of TSIMMIS mediators suggests the need for development of systems that facilitate mediator development. Our mediation system, *MedMaker*, provides a high level language, called *Mediator Specification Language (MSL)* that allows the declarative specification of mediators. 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. The process is analogous to expanding a query against a conventional relational database view. Indeed, MSL can be seen a view definition language that is targeted to the OEM data model and the functionality needed for integrating heterogeneous sources. The special requirements of integration led to the introduction of a number of useful concepts and properties, that are not found in conventional view definition languages. In this paper we present the following features:

- MSL mediator specifications can handle some schema evolution of the underlying sources without a need for rewriting of the specification.
- MSL can handle structure irregularities of the sources without producing erroneous or unexpected results.
- MSL can integrate sources for which we do not fully know their object structures.
- MSL can manipulate both the values and the descriptive semantic labels in the same fashion, getting around problems such as schematic discrepancies [KLK91].

The above capabilities are "packaged" in a high-level declarative language that combines power with simplicity and conciseness, thus allowing the client of an heterogeneous system to easily define an integrated view.

In the next section we present an extended example that illustrates the MSL language and some of its integration capabilities. Then, in Section 3 we discuss the architecture and implementation of MedMaker. Section 4 compares MedMaker to other systems for the integration of heterogeneous information sources and discusses ongoing and future work on MedMaker. The complete syntax and semantics of MSL are provided in [PGM].

2 A Mediator Specification Example

For our extended example, we consider two sources that contain information on the staff of a Computer Science department. The first source is a relational database containing two tables with schemas

```
employee(first_name,last_name,title,reports_to)
student(first_name,last_name,year)
```

A wrapper, named `cs`, exports this information as a set of OEM objects, some of which are shown in Figure 2.2. Notice how the schema information has now been incorporated into the individual OEM objects.[1]

A second source is a university "whois" facility that contains information about employees and students. A wrapper `whois` provides access to this source; several sample objects are shown in Figure 2.3. Notice that in this case there can be irregularities. For instance, object `&p1` contains an email subobject while `&p2` does not.

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 the combined information about this person. For example, since an object with information about Joe Chung is exported from both `cs` and `whois`, `med` combines this information and exports the object of Figure 2.4.

[1]Two minor points: (1) After translation, we have lost knowledge that objects at this source *must* have a regular structure. If this information is important to the applications, it could be exported as additional facts about this source. (2) One could consider it inefficient to repeat the schema in all objects, in this case where there is a regular pattern to objects. This problem can easily be addressed by data compression when objects are exported. Conceptually, we believe it is easier to think of each object as having its own labels.

```
<&e1, employee, set, {&f1,&l1,&t1,&rep1}>
  <&f1, first_name, string, 'Joe'>
  <&l1, last_name, string, 'Chung'>
  <&t1, title, string, 'professor'>
  <&rep1, reports_to, string, 'John Hennessy'>
<&e2, employee, set, {&f2,&l2,&t2}>
  <&f2, first_name, string, 'John'>
  <&l2, last_name, string, 'Hennessy'>
  <&t2, title, string, 'chairman'>
:
<&s3, student, set, {&f3,&l3,&y3}>
  <&f3, first_name, string, 'Pierre'>
  <&l3, last_name, string, 'Huyn'>
  <&y3, year, integer, 3>
:
```

Figure 2.2: The OEM object structure of the `cs` wrapper

```
<&p1, person, set, {&n1,&d1,&rel1,&elm1}>
  <&n1, name, string, 'Joe Chung'>
  <&d1, dept, string, 'CS'>
  <&rel1, relation, string, 'employee'>
  <&elm1, e_mail, string, 'chung@cs'>
<&p2, person, set, {&n2,&d2,&rel2}>
  <&n2, name, string, 'Nick Naive'>
  <&d2, dept, string, 'CS'>
  <&rel2, relation, string, 'student'>
  <&y2, year, integer, 3>
:
```

Figure 2.3: The OEM object structure of `whois`

Problems in Mediator Specification Creating the integrated view from the wrapper views requires the resolution of a number of problems:

- *schema-domain mismatch*: The `whois` source represents names by a long string that contains both the first and the last name, while the `cs` database represents names using the "last_name" and "first_name" subobjects.

- *schematic discrepancy*: Data in one database correspond to metadata of the other. In particular, the status of a person – `employee` or `student` – appears as a value in `whois` (it was part of a relational table), while it appears in the schema of `cs` (it was part of the relational schema).

- *schema evolution*: The format and contents of the sources may change over time, often without notification to the mediator implementor. For example, an attribute "birthday" may appear in either of the two sources, or the "e_mail" attribute may be dropped. We would like our mediator specification to be insensitive to as many of these changes as possible. For example, if "birthday" is included or dropped, it should be automatically included or dropped from the `med` view, without need to change the mediator specification.

```
<&cp1, cs_person, {&mn1,&mrel1,&t1,&rep1,&elm1}>
  <&mn1, name, string, 'Joe Chung'>
  <&mrel1, rel, string, 'employee'>
  <&t1, title, string, 'professor'>
  <&rep1, reports_to, string, 'John Hennessy'>
  <&elm1, e_mail, string, 'chung@cs'>
```

Figure 2.4: Object exported by `med`

- *structure irregularities:* Source `whois` does not follow a regular schema (i.e., it is a semistructured source.)

The Mediator Specification of `med` The following MSL specification MS1 defines the `med` mediator we have described, resolving the integration problems we have discussed above. We will explain this specification in the paragraphs that follow.

```
(MS1) Rules:
   <cs_person {<name N> <rel R> Rest1 Rest2}>
    :- <person {<name N> <dept 'CS'>
                <relation R>|Rest1}>@whois
       AND decomp(N, LN, FN)
       AND <R {<first_name FN>
               <last_name LN> | Rest2}>@cs
   External:
   decomp(string,string,string)(bound,free,free)
       impl by name_to_lnfn
   decomp(string,string,string)(free,bound,bound)
       impl by lnfn_to_name
```

A specification consists of rules that define the view provided by the mediator, and declarations of functions that will be called upon for translating objects from one format to another. Each rule (the above specification has only one rule) consists of a *head* and a *tail* that are separated by the `:-` symbol. The tail describes the patterns of objects that must be found at the sources, while the head describes the pattern of the top-level objects of the integrated view.

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 tail against the object structure of `cs` and `whois`, trying to bind the *variables* (represented by identifiers starting with a capital letter, such as `N`, `Rest1`, 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 specification is based on patterns of the form <*object-id label type value*>, where we may place constants or variables in each position. For simplicity we can drop some of the fields when they are irrelevant. If one field is dropped, we assume it is the type, so we have a pattern of the form <*object-id label value*>. If two fields are dropped, we assume they are the type and the object-id. When the object-id is missing in a tail pattern, it means that we do not care about the object-id's appearing at the sources. When an object-id is missing from a head pattern, it means we do not care what object-id the mediator uses for the "generated" object.

When the *label* (*value*) field contains a constant the pattern matches successfully only with OEM objects that have the same constant in their *label* (*value*) field. On the other hand, when the *label* (*value*) field contains a variable, the pattern can successfully match with any OEM object, regardless of the label (value) of the object. For example, the pattern `<name N>` can match with OEM objects `<&1, name, string, 'Fred'>` or `<&2, name, string, 'Tom'>`. As a result of a successful matching, the variable `N` will bind to the value of the specific OEM object (either `'Fred'` or `'Tom'` in the example).

Returning to our mediator specification example, we match the patterns of the tail against the top-level objects of the corresponding sources, trying to bind the variables of the tail to appropriate object components. In particular, we match the pattern `<person {<name N> ... | Rest1}>` against the objects of source `whois`, trying to bind the variables `N`, `R`, and `Rest1` to appropriate object components. That is, we try to find top-level "person" objects that have a "name" subobject, a "dept" subobject with value `'CS'`, and a "relation" subobject. The object identified by `&p1` (see Figure 2.3) satisfies these requirements. As a result, `N` binds to `'Joe Chung'`, `R` binds to `'employee'`, and `Rest1` binds to the remaining subobjects, i.e., it binds to {`<&elm1, e_mail, string, 'chung@cs'>`} Let us name this set of bindings $b_{w,1}$. Other objects may also satisfy these conditions and produce other bindings for `N`, `R`, and `Rest1`. For instance, `N` can bind to `'Nick Naive'`, `R` to `'CS'`, and `Rest1` to {`<&y2, year, integer, 3>`}.

The specification also indicates that we match the pattern `<R {<first_name FN> ... | Rest2}>` against the objects at source `cs`, obtaining bindings for the variables `R`, `FN`, `LN`, and `Rest2`. Referring to Figure 2.2, we see that one of these binding, call it $b_{c,1}$, will bind `R` to `'employee'`, `FN` to `'Joe'`, `LN` to `'Chung'`, and `Rest2` to {`<&t1, title, string, 'professor'> <&rep1, reports_to, string, 'John Hennessy'>`}.

The next step is to match the two sets of bindings. A binding $b_{w,i}$ from `whois` matches a binding $b_{c,i}$ from `cs` if the two bindings agree on the values assigned to common variables (in this case, `R`) and the name `N` found in `whois` "corresponds" to the last name, first name pair `LN, FN` found in `cs`. For example, binding $b_{w,1}$ matches $b_{c,1}$ because they both bind `R` to `'employee'` and the name `N = 'Joe Chung'` corresponds to last name `LN = 'Chung'` and first name `FN = 'Joe'`.

External Predicates The correspondence between names and first, last name pairs is given by the predicate `decomp(N,LN,FN)`. Conceptually, we can think of `decomp` as a predicate that evaluates to true if `N` is a valid decomposition of last, first names `LN, FN`. In practice, `decomp` is implemented as a pair of functions, `name_to_lnfn` and `lnfn_to_name` (in principle written in any programming language), and defined in the mediator specification. For example, the line `decomp ... by name_to_lnfn` indicates that `name_to_lnfn` can be called with a full name (the first bound parameter); the function decomposes the name and returns the last and first names (second and third free parameters). Similarly, `lnfn_to_name` can compose a last, first name pair and produce a full name. Thus, operationally, to check if `decomp('Joe Chung', 'Chung', 'Joe')` is true, we can call `name_to_lnfn` with input parameter `'Joe Chung'` and see if it returns `'Joe'` and `'Chung'`. If it does, the predicate holds. Equivalently, we can call `lnfn_to_name` to perform the check.[2] We assume that the resulting result would be the same in any scenario. (Having more than one function for `decomp` gives flexibility at execution time.)

Creation of the Virtual Objects For each set of matching bindings from the tail patterns, we conceptually create an object in the `med` view.[3] (We stress that objects are not really materialized by the mediator specification.) The head of the rule tells us how to construct the view objects. For example, the matching bindings $b_{w,1}$ and $b_{c,1}$ result in the object of Figure 2.4.

Note that even though `Rest1` and `Rest2` are bound to sets of objects, and `<name N>` and `<rel R>` are bound to single objects, we can include all four inside the curly braces that define the subobjects for a "cs_person" object. In general, when variables that have been bound to sets appear inside curly braces {} in a rule head, the first level of their contents is "flattened out" and included in the set value that is described by the curly braces pattern.

Note also that our sample head did not specify any types or object-id's for the view objects. The types, of course, are simply set to the types of the bound variables (`string` in our case.) For the object-id's, any arbitrary unique strings can be used (e.g., `&cp1, &mn1, ...` are used in Figure 2.4.)

MSL's Solutions to Mediator Specification Problems The specification of `med` solves the integration problems mentioned earlier, mainly by exploiting the free use of variables in the Mediator Specification Language, and the schema/data combination ability of OEM. For example, we were able simultaneously to bind variable `R` to a value in `whois` and a label in `cs`, thus addressing the schematic discrepancy. The schema evolution problem is handled by the use of variables `Rest1` and `Rest2`. If, say, new attributes such as "birthday" are added to `cs`, no change is required to the mediator specification. The new attribute will be included with `Rest1` and propagated to the integrated view. On the same time, the bindings of variables `Rest1` or `Rest2` are not required to

[2]Of course, if the implementor had provided a function `check_name_lnfn` that is called with all three parameters bound, we would simply call `check_name_lnfn` with input parameters `'Joe Chung'`, `'Chung'`, and `'Joe`.

[3]In reality, we first project the bindings of the variables of the tail, into bindings of the variables that appear in the head of the rule. Then we eliminate *duplicated* bindings, and finally we create an object of `med` for each set of bindings of the variables of the head.

carry homogeneous sets of objects. For example, binding $b_{w,1}$ binds `Rest1` to {`<&elm1, e_mail, string, 'chung@cs'>`} while $b_{w,2}$ binds `Rest1` to {}. In this way, MSL can handle the integration of unstructured sources that do not have a regular schema. Finally, the ability to use external predicates allows us to process atomic values in any desirable way.

One apparent limitation of the integrated view we have defined for `med` is that it only includes information for people that appear in both `cs` and `whois`. In particular, we may wish to include information in `med` even if it appears in a single source. In Section ?? we briefly present other features of the MSL that let us define such views and let us perform other useful integration tasks. (A complete description appears in [PGM].) However, before doing so, in the following section we illustrate how our Mediator Specification Interpreter (MSI) would process an incoming query against the sample definition we have given.

Other Features of the Mediator Specification Language In the previous two sections we illustrated the basic functionality of the MSL language. The language has additional features specifically designed to facilitate the integration and querying of heterogeneous sources. Due to space limitations, we cannot provide examples for all these features. Instead, we briefly summarize some features and refer the reader to [PGM] for examples and details.

First, note MSL's ability to *retrieve schema information*: One can place variables in the label positions of an MSL query, and thus retrieve information about labels and the object structure of a source. This is a useful feature for exploring new or changing sources. Second, MSL provides the *wildcard* feature that allows searches for objects at any level in the object structure of the source, without need to specify the entire path to the desired object. The wildcard feature is especially useful when we form queries without complete knowledge of the structure of the underlying data. (Without appropriate index structures, wildcard searches may be expensive, so some sources may not support them or may support them in a restricted fashion.) Finally, MSL allows the specification of *semantic object-id's* that semantically identify an exported object and they have meaning beyond the mediator call that yielded them. Semantic object-id's provide a powerful mechanism for object fusion. Due to space limitations we will not discuss any further this feature.

3 Architecture and Implementation of MSI

The Mediator Specification Interpreter (the run-time component of MedMaker) processes a query using a pipeline with the following three components (see Figure 2.5):

1. The *View Expander and Algebraic Optimizer (VE&AO)* reads the query and the mediator specification and discovers which objects it must obtain from each source. Furthermore, it determines the conditions that the obtained source objects must satisfy.

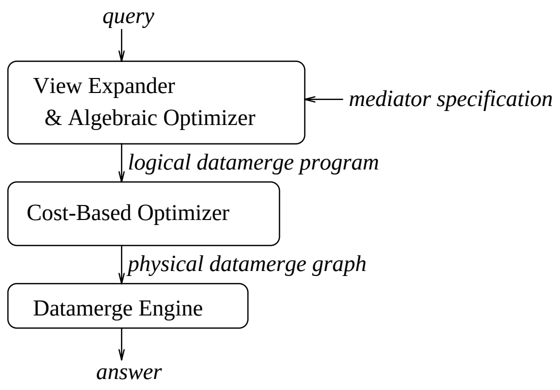


Figure 2.5: The basic architecture of MSI

2. The *cost-based optimizer* develops a plan for obtaining and combining the objects specified by the VE&AO. The plan specifies what queries will be sent to the sources, in what order they will be sent, and how the results of the queries will be combined in order to derive the result objects.

3. The *datamerge engine* executes the plan and produces the required result objects.

In the following subsection we use an example to overview MSI's query processing. Subsections 3.2 to 3.5 discuss each component, the languages at each interface, and various interesting query decomposition and optimization issues.

3.1 Query Processing Overview

Let us assume that a client of mediator `med` wants to retrieve all the data for 'Joe Chung.' In this paper, we use MSL (with one minor modification discussed below) as our query language.[4] The use of MSL simplifies our discussion, and furthermore, MSL makes a good query language because of its power and simplicity. Using MSL, our query can be expressed as:

(Q1) `JC :- JC:<cs_person {<name 'Joe Chung'>}>@med`

The object pattern (or object patterns) that appears in the tail of the query are matched against the object structure of `med` in exactly the same manner that tail patterns of MSL rules are matched against the sources. One new MSL feature that appears in the tail of our sample query is the *object variable* `JC`. The operator `:` indicates that `JC` must bind to "cs_person" objects that have a "name" subobject with value `'Joe Chung'`. The query head indicates that every object

[4]The TSIMMIS project at Stanford is also exploring a different query language, called *LOREL*. It is an object-oriented extension to SQL and is oriented to the end-user. LOREL is described in [Q+]. MSL is more powerful than LOREL (e.g., MSL allows the specification of recursive views) and is targeted to mediator specification.

that JC binds to is included in the result. Unlike mediator specification, when MSL is used for querying, the objects specified by the query rule head are materialized at the client.[5]

View Expansion Given our sample query, the VE&AO replaces the object pattern of the query tail with object patterns that refer to objects of the sources, thus deriving the datamerge rule R2:

```
(R2) <cs_person {<name 'Joe Chung'> <rel R>
                 Rest1 Rest2}>
       :- <person {<name 'Joe Chung'> <dept 'CS'>
                   <relation R> | Rest1}>@whois
          AND decomp('Joe Chung', LN, FN)
          AND <R {<first_name FN> <last_name LN>
                   | Rest2}>@cs
```

Intuitively, the MSI derived the above rule by matching the pattern `JC:<cs_person ...>@med` of the query tail against the head of the mediator specification rule of `med`.[6] After the matching, we generate a datamerge rule whose head is the head of the query and whose tail is the mediator specification rule's tail.

Execution Plan Now that the MSI knows what objects it has to find at the sources, the cost-based optimizer builds a physical datamerge program that specifies what queries should be sent to the sources, in what order they should be sent, and how the results of the queries should be combined in order to produce the query result. Here we informally describe a possible (and efficient) plan for our running example:

1. Bindings for the variables R and Rest1 are obtained from the source whois. The bindings are obtained in two steps. First the following query is sent to whois:

   ```
   <bind_for_whois {<bind_for_R R>
                    <bind_for_Rest1 Rest1>}>
     :- <person {<name 'Joe Chung'> <dept 'CS'>
                 <relation R> | Rest1}>@whois
   ```

   Labels bind_for_whois, bind_for_R and bind_for_Rest1 are simply place-holders that allow the MSI to conveniently pick out the desired information from the returned result objects.

2. Bindings for LN and FN can be obtained from one of the decomp functions, i.e., from name_to_lnfn. We call it with bound parameter N = 'Joe Chung' and obtain LN = 'Chung' and FN = 'Joe'.

3. For each of the R binding of step (1), we combine it with the single binding of step (2), and submit a query to cs to obtain a binding for Rest2. For example, for the binding R = 'employee' we send the following query to cs:

   ```
   <bind_for_cs {<bind_for_Rest2 Rest2>}>
    :-<employee {<first_name 'Joe'>
                 <last_name 'Chung'>|Rest2}>@cs
   ```

4. Once MSI obtains bindings for Rest2 as well, it generates objects that follow the pattern of the head of (R2). For example, considering the bindings we have illustrated so far, the MSI would generate the object of Figure 2.4.

3.2 View Expansion and Algebraic Optimization

The VE&AO matches the query against the mediator specification rules and rewrites the query so that references to the virtual mediator objects are replaced by references to source objects. The result is a *logical datamerge program* that is a set of MSL rules specifying the result. In Section 3.1 we illustrated the view expansion and algebraic optimization process. There, expression (R2) was the logical datamerge program. In the rest of this section we explain the VE&AO process in more detail. In general, the VE&AO formulates the logical datamerge programs in two steps:

- First it matches the query tail conditions with rule heads. The successful matches result in expressions called *unifiers*. Intuitively, our unifiers describe the match between the query and the rule, the conditions that must be pushed to the sources, and other information necessary for the rewriting of the query. Note, they can be viewed as extensions of the unifiers used in resolution of first order clauses [GN88].
- Then for every unifier a logical datamerge rule is formed. The rule head is formed by applying the unifier to the query head, while the rule tail is formed by applying the unifier to the mediator specification rule tails and subsequently forming their conjunction.

For example, consider the query Q1 (Section 3.1) and the specification MS1 of med. The match[7] results in the unifier θ_1 where

$$\theta_1 = \left[\begin{array}{ll} \texttt{N} \mapsto \texttt{'Joe Chung'}, & \\ \texttt{JC} \Rightarrow & \begin{array}{l} <\texttt{cs\_person}\ \{<\texttt{name 'Joe Chung'}> \\ \qquad <\texttt{rel R}>\ \texttt{Rest1 Rest2}\}> \end{array} \end{array} \right]$$

The above unifier consists of one *mapping*, indicated by the $\mapsto$, and one *definition*, indicated by the $\Rightarrow$. (The need for discriminating between mappings and definitions will become apparent in the next paragraphs.) The application of θ_1 to the query head

[5] Here we do not address the problem of materializing OEM objects at clients. The various issues and strategies are discussed in [PGMW95].

[6] If there were several rules, the MSI would look for one or more matching rule heads. If more than one head matches, then more than one rule will be considered; resulting objects will be added to the result.

[7] Before we match a query with one or more rules we must rename the variables that appear in the query and the rules, so that no two rules, or a query and a rule have identically named variables.

causes the substitution of `JC` by the structure following the $\Rightarrow$. Similarly, the application of θ_1 to the mediator rule tail causes the substitution of `N` by `'Joe Chung'`. Combining the transformed query head with the transformed mediator rule tail we obtain the logical datamerge rule Q2.

In general, a unifier may contain any number of mappings and/or definitions. When the VE&AO matches a query condition with a rule head it generates all unifiers θ such that

1. If we apply the mappings to the query condition and the mediator rule head, the transformed query condition pattern is *contained* in the rule head pattern. In the example, the transformed query condition `<cs_person {<name 'Joe Chung'>}>` is contained in the transformed rule head `<cs_person {<name 'Joe Chung'> <relation R> Rest1 Rest2}>` because they have the same label `cs_person` and every subobject pattern of the query condition pattern (i.e., the pattern `<name 'Joe Chung'>`) is identical to a subobject pattern of the rule head. Containment guarantees that any mediator object generated by the transformed rule satisfies the query condition pattern.

2. There is a definition for every object, value, or "rest" variable that appears in the query head and also appears in the query tail preceding a "`:`". The definition carries all the information about the structure of the mediator objects that bind to the query variable. For example the definition of `JC` carries all the required information about the mediators `cs_person` objects.

3.3 Pushing Conditions to the Sources

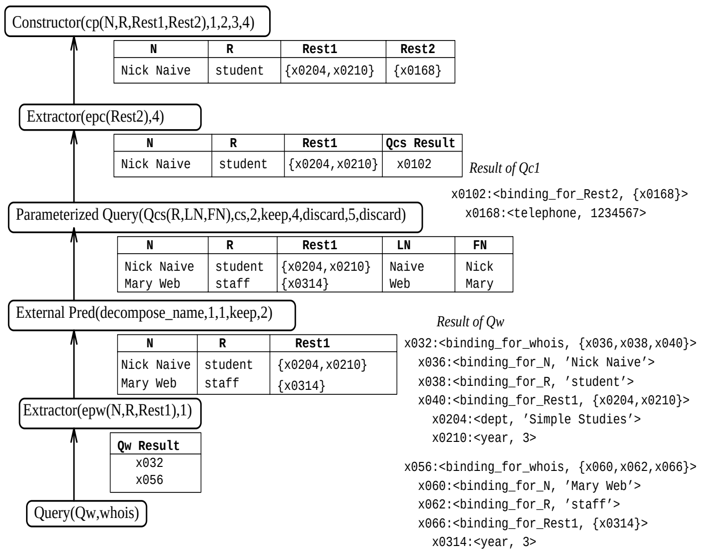
The VE&AO pushes conditions such as "the name must equal `'Joe Chung'`" to the corresponding source. Indeed, VE&AO pushes to the sources all conditions that can be pushed, thus implementing the (well-known in relational DB's) "push selections down" algebraic optimization. In our environment with nested objects that may have unknown structure, algebraic optimization is substantially more challenging than in a relational environment. To illustrate this point, assume that the following query, that retrieves the data of 3rd year students, is sent to mediator `med` (specified by MS1):

```
S :- S:<cs_person {<year 3>}>@med
```

Mediator `med` joins data from two sources, and we cannot tell in advance whether the "year" object comes from one source or the other. In particular, when we match the query against the mediator specification, the `<year 3>` pattern can be "pushed" either into `Rest1` or into `Rest2`. The two possibilities correspond to the unifiers τ_1 and τ_2:

$$\tau_1 = \left[\begin{array}{ll} \texttt{Rest1} \mapsto \{<\texttt{year 3}>\}, & \\ \texttt{S} \Rightarrow <\texttt{cs\_person} & \begin{array}{l}\{<\texttt{name N}> <\texttt{rel R}> \\ \texttt{Rest1 Rest2}\} > \end{array} \end{array} \right]$$

$$\tau_2 = \left[\begin{array}{ll} \texttt{Rest2} \mapsto \{<\texttt{year 3}>\}, & \\ \texttt{S} \Rightarrow <\texttt{cs\_person} & \begin{array}{l}\{<\texttt{name N}> <\texttt{rel R}> \\ \texttt{Rest1 Rest2}\} > \end{array} \end{array} \right]$$


Constructor(cp(N,R,Rest1,Rest2),1,2,3,4)
Extractor(epc(Rest2),4)
Result of Qc1
Parameterized Query(Qcs(R,LN,FN),cs,2,keep,4,discard,5,discard)
External Pred(decompose_name,1,1,keep,2)
Result of Qw
Extractor(epw(N,R,Rest1),1)
Query(Qw,whois)

N	R	Rest1	Rest2
Nick Naive	student	{x0204,x0210}	{x0168}

N	R	Rest1	Qcs Result
Nick Naive	student	{x0204,x0210}	x0102

```
x0102:<binding_for_Rest2, {x0168}>
  x0168:<telephone, 1234567>
```

N	R	Rest1	LN	FN
Nick Naive Mary Web	student staff	{x0204,x0210} {x0314}	Naive Web	Nick Mary

N	R	Rest1
Nick Naive Mary Web	student staff	{x0204,x0210} {x0314}

Qw Result
x032 x056

```
x032:<binding_for_whois, {x036,x038,x040}>
  x036:<binding_for_N, 'Nick Naive'>
  x038:<binding_for_R, 'student'>
  x040:<binding_for_Rest1, {x0204,x0210}>
    x0204:<dept, 'Simple Studies'>
    x0210:<year, 3>

x056:<binding_for_whois, {x060,x062,x066}>
  x060:<binding_for_N, 'Mary Web'>
  x062:<binding_for_R, 'staff'>
  x066:<binding_for_Rest1, {x0314}>
    x0314:<year, 3>
```

Figure 3.6: A physical datamerge graph

The two unifiers give rise to the following two rules, that constitute the logical datamerge program:

```
(R3)<cs_person {<name N> <rel R> Rest1 Rest2}>
    :- <person {<name N> <dept 'CS'> <relation R>
                |Rest1:{<year 3>}}>@whois
       AND decomp(N, LN, FN)
       AND <R {<first_name FN> <last_name LN>
                |Rest2}>@cs
(R4)<cs_person {<name N> <relation R> Rest1 Rest2}>
    :- <person {<name N> <dept 'CS'> <relation R>
                | Rest1}>@whois
       AND decomp(N, LN, FN)
       AND <R {<first_name FN> <last_name LN>
                | Rest2:{<year 3>}}>@cs
```

Note, mappings of the form $\texttt{Rest1} \mapsto \{< \texttt{year 3} >\}$ cause the attachment of the conditions specified inside the {} to the specified variable (`Rest1` in the example). If `Rest1` has already some conditions S associated with it, VE&AO would merge S with the the `<year 3>` condition.

Note, the examples of the previous paragraphs dealt with single condition queries. Nevertheless, the demonstrated techniques have been easily extended and implemented for multiple condition queries.

3.4 The Physical Datamerge Graph and the Datamerge Engine

The optimizer receives the logical datamerge program from the VE&AO and generates a *physical datamerge graph*. This graph specifies the queries to be sent to the sources as well as the mechanics for constructing the query result from the results received from the sources. The graph is then executed by the datamerge engine, which produces the query result.

In this section we illustrate datamerge graph execution through a detailed example. Our goal is not to describe our implementation in full detail, but rather to show the capabilities of our datamerge engine. As our starting point we use logical datamerge rule Q3. From it, the optimizer may generate the physical datamerge graph of Figure 3.6. This is a "dataflow" graph, where the nodes (rounded boxes) represent the operations to be executed by the engine. The rectangles next to the arcs of the graph represent tables that flow during a sample run of this graph. Typically, the tuples of the tables carry bindings for the logical datamerge program variables.

The datamerge engine executes the graph in a bottom-up fashion. First, the lower *query* node is executed. This causes query `Qw` to be sent to source `whois`, obtaining bindings for `N`, `R`, and `Rest1`. Query `Qw` is provided to the engine by the optimizer, and is defined as:

```
(Qw) <bind_for_whois {<bind_for_N N>
                      <bind_for_R R>
                      <bind_for_Rest1 Rest1>}>
     :- <person {<name N> <dept 'CS'>
                 <relation R>
                 | Rest1:{year 3}}>@whois
```

The result of `Qw` is placed in the mediator's memory. In Figure 3.6 we show this result at the bottom of the figure. The numbers with a "x" prefix represent object addresses in the mediator's memory. For example, one result object is at address `x032`; it has label `bind_for_whois` and its value is a set containing the objects at locations `x036`, `x038` and `x040`. For readability, we omit the object-id and type fields of the objects from the figure.

The query operator produces a table where each line contains the address of a top-level result object (`x032` and `x056` in the example). For readability, we add a heading row to our tables (`Qw Result` in this case), but these do not appear in practice.

The table is passed to the next operator in the graph, an *extractor* node that extracts bindings of the variables `N`, `R`, and `Rest1` (from the "binding_for_whois" objects) and outputs a table of corresponding (`N`, `R`, `Rest1`) tuples. The extractor node has two parameters: the first is the optimizer provided object pattern `epw`, defined by

```
<bind_for_whois {<bind_for_N N> <bind_for_R R>
                 <bind_for_Rest1 Rest1>}>
```

`epw` indicates where the desired bindings are found in the result objects; the second parameter (`1`) indicates the column of the input table that contains the objects that are the subject of the extraction. Again, the heading row in the output table is only for readability. Also for reabability, in the `N` and `R` columns we write strings, while in reality we have pointers to the strings. Similarly, in the `Rest1` column we write the full sets while in reality the column contains pointers to the indicated sets.

Then, for every tuple, the *external pred(-icate)* node invokes the predicate `decomp`. The other parameters for this node indicate: the number of arguments for `decomp` (`1`); the column of the input table containing the one input parameter (`1`); whether the input column is kept in the output table;[8] and the number of result arguments from `decomp` (`2`).

[8]As opposed to extractor nodes that always dicard their input column (after using it).

The next node is the *parameterized query* node. For each tuple of its input table, this node generates a query for source `cs` requesting bindings for `Rest2` that are needed to construct final result objects. The query to send is defined by `Qcs` which is provided by the optimizer along with the graph:

```
(Qcs(R,LN,FN)) <bind_for_Rest2 Rest2>
               :-<$R {<last_name $LN>
                      <first_name $FN>|Rest2}>@cs
```

The values for query parameters `$R`, `$LN`, and `$FN` are taken from the 2nd, 4th, and 5th columns of the incoming table. (The `keep` and `discard` parameters again indicate if the inputs columns remain in the output table.) Thus, for our sample data, two queries `Qcs1` and `Qcs2` are emitted:

```
(Qc1) <bind_for_Rest2 Rest2>
      :-<student {<last_name 'Naive'>
                  <first_name 'Nick'>|Rest2}>@cs
(Qc2) <bind_for_Rest2 Rest2>
      :-<staff {<last_name 'Web'>
                <first_name 'Mary'>|Rest2}>@cs
```

Let us assume that `Qc1` returns only the `x0102` "binding_for_Rest2" object and `Qc2` does not return anything. In this case, the parameterized query node outputs the table shown in Figure 3.6. After the upper extractor node extracts `Rest2` bindings from the results of the parameterized query node, the *constructor node* is activated and creates the final result objects. The form of these objects is defined by the pattern `cp(N,R,Rest1,Rest2)` where

```
cp(N,R,Rest1,Rest2) =
 <cs_person {<name N> <relation R> Rest1 Rest2}.
```

For each row in the input table, the constructor operator takes a row (1st, 2nd, 3rd, and 4th values), assigns them to the `N`, `R`, `Rest1`, and `Rest2` values in `cp`, creating one of the final result objects.[9]

Through this example we have illustrated how the entire mediation process can be described by a low level executable graph. The nodes of our datamerge graphs are the "machine language" of MedMaker which is run by our implementation of the datamerge engine. (Indeed, it is interesting to compare them with relational algebra expressions.)

3.5 Cost-Based Optimization Challenges

There are more than one physical datamerge graphs that correspond to a logical datamerge program. The optimizer has to select the "optimal" graph. However, optimization of a powerful object-oriented language that operates on autonomous and heterogeneous information sources is much harder than the optimization of traditional SQL queries on a conventional database. Our current implementation uses some very simple heuristics to guide datamerge graph selection. This seems to work well, at least for simple scenarios. In the rest of this section we briefly discuss some of our research directions for optimization.

[9]Our current implementation does not have a duplicate elimination feature, though the MSL semantics describe duplicate elimination in the OEM context.

Note the following two hard problems for the cost-based optimizer of a mediator: First, the limited query capabilities of the underlying sources may prohibit even simple algebraic optimizations, such as "push selections and projections down". For example, the source `whois` may not be able to evaluate the condition on "year" that appears in `Qw`. A solution to this problem appears in [PGH]. A second problem arises when the wrappers do not provide cost and statistics information. In this case, the optimizer has to rely on ad-hoc heuristics (e.g., the outer patterns of the join order are the ones that have the greatest number of conditions) or tries to build its own statistics database that is based on results of previous queries and on sampling.

4 Related Work and Discussion

In this section we contrast MedMaker to other heterogeneous information source integration systems, we discuss our motivation behind the design of OEM and MSL, and we describe our ongoing work on MedMaker.

It is widely accepted that the relational data model and the corresponding view definition languages are insufficient to provide integration, even of relational databases [KLK91]. Thus, many projects have adopted (or defined) OO models to facilitate integration (some examples are [C+94, A+91]). We described in Sections 1 and 2 the OEM features that make it suitable for integration of heterogeneous information systems. Another difference between OEM and conventional OO models is that OEM is much simpler and does not have a strong typing system. OEM supports only *object nesting* and *object identity*, while other features, such as classes, methods, and inheritance are not supported directly. (Nevertheless, classes and methods can be "emulated" [PGMW95]).

We believe that MSL mediator specifications tend to be short and simple and avoid questions such as "what is the class of the view objects?", that complicate object-oriented view definition[AB91]. In spite of its simplicity, MSL is quite powerful. For instance, it allows the construction of arbitrarily complex object structures (which XSQL [KKS92] does not).

MSL and OEM can be seen as a form of first-order logic. Indeed, we borrow many concepts from logic oriented languages such as datalog [Ull88, Ull89], HiLog [CKW93], O-Logic [Mai86], and F-Logic [KL89]. HiLog first proposed – under a logic framework – the idea of mixing schema and data information.[10]

A very important difference between MedMaker and other integration systems is that MedMaker can integrate conventional well-structured databases that have a static schema and at the same time can integrate sources that do not have a regular schema, or sources that have an often-changing schema. The ability to integrate all kinds of sources is due to:

1. OEM's absence of schema, that allows the intuitive representation of heterogeneous, semistructured, and changing information.
2. MSL's ability to exploit regularities and complete knowledge of the schema (the example of Section 3.3 demonstrated the tradeoff between performance and partial knowledge of the schema).

[10] [KLK91] has also proposed an interesting mixing of schema and data information for the relational data model.

Though systems that integrate well-structured conventional databases exist (e.g., [A+91, K+93, BLN86, LMR90, T+90, Gup89]) and recently systems for the integration of sources with minimal structure have also appeared [Fre, S+93], we do not know of view definition based systems ([A+91, Ber91, CWN94] and others) that handle the whole spectrum of information sources simultaneously, and with MSL's flexibility.

Note, MedMaker performs integration by "working" with the structures of the source objects. Semantic information is effectively encoded in the MSL rules that do the integration. There are many projects that follow MedMaker's "structural" approach [Ber91, DH86, B+86], as well as many projects that follow a semantic approach [HM93, H+92]. We believe that the power of the structural approach, along with the flexibility, generality, and conciseness of OEM and MSL make the "structural" approach a better candidate for the integration of widely heterogeneous and semistructured information sources.

Acknowledgments

We are grateful to Koichi Munakata for designing and implementing the datamerge engine. We also thank Joachim Hammer, Pierre Huyn, Dallan Quass, Anand Rajaraman, Anthony Tomasic, Vasilis Vassalos, Jennifer Widom, and the entire Stanford Database Group for numerous fruitful discussions and comments.

References

[A+91] R. Ahmed et al. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, 1991.

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD Conference*, pages 238–47, Denver, CO, May 1991.

[B+86] Y.J. Breibart et al. Database integration in a distributed heterogeneous database system. In *Proc. 2nd Intl. IEEE Conf. on Data Engineering*, Los Angeles, CA, February 1986.

[Ber91] E. Bertino. Integration of heterogeneous data repositories by using object-oriented views. In *Proc Intl Workshop on Interoperability in Multidatabase Systems*, pages 22–29, Kyoto, Japan, 1991.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18:323–364, 1986.

[C+94] M.J. Carey et al. Towards heterogeneous multimedia information systems: The Garlic approach. Technical Report RJ 9911, IBM Almaden Research Center, 1994.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. Hilog: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, February 1993.

[CWN94] S. Chakravarthy, Whan-Kyu Whang, and S.B. Navathe. A logic-based approach to query processing in federated databases. *Information Sciences*, 79:1–28, 1994.

[DH86] U. Dayal and H. Hwang. View definition and generalization for database integration in a multidatabase system. In *Proc. IEEE Workshop on Object-Oriented DBMS*, Asilomar, CA, September 1986.

[FK93] J.C. Franchitti and R. King. Amalgame: a tool for creating interoperating persistent, heterogeneous components. *Advanced Database Systems*, pages 313–36, 1993.

[Fre] M. Freedman. WILLOW: Technical overview. Available by anonymous ftp from `ftp.cac.washington.edu` as the file `willow/Tech-Report.ps`, September 1994.

[G$^+$92] M.R. Genesereth et al. Knowledge Interchange Format. Version 3.0. Reference Manual. Technical Report Logic-92-1, Stanford University, 1992. Also available by URL `http://logic.stanford.edu/kif.html`.

[GN88] M.R. Genesereth and N.J. Nillson. *Logical Foundations of Artificial Intelligence*. Morgan Cauffman, 1988.

[Gup89] A. Gupta. *Integration of Information Systems: Bridging Heterogeneous Databases*. IEEE Press, 1989.

[H$^+$92] M. Huhns et al. Enterprise information modeling and model integration in Carnot. Technical Report Carnot-128-92, MCC, 1992.

[HM93] J. Hammer and D. McLeod. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *Intl Journal of Intelligent and Cooperative information Systems*, 2:51–83, 1993.

[K$^+$93] W. Kim et al. On resolving schematic heterogeneity in multidatabase systems. *Distributed And Parallel Databases*, 1:251–279, 1993.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. ACM SIGMOD*, pages 59–68, 1992.

[KL89] M. Kifer and G. Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Conf.*, pages 134–46, Portland, OR, June 1989.

[KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of heterogeneous databases with schematic discrepancies. In *Proc. ACM SIGMOD*, pages 40–9, Denver, CO, May 1991.

[LMR90] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22:267–293, 1990.

[Mai86] D. Maier. A logic for objects. In J. Minker, editor, *Preprints of Workshop on Foundations of Deductive Database and Logic Programming*, Washington, DC, USA, August 1986.

[MR87] L. Mark and N. Roussopoulos. Information interchange between self-describing databases. *IEEE Data Engineering*, 10:46–52, 1987.

[PGH] Y. Papakonstantinou, A. Gupta, and L. Haas. Capabilities-based query rewriting in mediator systems. Available via ftp at `db.stanford.edu` file `/pub/papakonstantinou/1995/cbr-extended.ps`.

[PGM] Y. Papakonstantinou and H. Garcia-Molina. Object fusion in mediator systems (extended version). Available by anonymous ftp at `db.stanford.edu` as the file `/pub/papakonstantinou/1995/fusion-extended.ps`.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proc. ICDE Conf.*, pages 251–60, 1995.

[Q$^+$] D. Quass et al. Querying semistructured heterogeneous information. To appear in DOOD95. Available by anonymous ftp at `db.stanford.edu` as the file `/pub/quass/1994/querying-submit.ps`.

[S$^+$93] K. Shoens et al. The Rufus system: Information organization for semistructured data. In *Proc. VLDB Conference*, Dublin, Ireland, 1993.

[T$^+$90] G. Thomas et al. Heterogeneous distributed database systems for production use. *ACM Computing Surveys*, 22:237–266, 1990.

[Ull88] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. I: Classical Database Systems*. Computer Science Press, New York, NY, 1988.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies*. Computer Science Press, New York, NY, 1989.

[Wie92] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25:38–49, 1992.