Object Fusion in Mediator Systems (Extended Version)*

Yannis Papakonstantinou, Hector Garcia-Molina

Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA
{yannis,hector,ullman}@cs.stanford.edu
FAX: +1 (415) 725-2588

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, 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 discuss the implementation of the system that interprets the MSL specifications.

Keywords: data modeling; datalog; declarative language; heterogeneous information; information integration; integration architecture; logic programming; mediator; object identity; object-orientation; query language; semistructured information; view definition.

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: *Translators* [PGMW] (also called *wrappers* [C+94]) convert data from each source into a common model, as illustrated in Figure 1.1. The translators also provide a common query language for extracting information. Applications can access data directly through translators, but they may also go through *mediators* [PGMW, Ire, Wie92]. A mediator combines, integrates, or refines data from translators, 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

*This work was supported by ARPA Contract F33615-93-1-1339, by the Center for Integrated Systems at Stanford University, and by equipment grants from Digital Equipment Corporation and IBM Corporation. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the US Government.

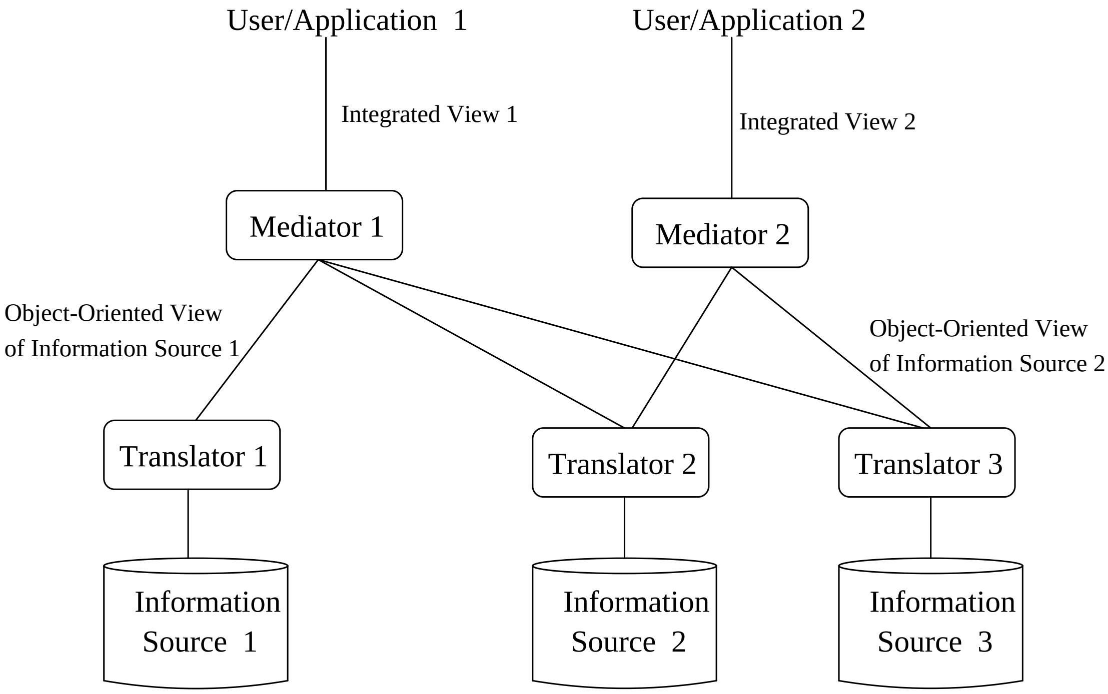

Figure 1.1: The TSIMMIS architecture for integration

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 other fields 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, Mos83, MY89], and many others.

1.1 The OEM Model

Most applications that have to deal with unpredictable, unstructured information use a *self-describing* model, where each data item has an associated descriptive label. Applications include tagged file systems [Wie87], Lotus Notes [Mar93], the Teknekron Information Bus [O+93], LOOM frames [MY89], electronic mail, RFC1532 bibliographic records, and many more. In [PGMW] we have defined a self-describing data model, called the Object Exchange Model (OEM), which 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, 'Systems Lab'>
        <&142, affiliation, string, 'Housing Committee'>
```

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 Section 4.5.) 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 that will become clear in Section 4.2, 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 4.2.

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 explicit 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 translators 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 [Ire] 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 as a view definition language that is targeted to the OEM data model and to the functionality needed for integrating heterogeneous sources. The special requirements of integration led to the introduction of a number of useful concepts and properties not found in conventional view definition languages:

- MSL mediator specifications can handle some schema evolution of the underlying sources without a need for rewriting of the specification (see Section 2).
- MSL can handle structure irregularities of the sources without producing erroneous or unexpected results.
- MSL can integrate sources whose object structures we do not fully know (see Section 2).
- MSL can manipulate both the values and the descriptive semantic labels in the same fashion, getting around problems such as schematic discrepancies [KLK91] (see Section 2). Further-

more, MSL can also manipulate object-id's in the same fashion as values, providing object identification on top of value-oriented sources (see Section 4.5).

- MSL provides a flexible technique (based on object identity) for grouping of source objects that relate to the same entity (see Section 4.6).
- MSL can express operations that are not found in traditional view definition languages. For example, assume that a mediator may access two sources that charge the user for exported information. An MSL specification may indicate that some part of the view should be obtained from the more expensive source, only if it can not be obtained from the cheaper one(see Section 4.4).

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. The example is continued in Section 3, where we introduce the query processing mechanisms of MedMaker. In Section 4 we present the key features of MSL. Then, in Section 5 we discuss the architecture of MedMaker along with some interesting implementation issues. Section 6 compares MedMaker to other systems for the integration of heterogeneous information sources and discusses ongoing and future work on MedMaker. Appendix A provides the complete syntax of the language and Appendix B provides the semantics.

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 translator, named `cs`, exports this information as a set of OEM objects suggested by Figure 2.2. Notice how the schema information has now been incorporated into the individual OEM objects.[1]

A second source is the university "whois" facility that contains information about employees and students. A translator `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 translators `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` translator

```
<&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, 'Simple Studies'>
     <&rel2, relation, string, 'student'>
⋮
```

Figure 2.3: The OEM object structure of `whois`

2.1 Problems in Mediator Specification

Creating the integrated view from the translator 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 corresponds 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

```
<&cp1,cs_person, {&mn1,&mrel1,&t1,&rep1,&elm1}>
      <&mn1, name, string, 'Joe Chung'>
      <&mrel1, relation, 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`

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.

- *structure irregularities:* Source `whois` does not follow a regular schema.

2.2 The Mediator Specification of `med`

The following MSL specification 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.

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

A specification consists of

1. rules that define the view provided by the mediator, and
2. 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*, followed by the symbol `:-` and a *tail*. 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.

Appearing in the rule are patterns of the form <*object-id label type value*>, with 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* or *value* field contains a constant, the pattern only matches OEM objects with the same constant in their label or value field, respectively. On the other hand, when the label or value field contains a variable, the pattern can successfully match with any OEM object, regardless of the label or 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> <dept 'CS'> <relation R> | 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 some "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 `'Simple Studies'`, and `Rest1` to the empty set.

The specification also indicates that we match the pattern

```
<R {<first_name FN> <last_name LN> | 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 following conditions hold:

1. The two bindings agree on the values assigned to common variables, in this case, `R`.

2. The name `N` found in `whois` "corresponds" to the (last_name, first_name) pair `LN, FN` found in `cs`.

For example, we see that 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'`.

2.3 External Predicates

The correspondence between names and (first_name, last_name) pairs is given by the predicate

```
decompose_name(N,LN,FN)
```

Conceptually, we can think of `decompose_name` as a predicate that evaluates to true if `N` is a valid decomposition of last, first names `LN, FN`. In practice, `decompose_name` is implemented in some programming language, as a pair of functions, `name_to_lnfn` and `lnfn_to_name` which are refered in the mediator specification. For example, the line

```
decompose_name(string,string,string)(bound,free,free) implemented 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 `decompose_name('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 answer would be the same in any scenario. (Having more than one function for `decompose_name` gives flexibility at execution time; see Section 3.)

2.4 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 `<relation 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, their contents are "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.)

2.5 How Medmaker Solves Problems of Information Integration

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.

[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.

- We addressed the schematic discrepancy problem by simultaneously binding variable `R` to a value in `whois` and a label in `cs`.
- We addressed the schema evolution problem using the 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.
- At the same time, MSL handles the integration of unstructured sources with no regular schema: the bindings of variables `Rest1` or `Rest2` are not required to 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 `{}`.
- 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 4.6 we will present other features of the MSL that let us define such views and let us perform other useful integration tasks. 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.

3 MedMaker's Query Processing

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 two minor modifications 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:

```
JC :- JC:<cs_person {<name 'Joe Chung'>}>@med            (3.1)
```

The object patterns that appear 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 that `JC` binds to is included in the result. All of the generated objects are packaged in the set value of the following object that has a distinguished label `answer` :

```
<&a1,answer, set, {&a11, ...}>
     <&a11,cs_person, {&a11n, &a11r, ...}>
           <&a11n, name, string, 'Joe Chung'>
           ⋮
```

[4] The TSIMMIS project at Stanford is also exploring a different query language, called *LOREL*, intended for queries among users, mediators, and translators. LOREL is described in [Q^+] and is concurrently being submitted to VLDB 95. MSL is more powerful than LOREL (e.g., MSL allows the specification of recursive views) and is targeted to mediator specification.

As our query illustrates, there are only two differences when MSL is used for queries: First, the objects described by the query head are packaged into a single "answer" object. Second, the objects specified by the head of the query are materialized at the client. [5]

3.1 View Expansion

Given our sample query, we now illustrate how the MSI expands it against the mediator specification and submits appropriate queries to the sources. To perform view expansion, the MSI replaces the object pattern of the query tail with object patterns that refer to objects of the sources, thus deriving the query:

```
<cs_person {<name ‘Joe Chung’> <relation R> Rest1 Rest2}>        (3.2)
:- <person {<name ‘Joe Chung’> <dept ‘CS’> <relation R> | Rest1}>@whois
AND decompose_name(‘Joe Chung’, LN, FN)
AND <R {<first_name FN> <last_name LN> | Rest2}>@cs
```

Intuitively, the MSI derived the above query by matching the pattern

```
JC:<cs_person {<name ‘Joe Chung’>}>@med
```

of the query tail against the head of the mediator specification rule of med.[6] The matching is successful because the head of the rule declares that the mediator has top-level "cs_person" objects, that in turn have "name" subobjects. After the matching, the new query consists of:

- New query tail: This is the rule tail with variables that matched replaced by the corresponding bindings. In the matching, N was bound to ‘Joe Chung’.
- New query head: This is the old query head with variables (in this case, only JC) replaced by their matching expressions. Since JC matched

  ```
  <cs_person {<name N> <relation R> Rest1 Rest2}>
  ```

  and N was bound to ‘Joe Chung’, the resulting head is

  ```
  <cs_person {<name ‘Joe Chung’> <relation R> Rest1 Rest2}>
  ```

3.2 Execution Plan

Now that the MSI knows what objects it has to find at the sources, it builds a plan 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. This binding is achieved in two steps:

   (a) the following query is sent to whois:

   ```
   <binding_for_whois { <binding_for_R R> <binding_for_Rest1 Rest1>}>
     :- <person {<name ‘Joe Chung’> <dept ‘CS’> <relation R> | Rest1}>
   ```

---

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

[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 answer.

Labels binding_for_whois, binding_for_R and binding_for_Rest1 are simply placeholders that allow the MSI to pick out the desired information in the returned answer object (see next step).

(b) An "answer" object is received from whois. For the example of Section 2, the answer would look like:

```
<&a2,answer, set, {&a21, ...}>
      <&a21,binding_for_whois, set, {&a21r, &a21b}>
             <&a21r, binding_for_R, string, 'employee'>
             <&a21b,binding_for_Rest1, set, {&a21b1}>
                     <&a21b1, e_mail, string, 'chung@cs'>
      ⋮
```

From this the MSI can easily extract the bindings. For example, the first binding in the answer above is R = 'employee' and Rest1 = {<&a21b1, e_mail, string, 'chung@cs'>}.

2. Bindings for LN and FN can be obtained from one of the decompose_name 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:

```
<binding_for_cs {<binding_for_Rest2 Rest2>}>
  :- <employee {<first_name 'Joe'> <last_name 'Chung'> | Rest2}>
```

Notice again how it is very easy to use a value from whois as a label (i.e., relation attribute 'employee') in cs.

4. Once MSI obtains bindings for Rest2 as well, it generates objects that follow the pattern

```
<cs_person {<name 'Joe Chung'> <relation R> Rest1 Rest2}>
```

For example, considering the bindings we have illustrated so far, the MSI would generate the object of Figure 2.4. All the generated "cs_person" objects are packaged into the "answer" object which is then sent to the client.

4 The Mediator Specification Language

In this section we will present a number of MSL features. First, we discuss those that are especially important for querying sources with unknown structure and contents. Subsection 4.1 demonstrates the retrieval of schema information, and Subsection 4.2 shows how we avoid starting searches from top-level OEM objects. Subsection 4.3 then presents facilities for predicates, both built-in predicates as well as those exported by translators and mediators. Then, we present MSL features especially targeted to integration, for example, choosing between a number of alternative sources (Subsection 4.4), adding object identification on top of value-oriented sources (Subsection 4.5), and grouping heterogeneous source objects that refer to the same entity (Subsection 4.6). Due to space limitation we continue to focus on examples; the MSL syntax and semantics is given in Appendixes A and B.

4.1 Variables

The flexibility of MSL variables allows us to retrieve information about the labels and structure of a source. For example, the following rule (that can be viewed either as a query or as part of a mediator specification) retrieves the second-level labels of source med:

```
<second_level_label Level2_Label>
  :- <Top_Level_Label {<Level2_Label Data>}>@med
```

MSL's free use of variables allows us to move information that originally appears in a label field to a value field.[7] Assuming that the above rule stands for a query we will receive an answer object such as:

```
<&a2,answer, set, {&a21, &a22, ...}>
     <&a21, second_level_label, string, 'name'>
     <&a22, second_level_label, string, 'relation'>
     ⋮
```

An MSL variable may appear in place of value, label, and object-id fields without violating the typing requirements of the language. For example, assume that we want to create a mediator that exports each "employee" object of translator cs, but with the employee label being renamed to the title of the specific employee (e.g. professor, secretary, and so on). The mediator specification

```
<Title Rest> :- <employee {<title Title> | Rest}>@cs
```

binds the variable Title to the value of the "title" objects but then assigns the bindings of Title to the label fields of the top-level exported objects.

Note that a Title binding may not qualify as a valid label (e.g., Title may bind to a set value). For any such binding, MSL will not create a corresponding object of the integrated view. In general, MSL poses only a few typing restrictions on the use of variables. For example, it does not allow an object variable to appear in plave of a non-object variable. See [PGMU] for a complete list of MSL's typing restrictions.

4.2 Wildcards

The query of Section 3 searches for "cs_person" and "name" objects at particular object levels (e.g., name must be a second level label). The wildcard feature allows searches for objects at any level. For example, the following query retrieves all "cs_person" objects, regardless of their depth in the med object structure, provided that they have a direct or indirect "name" subobject with value 'Joe Chung'.

```
JC :- *JC:<cs_person {*<name 'Joe Chung'>}>@med
```

In general, the * may precede any tail pattern in a query or a mediator specification. When it precedes a top-level pattern (such as JC:<cs_person ...>) it allows the pattern to match with objects at any level in the source object structure. When it precedes a pattern (such as <name 'Joe Chung'>) enclosed by other patterns, it allows the pattern to match with any direct or indirect subobject of the object that matched the enclosing pattern.

[7] The type of the value field of the generated object is by default string

Clearly, the expressive power of wildcards comes with a performance price. If the underlying source(s) do not have facilities for performing wildcard searches, the mediator will have to retrieve entire sub-structures to do the search itself. For this reason by default query tails match only against mediator top-level objects. In this way, only users that wish to browse and explore sources can use the wildcard feature and pay the performance penalty.

4.3 Predicates

The Mediator Specification Interpreter has a number of *built-in* predicates, functions, and operators that perform the common arithmetic operations and comparisons of scalar values. The built-in predicates and operators often save the user from having to create external predicates. To illustrate, consider the following mediator that uses the `-`,`*`,`/`, and `=` operators in order to transform a Fahrenheit temperature into a Celsius temperature:

```
<location {<celsius C> Rest}> :-<location {<fahrenheit F> | Rest}>@tempsource
                                  AND C = (F-32)*5/9
```

In addition to the external predicates illustrated in Section 2.2, a mediator may also call on source predicates that perform source specific processing. For example, a bibliographic source may provide a function for approximately matching author names, `author_match(N, M)@biblio`. (Example: "Joe Chung" matches "J. Chung" matches "J. R. Chung.") This function can be included in a MSL definition. (We assume that the mediator implementor has access to facilities that help him discover what predicates each source may offer.) Furthermore, the mediator implementor may define predicates exported by the mediator. A mediator exported predicate is defined by MSL rules that have the predicate in the head.

4.4 Choosing Among Alternative Sources

In an environment with autonomous sources, information may be duplicated at several sources, with different retrieval costs. MSL offers *alternative rules* for specifying search order and avoiding visiting sources unnecessarily.

For example, assume that the sources `cs` and `whois` contain similar, but not necessarily identical, data about CS persons. Furthermore, assume that the source `cs` charges a service fee. In this case we may wish to have a mediator `save` that collects from `whois` all information about CS persons, while it collects from `cs` only information about persons that do not appear in `whois`. Then, when `save` is asked for information on a particular person, say Joe Chung, it will first try to retrieve Joe Chung's data from `whois`, and only if it does not find any will send a query to `cs`. Mediator `save` can be specified with an `ALT` construct as follows:

```
<cs_person{<name N> Rest}>
          :- ALT(N) (<person {<name N> <dept 'CS'> | Rest}>@whois ,
                     (decompose(N, LN, FN)
                      AND <person {<last_name LN> <first_name FN> | Rest}>@cs )
                     )
```

From a conceptual point of view, the `ALT` construct operates as follows : First, we try to obtain bindings for `N` and `Rest` from the first alternative, i.e., from

```
<person {<name N> <dept 'CS'> | Rest}>@whois
```

For every valid binding of the pair `N` and `Rest` a "cs_person" object is created, as dictated by the head of the rule. Furthermore, the structure `ALT(N)` indicates that all bindings of `N` are inserted in a set N_b that initially is empty. Using the `whois` data of Figure 2.3, we would obtain `N` bindings for "Joe Chung" and "Nick Naive."

Then we try to obtain pairs of bindings for `N` and `Rest` from the second alternative. For every binding n_i and $rest_i$ of `N` and `Rest` that we obtain from the second alternative we check whether n_i is already included in N_b. If it is not then we create a "cs_person" object using n_i and $rest_i$, otherwise we discard n_i and $rest_i$. For example, assuming the data of Figure 2.2, after the second alternative has been considered we will obtain `N` bindings for "Joe Chung," "John Hennessy," and "Pierre Huyn." The first of these is discarded because "Joe Chung" is already a member of N_b. At the end, we will have two bindings from `whois` (Chung, Naive) and two from `cs` (Hennessy, Huyn), and `save` will have four virtual objects. If there were more than two alternatives in the `ALT` structure, then the bindings of `N` that we obtain from the second alternative, will be included in the set N_b before we consider the third alternative, and so on.

Note that from a conceptual point of view it seems that the evaluation of the `ALT` structure required that Joe Chung's data were retrieved from both `whois` and `cs`. However, in practice, if a query asking for Joe Chung's data is given to `save`, the MSI matches the query against the `ALT` structure and creates a query plan that first tries to retrieve Joe Chung's data from `whois` and only if it does not find it at `whois` does it send a query to `cs`.

4.5 Specified Object-Ids

In the mediator specifications presented so far, no object-id is defined for the mediator objects. Therefore the MSI assigns random object-ids for the objects it exports; by convention they start with the symbol `&` and are guaranteed to be unique within the result of a query. In Figure 2.4, `&cp1` and `&mn1` are examples of these ids.

By convention, these arbitrary object ids can only be used by a client to infer sub-object relationships within an answer object. A client should not take one of these ids, e.g., `&cp1` and ask the mediator that generated it to find the object. We follow this convention so that mediators are not forced to keep a permanent database of object ids and the objects they correspond to.

If a mediator wishes to identify an exported object, then it should use *specified* object ids. These are ids that semantically identify an exported object. To illustrate, assume that the "name" field uniquely identifies "cs_person" objects exported by `med`. The following specification creates a mediator `med-oid` that exports the same objects as `med` but with each top-level object containing a specified object id:

```
<N cs_person V> :- <cs_person V:{<name N>}>@med
```

The head object pattern `<N cs_person V>` has three fields: `N` represents the specified object-id, `cs_person` is the label, and `V` is the value. A sample object exported by `med-oid` is:

```
<'Joe Chung', cs_person, set, {&mn1, &mrel1, ...}
              <&mn1, name, string, 'Joe Chung'>
              <&mrel1, relation, string, 'professor'>
```

Having received the specified object id `'Joe Chung'`, a client can at a later time submit the following query to the mediator to retrieve the object once again:

```
Person :- Person:<'Joe Chung' cs_person V>@med-oid
```

Notice that the mediator can answer this query with the same machinery it already has (see Section 3). By matching the query tail to the mediator head we obtain the equivalent query:

```
<'Joe Chung' cs_person V> :- <cs_person V:{<name 'Joe Chung'>}>@med
```

To evaluate this rule, the query

```
<binding_for_V V> :- <cs_person V:{<name 'Joe Chung'>}>@med
```

is submitted to `med`, and the desired object is fetched. Note that the latter query is purely value oriented, in the sense that it does not pose any conditions on object-ids at `med`.

By convention, specified object ids are strings that do not start with a single `&`. They can be meaningful to the end user, providing a semantic key for an object. [8] Specified ids have meaning beyond the mediator call that yielded them; however, they are not "permanent" in the sense that the underlying source or mediator can delete or modify the objects.

Specified ids can also be composed from several values In particular we use logic's ground terms [Ull89] to structure object-id's. For example, consider a mediator that exports objects whose contents are assembled from an "employee" object and a "department" object. The combined object may be given a specified id `join(N, D)`, where the *argument* `N` is the employee name, *the argument* `D` is the department identifier, and the *function symbol* `join` is simply a string. A composed id can later be used to re-assemble the object.

4.6 Using Specified Object-Ids to Group Objects

Suppose that we want to create a mediator `medall` that exports a "univ_person" object for every person that appears either in source `whois`, or `cs`, or on both. The mediator of Section 2.2 only generates a person object if objects with equivalent names appear in both sources. Therefore, to implement `medall` we would have to add two extra rules: one that exports an object for every person that appears in `whois` but does not appear on `cs`, and another that exports an object for every person that appears on `cs` but does not appear on `whois`. Unfortunately, this technique does not scale well to scenarios with many sources, since we need an exponential number of rules.

Specified object-ids offer a much better way to specify `medall`. Before we show how, consider the following simple example. Assume that we want to export an object with label `l` and two subobjects `<s1 1>` and `<s2 2>`. We may split the specification of the "l" object in two rules if we assign to the object an Object-Id, say `loid`, and have each of the two rules define a "part" of the "l" object:

```
<loid l {<s1 1>}> :- TRUE
<loid l {<s2 2>}> :- TRUE
```

The Object-Id `loid` forces the two rules to refer to the object identified by `loid`. The first (second) rule specifies that there is an "l" object identified by `loid` that has a subobject with label `s1` (`s2`) and value `1` (`2`). Note that the first (second) rule does not exclude the possibility that the `loid` object has subobjects other than the "s1" ("s2") object, thus allowing the other rule to add one more subobject to the "l" object.

Using this idea we can specify `medall` as follows:

```
<N univ_person V> :- <person V:{<name N>}>@whois
<N univ_person {<name N> <relation R> RestCS}>
   :- <R {<last_name LN> <first_name FN> | RestCS}>@cs
      AND decompose_name(N, LN, FN)
```

[8] If the specified id happens to start with a `&`, then a double `&` is used.

Each of the rules specifies some of the "univ_person" subobjects. For example, consider "Joe Chung" who appears both in `whois` and `cs`. In this case, each source contributes information to the `medall` object for `'Joe Chung'`. On the other hand, person "Nick Naive" only appears in `whois`, so only one rule contributes to the `medall` object for this person.

As a small point, notice that a "univ_person" object may have multiple "name" subobjects with the same value (i.e., `'Joe Chung'`). This is because each came from a different source and was assigned a different, randomly generated object id. The following mediator specification solves the problem by assigning the specified object-id `nameOID('Joe Chung')` to the "name" subobjects. (As an alternative, we could have defined a `UNIQUE` operator that eliminates all objects with identical label and value within a set, similar to the SQL operator.)

```
<N univ_person {<nameOID(N) name N> RestWhoIs}>
               :- <person {<name N> | RestWhoIs}>@whois

<N univ_person {<nameOID(N) name N> <relation R> RestCS}>
               :- <R {<last_name LN> <first_name FN> | RestCS}>@cs
                  AND decompose_name(N, LN, FN)
```

The outerjoin operator has also been suggested for grouping information about an entity. We believe that grouping based on specified object-id's is preferable to outerjoin grouping for a number of reasons:

1. it is more suitable to an object-oriented view definition language, because it is based on the fundamental notion of object identity.

2. it provides a clean and modular solution: If we want to integrate one more information source, we simply introduce one more rule.

3. grouping by object-id is very natural in MSL, allowing us to handle multiple integration problems simultaneously.

To illustrate the last point, consider the following example. Assume that the `cs` database has a large number of relations of the form `relation_name(last_name, first_name, ...)`. Furthermore, some persons may appear in more than one `cs` relation. For example, a teaching assistant appears both in the `employee` and the `student` relations. Consider now a mediator `medcs` that groups in a "cs_person" object all information about each person that appears in one or more of the `cs` relations. Furthermore, `medcs` groups all the "relation" objects for a person into a "relations" subobject of "cs_person." This mediator can be defined by the single rule:

```
<N cs_person{<nameOID(N) name N>
            <relsOID(N) relations {<relation R>}>
            RestCS
            }>
  :- <R {<last_name LN> <first_name FN> | RestCS}>@cs
     AND decompose_name(N, LN, FN)
```

Had we solved this problem by using an outerjoin operator, it would have been necessary to know in advance the names of all the relations involved. Whenever a new relation were introduced, we would need to update the outerjoin-based mediator specification. This is not the case with the MSL specification. Furthermore, we do not know of any outerjoin based specification that could group all the "relation" objects for a person into a single "relations" objects in a straightforward way.

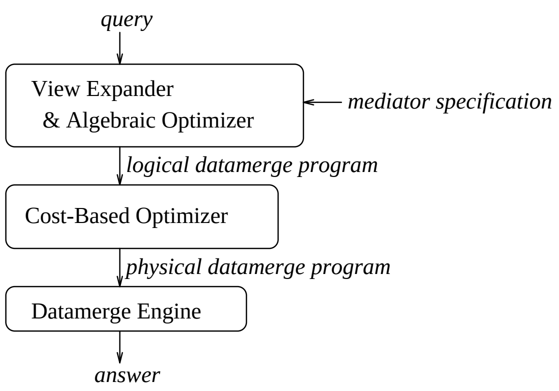

Figure 4.5: The basic architecture of MSI

5 Architecture and Implementation of MSI

MSI – the run-time component of MedMaker – processes a query using a pipeline with the following three components (see also Figure 4.5):

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.

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 "answer" object.

3. the *datamerge engine* executes the plan and produces the required "answer" object.

In the following two subsections we will discuss each component and the languages at each interface. In Subsection 5.3 we will discuss some of the challenging optimization problems that MedMaker has identified.

5.1 View Expansion and Algebraic Optimization

The VE&AO matches queries against the mediator specification, using the query/specification matching algorithm illustrated in Section 3. The VE&AO outputs a *logical datamerge program* that is a set of MSL rules specifying the answer. The only difference between a logical datamerge program and an ordinary query is that the former refers and imposes conditions only on the source objects. For example, in Section 3 we matched the query 3.1 with the mediator specification of `med` and we produced the logical datamerge program 3.2.

VE&AO is also responsible for algebraic optimizations, the most important being the translation of conditions on the mediator's virtual objects to conditions on the source objects. This is an important optimization because it allows us to "push" conditions to the sources. However, unlike view expansion and algebraic optimization in conventional databases, translating a condition on the virtual objects to the source objects is not straightforward because we may not have complete knowledge of the "schemas" of the sources. To illustrate this point, assume that the following query, that retrieves the data of 3rd year students, is sent to `med` (defined in Section 2):

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

We cannot tell in advance whether the "year" object comes from source `cs` or from source `whois`. In particular, when we match the query against the mediator specification, the `<year 3>` pattern can match either with `Rest1` or with `Rest2`. In the former (latter) case, VE&AO assumes that `<year 3>` is a subobject of the "person" ("employee" or "student") objects that come from `whois` (`cs`). The two possible cases give rise to the following two rules, that constitute the logical datamerge program:

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

5.2 Execution

The optimizer receives the logical datamerge program and creates a *physical datamerge program* that consists of a list of (possibly parameterized) queries that will be sent to the sources, and a variant of MSL rules, called *physical datamerge rules*, which specify the mechanics of constructing the answer. To illustrate, after the optimization, the first rule of the logical datamerge program above may result in the following physical datamerge program:

```
QUERY LIST :
Query Qw
 <binding_for_whois {<binding_for_N N>
                            <binding_for_R R>
                            <binding_for_Rest1 Rest1>}>
           :- <person {<name N> <dept 'CS'> <relation R> | Rest1:{<year 3>}}>
Query Qcs($R, $LN, $FN)
 <binding_for_Rest2 Rest2>
                         :- <$R {<last_name $LN> <first_name $RN> | Rest2}>
PHYSICAL DATAMERGE RULES
<cs_person {<name N> <relation R> Rest1 Rest2}>
   :- ( <binding_for_whois {<binding_for_name N>
                            <binding_for_relation R>
                            <binding_for__Rest1 Rest1>}>@(Qw,whois)
        AND(N ->) decompose_name(N, LN, FN) )
      AND(R,LN,FN ->) <bind_for_Rest2 Rest2>@(Qcs(R,LN,FN),cs)
```

Let us first explain the physical datamerge rule above (in general, there may be more than one rule). The symbols `AND(... ->)` that appear in the tail of the rule specify the order in which the patterns of the tail should be evaluated (left to right, in this example) . In particular, first the query `Qw`, that is fully listed in the `QUERY LIST` section, will be sent to the source `whois`. The received objects will be matched with the pattern `<binding_for_whois ...>` and from the matching we will obtain bindings for `N` and `R`. For every binding of `N` we will invoke `decompose_name` to obtain bindings for `LN` and `FN`. Then, for every triplet of `R`, `LN`, and `FN` bindings we will replace the `$R`, `$LN`, and `$FN` *parameters*, that appear in the parameterized query `Qcs`, with the corresponding bindings, and we will issue the "instantianted" query `Qcs` to `cs`. For every query `Qcs` that we issue we match the result with `<bind_for_Rest2 ...>` to obtain bindings for `Rest2`. Finally we combine the bindings of `N`, `R`, `Rest1`, and `Rest2` to produce the object specified by the head of the rule.

Note that the `AND(`*variables* `->)` annotations implied a nested-loops join policy, where first we obtain bindings for the variables from the query/pattern to the left, and for every other binding we produce another instance of the query to the right. Other join policies are also available. For example, `AND(`*variables*`)` implies that the two patterns on the sides of `AND` are evaluated independently, and then a join on the variables is taken locally.

Finally, the datamerge engine executes the physical datamerge program produced by the cost optimizer. However, since the physical datamerge program is expressed in a high-level declarative notation that is difficult to directly execute, the datamerge engine executes it into two steps. The first step, called *physical datamerge program compilation*, reads the program and produces a *datamerge graph* that is an algebraic description of the physical datamerge program. Then, the *datamerge graph engine* directly executes the low-level operations that are dictated by the nodes of the datamerge graph. The simplicity of OEM and MSL make it possible to translate any physical datamerge program into a small and concise set of low-level operators (nodes). Examples of these operators are "send query Q to source S," "find the subobjects of object O," "create an object with identifier *oid*, label L, and value V," and so on.

5.3 Optimization Challenges

There are usually several physical datamerge programs that correspond to a logical datamerge program. The optimizer has to select the "optimal" physical datamerge program. However, optimization of a powerful object-oriented language that operates on autonomous and heterogeneous information sources [LOG93] is much harder than the optimization of traditional SQL queries on a conventional database. We discuss in the next paragraphs some of the most important challenges, along with preliminary ideas we are working on.

The limited query capabilities of the underlying sources may prohibit even simple algebraic optimizations, such as "push selections and projections down." For example, we saw earlier that the sample query `Qw` contains a condition on the "year" object and a condition on the "dept" object. However, source `whois` may not be able to evaluate the condition on "year", although it might be able to evaluate the "dept" condition.

As a solution to this problem, we have developed [PG] a language for describing the queries that are supported by a translator. We have also developed an algorithm that, given a query Q and the description of the queries supported by the translator, can tell whether the query Q is directly supported by the translator or whether there is a query Q' supported by the translator that contains all the information that is necessary for computing the result to the given query Q. For example, if the query `Qw` is given to this algorithm , the algorithm will inform the mediator that the translator does not support `Qw` but it supports a query `Qd` that poses a condition on "dept" only. Consequently, the mediator can use `Qd` and evaluate the condition on "year" locally (i.e., the mediator will select objects with the appropriate "year" subobject from the data it receives from the source.)

The optimizer must be able to operate with or without statistics and cost information from the translators. Nevertheless, even if the translators are willing to provide statistics and cost information, the optimization problem is not trivial because of the object-oriented nature of MSL queries. Our current efforts concentrate on "reducing" the problem of describing statistics of an object-oriented query to the well-understood techniques for describing statistics of relational queries. In particular, the optimizer may *probe* the translator by sending a (possibly parameterized) query, and the translator may return an estimate of the query's cost and of the number of bindings of various variables that appear in the query. For example, the translator may return the following upon being probed by `Qw`:

```
<binding_for_whois {<binding_for_N N[80]>
                    <binding_for_R R[3]>
                    <binding_for_Rest1 Rest1>}>[100]
          :- <person {<name N> <dept 'CS'> <relation R> | Rest1:{<year 3>}}>
Response:10 sec
```

The annotation `[100]` indicates that approximately 100 "binding_for_whois" objects are expected in the result. The annotations `[80]` and `[3]` indicate there will be approximately 80 different "`N`" values and 3 different "`R`" values. Using such statistics and cost information, we believe we can calculate cost estimates of physical datamerge programs.

Optimization is much harder when the translator does not provide cost and statistics information [DKS92]. In this case, the optimizer may use ad-hoc heuristics (e.g., the outer patterns of the join order are the ones that have the greatest number of conditions) or the optimizer tries to build its own statistics database [DKS92, ZL94], that is based on results of previous queries and on sampling. Another option is to specify optimization "manually", assuming that the mediator specifier knows in advance the best general way to execute (most of the) queries. He may then provide hints, e.g. what order to use for joins, what conditions to push to sources, and so on. We are currently working on ways to specify these general hints.

The most powerful features of MSL, such as the wildcards, and the grouping by object-id, pose additional problems for query optimization. For example, it may be unreasonable to expect that a translator will be able to provide statistical information for all "wildcard" queries that may be asked. On the absence of statistics, the optimizer will have to infer the optimal plan using indirect schemes.

Undoubtedly, optimization of OO languages in an environment of heterogeneous sources poses many problems that have to be solved if we want to have powerful and flexible integration frameworks, such as MedMaker. We strongly believe that the demand for powerful integration tools will provide the necessary motivation for the development of solutions to the above mentioned problems.

6 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, LM93]). A difference between OEM and conventional OO models is that OEM is much simpler and does not have a strong type 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" [PGMW]). Indeed, the simplicity of the model has facilitated the implementation of translators [PGMW], mediators [Ire], and client programs that browse heterogeneous object structures [H^+]. In addition, we believe that simple models, like OEM, have an advantage over complex models when used for integration (as it is pointed out in [BLN86]), since fewer operations are needed for the construction of an integrated view. Consequently, 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[Rud92, 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) and allows creation and manipulation of object-id's (ODMG's query language [Cat94], for instance, does not allow explicit creation and manipulation of object-id's).

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 [D.M86], and F-Logic [KL89]. HiLog first proposed – under a logic framework – the idea of mixing schema and data information.[9] It also demonstrated the reduction of nested structures into first-order structures, a feature of OEM and MSL that will be illustrated in Appendix B. Nevertheless, we should not see MSL as a front-end to a datalog system, since MSL contains structures (such as the object variables, the "`Rest`" variables, wildcards, and so on) that facilitate the manipulation of objects and the integration of heterogeneous systems. When these structures are modeled directly in datalog, they result in a large number of rules that are difficult to write and to process. We have also adopted from the area of logic and deductive databases the usage of object-id's that indicate how the corresponding object was derived from source objects [KKS92, KL89]. [AK89] uses invented object-id's for the manipulation and creation of sets, and as it also happens with MSL, they conclude that object-id based set formation replaces the need for explicit grouping operators. Finally, ILOG [HY90] provides declarative creation and manipulation of object identifiers that are based on Skolem functors.

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 knowledge of the schema (the example of Section 5 demonstrated the tradeoff between performance and partial knowledge of the schema).

Though systems that integrate well-structured conventional databases exist (e.g., [A^+91, K^+93, BLN86, LMR90, T^+90, Gup89, FLNS88, ACHK93]) and recently systems for the integration of sources with minimal structure have also appeared [Fre, RJR94, S^+93], we do not know of view definition based systems ([LM93, A^+91, Ber91, CWN94, FLNS88] and others) that handle the whole spectrum of information sources simoultaneously, 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

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

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, FN92]. 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.

Currently, we are working on the implementation of MedMaker. We are building it on top of the CSL/SSL client/server interaction support libraries [PGMW]. In parallel with the implementation of the – relatively straightforward – view expansion and datamerge engine modules we investigate optimization techniques, as described in Subsection 5.3.

Acknowledgements

We are grateful to Jennifer Widom, Joachim Hammer, Vasilis Vassalos, Anand Rajaraman, Pierre Huyn, Dallan Quass, 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.

[ACHK93] Y. Arens, C.Y. Chee, C.-N. Hsu, and C.A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal of Intelligent and Cooperative Informations Systems*, 2:127–58, June 1993.

[AK89] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In *Proc. ACM SIGMOD Conference*, pages 159–73, Portland, OR, May 1989.

[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 *Proceedings of the 1st International Workshop on Interoperability in Multidatabase Systems*, pages 22–29, Kyoto, Japan, April 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.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proc. ACM SIGMOD Conference*, pages 16–52, Washington, DC, USA, May 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.

[Cat94] R.G.G Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, 1994. with contributions from Tom Atwood et.al.

[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.

[DKS92] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in heterogeneous DBMS. In *Proc. VLDB Vonference*, pages 277–91, Vancouver, Canada, August 1992.

[D.M86] 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.

[FLNS88] P. Fankhauser, W. Litwin, E.J. Neuhold, and M. Screfl. Global view definition and multidatabase languages: two approaches to database integration. In *Research into Networks and Distributed Applications. European Teleinformatics Conf.*, pages 1069–1082, Vienna, Austria, April 1988.

[FN92] P. Fankhauser and E. Neuhold. Knowledge based integration of heterogenous databases. Technical report, Technische Hochschule Darmstadt, 1992.

[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`.

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

[H+] J. Hammer et al. Information translation, mediation, and mosaic-based browsing in the TSIMMIS system. To appear in the Exhibits Program of SIGMOD '95, San Jose, CA. Available by anonymous ftp at `db.stanford.edu` as the file `~/pub/hammer/1994/mobie-demo-proposal.ps`.

[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. *International Journal of Intelligent and Cooperative information Systems*, 2:51–83, 1993.

[HY90] R. Hull and M. Yoshikawa. Ilog: Declarative creation and manipulation of object identifiers. In *Proc. VLDB Conference*, pages 455–68, Brisbane, Australia, August 1990.

[Ire] K.L. Ireland. TSIMMIS union and join mediators detailed design specification. Available by anonymous ftp at `db.stanford.edu` as the file `~/pub/ireland/mediator_design.ps`.

[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, San Diego, California, June 1992.

[KL89] M. Kifer and G. Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. In *Proc. ACM SIGMOD Conference*, 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.

[LM93] Y. Lee and S. Moon. Heterogeneous integration method for multidatabase system. In *Nineteenth EUROMICRO Symposium on Microprocessing and Microprogramming*, pages 265–72, Barcelona, Spain, September 1993.

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

[LOG93] H. Lu, B.-C. Ooi, and C.-H. Goh. Multidatabase query optimization: Issues and solutions. In *Proc. RIDE-IMS '93*, pages 137–43, Vienna, Austria, February 1993.

[Mar93] D. S. Marshak. Lotus Notes release 3. *Workgroup Computing Report*, 16:3–28, 1993.

[Mos83] M.G. Moser. An overview of NIKL, the new implementation of KL-ONE. Technical Report 5421, Bolt, Beranek, and Newman, Inc. Cambridge, MA, 1983.

[MY89] R. MacGregor and J. Yen. LOOM:integrating multiple AI programming paradigms. *Proc. Intl. Joint Conf. on Artificial Intelligence*, August 1989.

[O$^+$93] B. Oki et al. The Information Bus—an architecture for extensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993.

[PG] Y. Papakonstantinou and A. Gupta. Query translation in heterogeneous information sources. Available by anonymous ftp at `db.stanford.edu` as the file `~/pub/papakonstantinou/1995/querytran.ps`.

[PGMU] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman. Medmaker: A mediation system based on declarative specifications(extended version). Available by anonymous ftp at `db.stanford.edu` as the file `~/pub/papakonstantinou/1995/medmaker-extended.ps`.

[PGMW] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. To appear in ICDE 95. Available by ftp at `db.stanford.edu`, file `~pub/papakonstantinou/1994/icde95.ps`.

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

[RJR94] R. Rao, B. Janssen, and A. Rajaraman. GAIA technical overview. Technical report, Xerox Palo Alto Research Center, 1994.

[Rud92] E. Rudensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proc. VLDB Conference*, pages 187–98, Vancouver, Canada, August 1992.

[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.

[Wie87] G. Wiederhold. *File Organization for Database Design.* McGraw Hill, New York, 1987.

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

[ZL94] Q. Zhu and P.-A. Larson. A query sampling method for estimating local cost parameters in a multidatabase system. In *Proc. IEEE Data Engineering Conf.*, pages 144–53, Houston, TX, February 1994.

Appendix

A MSL Syntax

A complete syntax for the MSL language appears in Figure A.6. Note two features that have not been presented in the body of the paper: First, the keyword `isatom` may precede an object pattern of the MSL rule tail, forcing the pattern to bind to atomic objects only. Second, the pattern $\$\langle variable \rangle$ may appear in the head of an MSL rule, in place of an object pattern or an object variable. For example, the rule

```
$JC :- <JC cs_person {<name 'Joe Chung'>}>@source
```

declares that the mediator exports, as top-level objects, the `source` objects identified by `$JC`. Intuitively, `$` acts as a dereferencing operator. If the pattern $\$\langle variable \rangle$ appears nested inside an object pattern then the objects identified by $\langle variable \rangle$ become subobjects of the objects "created" by the pattern. For example, the rule

```
<JC cs_person {$R}> :- <JC cs_person {<R L V>}>@source
```

declares that every object identified by `R` must be exported as a subobject of a "cs_person" object identified by corresponding bindings of `JC`.

B Formal Semantics of MSL Specifications

In this section we present the model-theoretic semantics of MSL. Besides its usefulness in formally defining the meaning of mediator specifications and queries, they also show the connection between MSL and logic and they provide insight into the implementation of MSI. The formal semantics is based on the intuition that OEM and MSL are essentially a form of logic that has been appropriately reformulated in order to express object nesting, object identity, and the combination of schema and data information. We substantiate this intuition by reducing OEM data and MSL mediator specifications[10] and queries into *first-order* formulas. In particular, we first describe how OEM objects can be reduced to an equivalent *relational representation* that is based on first-normal-form relations. Then, we show how mediator specifications can be reduced to datalog rules that contain function symbols. The minimal model of the datalog rules describes the relational representation of the object structure that is exported by the mediator. The same idea applies to the specification of query semantics, since we can view queries as mediator specifications that define the query result.

The transformation of an arbitrary MSL specification to a logic program is accomplished in two steps:

1. we reduce every MSL rule into one or more *normal-form* MSL rules, that is, MSL rules conforming to the following restrictions:

   (a) they contain only 4-field patterns that explicitly specify the object-id, label, type, and value of the source and mediator objects.

[10] We assume that the MSL specification satisfies the obvious safety and typing constraints, such as "no object variable can appear where a non-object variable is expected". A complete list of the safety and typing restrictions appears in Subsection B.3.

```
(0)   ⟨specification⟩       ::= Rules ⟨rules⟩ [External ⟨external pred defs⟩]
(1)   ⟨rules⟩               ::= ⟨rules⟩ ⟨rule⟩
                              | ε
(2)   ⟨rule⟩                ::= ⟨head⟩ : − ⟨condition list⟩ .
(3)   ⟨head⟩                ::= ⟨object⟩   − − − generatedobject
                              | ⟨predicate name⟩ ( ⟨argument list⟩ )   − − − exported predicate
(4)   ⟨object⟩              ::= [$] ⟨variable⟩
                              | < [⟨object id⟩] ⟨label⟩ [⟨type⟩] ⟨value⟩ >
(5)   ⟨object id⟩           ::= ⟨term⟩
(6)   ⟨term⟩                ::= ⟨function symbol⟩ ( ⟨term list⟩ )
                              | ⟨simple term⟩
(7)   ⟨term list⟩           ::= ⟨term list⟩ , ⟨term⟩
                              | ⟨term⟩
(8)   ⟨simple term⟩         ::= ⟨variable⟩
                              | ⟨constant⟩
(9)   ⟨label⟩               ::= ⟨simple term⟩
(10)  ⟨type⟩                ::= ⟨simple term⟩
(11)  ⟨value⟩               ::= ⟨simple term⟩
                              | { ⟨subobjects list⟩ }
(12)  ⟨subobjects list⟩     ::= ⟨subobjects list⟩ ⟨object⟩
                              | ε
(13)  ⟨argument list⟩       ::= ⟨argument list⟩ , ⟨term⟩
                              | ⟨term⟩
(14)  ⟨condition list⟩      ::= ⟨condition list⟩ AND ⟨condition⟩
                              | ⟨condition⟩
                              | TRUE
(15)  ⟨condition⟩           ::= ⟨object condition⟩ @ ⟨site⟩   − − − top object condition
                              | ⟨predicate name⟩ ( ⟨argument list⟩ )   − − − external predicate condition
                              | ⟨predicate name⟩ ( ⟨argument list⟩ ) @ ⟨site⟩   − −source predicate condition
(16)  ⟨object condition⟩    ::= [isatom] [∗] [⟨object variable⟩ :]
                                 < [⟨object id⟩] ⟨label⟩ [⟨type⟩] ⟨value condition⟩ >
(17)  ⟨value condition⟩     ::= ⟨simple term⟩
                              | [⟨set variable⟩ :] { ⟨obj cond list⟩ [| ⟨rest⟩] }
(18)  ⟨rest⟩                ::= ⟨set variable⟩ [: { ⟨obj cond list⟩ [| ⟨rest⟩] }]
(19)  ⟨obj cond list⟩       ::= ⟨object cond list⟩ ⟨object condition⟩
                              | ε
(20)  ⟨site⟩                ::= ⟨simple term⟩
(21)  ⟨external pred defs⟩  ::= ⟨external pred defs⟩ ⟨external pred def⟩
                              | ε
(22)  ⟨external pred def⟩   ::= ⟨pred name⟩ ( ⟨type list⟩ ) ( ⟨binding list⟩ )
                                 impl by ⟨C function name⟩
(23)  ⟨type list⟩           ::= ⟨type list⟩ , ⟨type⟩
                              | ⟨type⟩
(24)  ⟨binding list⟩        ::= ⟨binding list⟩ , ⟨binding⟩
                              | ⟨binding⟩
(25)  ⟨binding⟩             ::= bound|free
```

Figure A.6: MSL's Syntax

(b) they do not contain object variables or rest variables such as the variables `Rest1` and `Rest2` of `med`'s specification (see Section 2).

(c) all value variables that appear more than once (in the MSL rule) bind to atomic constants only. We can be sure that a value variable binds to atomic constants only if it also appears in place of a label, object-id, or type field, or it appears inside an object pattern preceded by the keyword `isatom`.

2. we reduce every normal-form MSL rule into one or more datalog rules.

Subsection B.1 provides the rules for computing the relational representation of the object structure exported by an OEM source (i.e., translator or mediator). Subsection B.1.2 illustrates the reduction of a normal-form MSL rule to datalog rules. Subsection B.1.3 illustrates the reduction of an MSL rule to a normal-form MSL rule. Subsections B.2 to B.5 describe the complete algorithm for the reduction of an arbitrary MSL specification to a datalog program. Finally, Subsection B.6 provides conditions that guarantee the well-defined semantics of rules containing external predicates. (In particular, we follow the ideas presented in [BR86].)

B.1 Relational Representation of OEM Objects

We can represent the OEM object structure that is exported by an OEM source using three relations, named *top*, *object*, and *member*. For example, Figure B.7 demonstrates the relational representation of part of the object structure of translator `cs` (see Figure 2.2 for the corresponding part of the object structure). The relational representation of an object structure can be derived mechanically by the following three straightforward rules:

1. If at the source s there is an object with object-id oid, label l, type t, and atomic value (i.e., non-set value) v, then we introduce a tuple

$$object(s, oid, l, t, v)$$

2. If at the source s there is an object o with object-id oid, label l, and a set value that consists of the objects $o_1, o_2, \ldots, o_n$ that are identified by $oid_1, oid_2, \ldots, oid_n$ then we introduce the tuples

$$\begin{array}{l} object(s, oid, l, set, set) \\ member(s, oid, oid_1) \\ member(s, oid, oid_2) \\ \quad\vdots \\ member(s, oid, oid_n) \end{array}$$

   where *set* is a special value indicating that the object identified by oid is a set object.

3. If at the source s the object identified by oid is a top-level object, we introduce the tuple

$$top(s, oid)$$

We treat structured object-id's in the same way that we treat any object-id. For example, if at source `src` the object identified by `h1(a52,'Smith')` is a member of the object identified by `h2('Jones')` we introduce the tuple:

$$member(\texttt{src}, \texttt{h2('Jones')}, \texttt{h1(a52,'Smith')})$$

top

SRC	*OID*
cs	&e1
cs	&e2
cs	&s3
⋮	

object

SRC	*OID*	*Label*	*Type*	*Value*
cs	&e1	employee	*set*	*set*
cs	&f1	first_name	string	'Joe'
cs	&l1	last_name	string	'Chung'
cs	&t1	title	string	'professor'
cs	&rep1	report_to	string	'John Hennessy'
cs	&e2	employee	*set*	*set*
cs	&f2	first_name	string	'John
cs	&l2	last_name	string	Hennessy'
cs	&t2	title	string	'chairman'
⋮				

member

SRC	*OID*	*SubOID*
cs	&e1	&f1
cs	&e1	&l1
cs	&e1	&t1
cs	&e1	&rep1
cs	&e2	&f2
cs	&e2	&l2
cs	&e2	&t2
⋮		

Figure B.7: Relational representation of `cs`'s exported object structure

The above rules can be used in reverse as well, i.e. given the relational representation of a set of sources we can use the above rules to derive the OEM object structures of the sources. Note that the relational representation must satisfy obvious constraints (fully described in Subsection B.1.1) such as the constraint that the existence of a tuple

$$member(s, oid, oid_1)$$

implies the existence of a tuple

$$object(s, oid, l, set, set)$$

for some label l. Indeed, the mediator specification must be such that for every query Q the relational representation of the answer satisfies the constraints for being reducible to an OEM answer object. If the required constraints are not satisfied the system informs the user of the constraint violation. External predicates and predicates exported by the sources and/or the mediator can also be incorporated in the relational representation of sources. In particular,

- if source s exports the source-specific predicate *pred* and

  $$pred(arg_1, \ldots, arg_n)@s$$

  holds, then the fact

  $$pred(s, arg_1, \ldots, arg_n)$$

  holds in the equivalent relational representation.

- if the external predicate *pred* is satisfied for the arguments $arg_1, \ldots, arg_n$, i.e. the external predicate

  $$pred(arg_1, \ldots, arg_n)$$

  holds, then the fact

  $$ext\_pred(arg_1, \ldots, arg_n)$$

  holds in the equivalent relational representation.

B.1.1 Constraints on the Relational Representation of an OEM Object Structure

The *top*, *member*, and *object* relations that represent an OEM object structure must satisfy the following constraints:

1. The "source" and "object-id" attributes of the *object* relation constitute a key for the tuples of *object*.

2. The "type" attribute of an *object* tuple is *set* if and only if the "value" of the tuple is also *set*.

3. If there is a tuple
$$member(s, oid, oid_1)$$
there must also be a tuple
$$object(s, oid, l, set, set)$$

4. If there is a tuple
$$object(s, oid, l, t, c)$$
where $c \neq set$, then there is no tuple
$$member(s, oid, oid_1)$$

5. If there is a tuple
$$member(s, oid, oid_1)$$
there is also a tuple
$$object(s, oid_1, l, t, v)$$

B.1.2 Example of Reducing Normal-Form MSL Rules to Logic Rules

Consider the following trivial specification of mediator `simple` that for every top-level object of source `src`, with object-id `O` and label `L`, exports a "top-label" object whose value is the binding of `L` and whose object-id is the binding of `O`. Note that the following rule happens to be a normal-form rule since it uses only 4-field object patterns, does not contain object or rest variables, and the value variable `V` appears only once.

```
<O top_label string L> :- <O L T V>@src        (1)
```

The "informal" semantics of MSL described in Section 2 requires that first we obtain bindings for the variables that are used in the head of the MSL rule. We formalize in logic our notion of obtaining bindings for `L` and `O` by introducing a predicate *bind* that "collects" valid pairs of bindings for the pair `L` and `O`. In general, *bind* collects bindings for the variables that appear in the tail of the MSL rule.

A logic rule that corresponds to the tail of the MSL rule defines the tuples of *bind*:

$$bind(O, L) :- top(\texttt{src}, O) \wedge object(\texttt{src}, O, L, T, V)$$

The tail of the above logic rule was created by applying the following two straightforward principles:

- if the MSL pattern
$$<\langle oid\rangle\ \langle label\rangle\langle type\rangle\langle value\rangle>$$
appearing in the MSL rule tail refers to source $\langle src\rangle$,[11] and $\langle value\rangle$ is either a variable or a constant, we put in the tail of the datalog rule the *literal*
$$object(\langle src\rangle, \langle oid\rangle, \langle label\rangle, \langle type\rangle, \langle value\rangle)$$

- if the MSL pattern
$$<\langle oid\rangle \ldots>\ @\langle src\rangle$$
appears in the MSL rule tail, we put in the tail of the logic rule's tail the *literal*
$$top(\langle src\rangle, \langle oid\rangle)$$

Now that we have obtained bindings for the variables that appear in the head, we write the rules that create the (relational representation) of the object structure of `simple`.

$$top(\texttt{simple}, O) :- bind(O, L)$$
$$object(\texttt{simple}, O, \texttt{top\_label}, \texttt{string}, L) :- bind(O, L)$$

The above rules were created by applying two straightforward principles for reducing the MSL rule head in datalog rules:

- If the MSL pattern
$$<\langle oid\rangle\langle label\rangle\langle type\rangle\langle value\rangle>$$
appears in the head of the mediator specification of $\langle medname\rangle$, and $\langle value\rangle$ binds to constants only, we introduce the logic rule[12]
$$object(\langle medname\rangle, \langle oid\rangle, \langle label\rangle, \langle type\rangle, \langle value\rangle) :- bind(\ldots)$$
Note, the arguments of the predicate *bind* are the variables that appear in the head of the MSL rule.

- If the top-level object
$$<\langle oid\rangle \ldots>$$
appears in the head of the mediator specification of $\langle medname\rangle$, then we introduce the datalog rule
$$top(\langle medname\rangle, \langle oid\rangle) :- bind(\ldots)$$

B.1.3 Example of Reducing MSL Rules to Normal-Form

Whenever a 2-field pattern appears at an MSL rule we have to replace it with a 4-field pattern. Let us consider the following mediator `fl` that exports an object "top_label" for every label of a top-level object that it finds at source `src`.

```
<top_label L> :- <O L T V>@src
```

[11]Note, the pattern either is followed by a `@`$\langle src\rangle$ or is nested inside a pattern that is followed by a `@`$\langle src\rangle$

[12]The normal-form assumption guarantees that $\langle value\rangle$ does not bind to a set of objects, and thus it can be safely used in the logic rule.

The above rule is transformed into the normal-form MSL rule

```
<G top_label string L> :- <O L T V>@src AND genoid_{L→G}(L,G)
```

The type of the generated "top_label" objects is `string` because the values of the "top_label" objects come from labels. The object-id field is filled with a variable `G`. Then, for every valid binding of the variable `L` we have to generate one binding of `G`. This is accomplished by the predicate $genoid_{L \to G}$ that appears above. $genoid_{L \to G}(l, g)$ associates a unique g with each l.

B.2 Categorization of Variables

We will use the following categorization of MSL variables in the expression of safety and typing restrictions (Section B.3), and in the reduction of arbitrary MSL rules to normal-form MSL rules (Section B.4).

The variables of an MSL rule are classified into the following categories (note that a variable may be classified in more than one of the following categories):

1. *atomic variables*, which appear
   - in the object-id, label, and type fields of object patterns of the head and/or the tail of the MSL rule,
   - in the value field of an object pattern of the head and/or the tail of the MSL rule, where the type field of the object pattern has a value different from `set`.
   - in the arguments of external and/or source-specific predicates, either in the head or the tail of the MSL rule,
   - following a `$` in the MSL rule's head,
   - in the value field of object patterns that are preceded by the keyword `isatom`. For example, in the following MSL rule `V` is an atomic variable:

     ```
     <L V> :- isatom:<L V>@src
     ```

2. *object variables*, which appear in place of ⟨*object variable*⟩ in the ⟨*object condition*⟩ production of the syntax.
3. *rest variables*, which appear in place of ⟨*set variable*⟩ in the ⟨*rest*⟩ production of the syntax. For example, the variables `Rest1` and `Rest2` of the mediator specification of `med` (see Section 2) are rest variables.
4. *value variables*, which appear in place of value fields of object patterns of the tail. Value variables are further classified into the following three categories:
   (a) *value variables that bind to atomic constants only*: they are value variables that are also atomic variables.
   (b) *value variables that bind to sets (of objects) only*: they are value variables that appear
      i. in place of a ⟨*set variable*⟩ in the ⟨*value condition*⟩ production of the syntax,
      ii. in place of ⟨*variable*⟩, not preceded by `$`, in the ⟨*object*⟩ production of the syntax,
      iii. in place of a value field of an object pattern that has the constant `set` in its type field

For example, in the following MSL rule V1 binds to sets only

```
O:- <L V1:{<l1 X>}>@src
```

(c) *value variables that bind to atomic constants or sets*: all the value variables that can not be classified in one of the categories above.

B.3 Safety and Typing Restrictions

MSL enforces the safety restriction that all variables appearing in the head of an MSL rule must also appear in its tail. MSL also enforces the following typing restrictions on the use of variables. The expression of the typing restrictions is based on the variable's categorization that was described in Section B.2:

1. No object, rest, or value variable that binds to sets only can also be an atomic variable. For example, the following MSL rule is prohibited because the variable O appears as an object variable (in the head) and as an atomic variable (in the tail).

   ```
   O :- <O V>@src
   ```

2. No atomic variable may appear in place of $\langle variable \rangle$ in the $\langle object \rangle$ production without being preceded by a $. For example, the atomic variable A may not appear as the head of the MSL rule:

   ```
   A :- <A L T V>@src
   ```

3. No object variable, rest variable, or value variable that binds to sets only may appear two times in the MSL rule tail. For example, the following rule is prohibited

   ```
   O:- <l1 V:{<l2 v1>}>@src AND <l3 V>@src
   ```

   Whenever comparison of non-atomic variables is required appropriate set comparison (or object comparison) predicates should be used.

4. No variable V that appears in place of the value field of a 3-field or 2-field object pattern of the head, may appear in the MSL rule tail only inside a term $f(\ldots V \ldots)$, or only as an argument of a source-specific predicate. (The reason is that we can not infer its type.)

B.4 Reduction of Arbitrary MSL rules to Normal-Form MSL Rules

The reduction of arbitrary MSL rules to normal-form MSL rules proceeds in the following steps (that must be executed in the specific order):

1. we reduce all 2-field and 3-field object patterns that appear in the MSL rule's tail to 4-field object patterns. In particular, we fill the missing object-id fields with invented unique variables, and we fill the missing type fields with the appropriate type, when we can infer it, or with an invented unique variable when we can not. See Subsection B.4.1 for details.

2. we eliminate all value variables that may bind both to sets and atomic constants (Subsection B.4.2).

3. we associate all object, rest, and value variables that may bind to sets to object-id variables that are sufficient for describing the bindings of the object, rest, and value variables. See Subsections B.4.3 for details.

4. we reduce every 2-field head object pattern, i.e. a pattern that does not specify the object-id of the generated object, to a 3-field pattern. We fill the missing object-id field with an invented variable that stands for the automatically generated object-id. See Subsection B.4.4 for the details of object-id generation.

5. we eliminate all object, rest, and value variables that bind to sets (see Subsection B.4.5).

6. we reduce every 3-field head object pattern to a 4-field pattern by finding out what the type of the generated object must be. See Subsection B.4.6 for the details.

B.4.1 Reduction of 2-field and 3-field Tail Object Patterns to 4-field Patterns

We replace all 2-field and 3-field object patterns that appear in the MSL rule tail with 4-field patterns that explicitly specify object-id, label, type, and value. In particular:

- every 3-field pattern

$$<\langle oid\rangle\langle label\rangle\langle value\rangle>$$

  is replaced by a 4-field pattern

$$<\langle oid\rangle\langle label\rangle \; \langle inferred\ type\rangle \; \langle value\rangle>$$

  where the $\langle inferred\ type\rangle$ field is computed by the following rules:

  - if $\langle value\rangle$ is a subobjects list, i.e. a list of object patterns enclosed by {}, then the inferred type is `set`.
  - otherwise, the $\langle inferred\ type\rangle$ is an invented variable T that does not appear anywhere else in the MSL rule.[13]

- every 2-field pattern

$$<\langle label\rangle\langle value\rangle>$$

  is replaced by a 4-field pattern

$$<O \; \langle label\rangle \; \langle inferred\ type\rangle \; \langle value\rangle>$$

  where O is an invented variable that does not appear anywhere else in the MSL rule, and $\langle inferred\ type\rangle$ is computed by the rules stated above.

B.4.2 Elimination of Value Variables that Bind to Both Sets and Atomic Constants

In order to eliminate a value variable `V` that appears in the MSL rule head, and may bind both to atomic constants and sets of objects, we replace the corresponding MSL rule with two MSL rules: The first one has conditions that guarantee that `V` binds to atomic constants only, and the second

[13] We could infer for more cases what the $\langle inferred\ type\rangle$ is, but for simplicity we do not.

one has conditions that guarantee that `V` binds to sets of objects only. (Later, we will describe the elimination of value variables that bind to sets only.) In particular, if the pattern P

$$<\langle oid\rangle\langle label\rangle\langle type\rangle \texttt{ V } >$$

appears in the MSL rule tail of rule R, we replace R with two rules R_1 and R_2 that are identical to R with the exception that

1. R_1 differs from R in that P is replaced by a pattern

   $$\texttt{isatom} :<\langle oid\rangle\langle label\rangle\langle type\rangle \texttt{ V } >$$

   Note, if `V` appears in place of $\langle variable\rangle$ in the $\langle object\rangle$ production without being preceded by a `$`, then the rule R_1 must be discarded because it violates the typing conditions.

2. R_2 differs from R in that P is replaced by a pattern

   $$<\langle oid\rangle\langle label\rangle\langle type\rangle \texttt{ V} : \{\} >$$

   Note, if `V` appears twice in the tail of R_2 we discard R_2 because we do not allow value variables that bind to sets to appear two times in the tail.

We repeatedly apply the above transformation until there is no rule that contains a variable `V` that may bind both to atomic constants and sets of objects.

B.4.3 Associations of Non-Atomic Variables to Atomic Variables

Since our eventual goal is to reduce MSL rules, that may contain variables that bind to non-atomic structures such as objects and sets of objects, to datalog rules, that contain variables that bind to atomic constants only, we associate every "non-atomic" variable V_n with one or more atomic variables $V_1^a, \ldots, V_m^a$ that will be sufficient for simulating the behavior of V_n. For example, we associate every object variable O with an object-id term T that binds to the object-id's of the objects that O binds to. The complete list of variable associations is the following:

1. if `O` is an object variable that appears in the MSL rule head, then we associate `O` with an object-id term $\langle oid\rangle$ such that the object pattern

   $$\texttt{O} :<\langle oid\rangle\langle label\rangle\langle type\rangle\langle value\rangle>$$

   appears in the MSL rule's tail.

2. if `R` is a rest variable that appears in the MSL rule head, then we associate with `R` an object-id term $\langle oid\rangle$ such that the object pattern

   $$<\langle oid\rangle\langle label\rangle \texttt{ set } \{...|\texttt{R}...\} >$$

   appears in the MSL rule's tail. We will name $\langle oid\rangle$ "the *oid-rest* term associated with `R`". Furthermore, we associate `R` with the object-id terms $\langle oid\rangle_i$ of the subobjects of $\langle oid\rangle$ that must be excluded from `R`. We name each $\langle oid\rangle_i$ "an *oid-excluded* term associated with `R`".

   For example, consider the mediator specification

```
<N L T R> :- <O L set {<I name string N> | R}>@src
```

The oid-rest term associated with `R` is `O`. The oid-excluded term – in this case only one – associated with `R` is `I`.

3. If the value variable `V` that binds to sets appears in the MSL rule head we associate `V` with an object-id term $\langle oid \rangle$ such that the object pattern

   `<`$\langle oid \rangle\langle label \rangle\langle type \rangle$ `V>`

   or

   `<`$\langle oid \rangle\langle label \rangle\langle type \rangle$ `V:{...}>`

   appears in the MSL rule tail.

B.4.4 Object-Id Generation

We replace every 2-field object pattern that appears in the MSL rule head with a 3-field pattern, by inventing an object-id variable that appears in place of the object-id field. I.e., if the 2-field pattern

$$<\langle label \rangle\langle value \rangle>$$

appears in the MSL rule head, it is replaced by the 3-field pattern

$$<\texttt{G}\ \langle label \rangle\langle value \rangle>$$

where `G` is an invented variable that does not appear in the MSL specification. Furthermore, we add to the MSL rule tail a predicate $genoid_G(\bar{X}, G)$ that generates bindings for G. $genoid_G(\bar{X}, G)$ is considered a built-in predicate, and consequently we do not have to provide its definition and implementation. The vector $\bar{X}$ is computed as follows:

1. we insert in $\bar{X}$ all atomic variables and value variables that bind to atomic constants only and appear in the MSL rule head.

2. if the object variable `O` appears in the MSL rule's head, then we insert in $\bar{X}$ the object-id term $\langle oid \rangle$ that is associated with `O`.

3. if the rest variable `R` appears in the MSL rule's head, then we insert in $\bar{X}$ the oid-rest term $\langle oid \rangle$ that is associated with `R`.

4. if the value variable V that binds to sets only appears in the MSL rule's head, then we insert in $\bar{X}$ the object-id term $\langle oid \rangle$ that is associated with `V`.

The predicate $genoid_G$ associates exactly one unique binding g of `G` with every tuple $(x_1, \ldots, x_n)$ of bindings of $\bar{X}$.

B.4.5 Elimination of Object, Rest, and Value Variables that Bind to Sets

We apply the following rules to eliminate all object, rest, and value variables that bind to sets.

1. if the object variable `O` appears in the MSL rule's head, and `O` is associated with the object-id term $\langle oid \rangle$, we replace instances of `O` in the MSL rule's head with $\$\langle oid \rangle$.[14] We remove `O` from the MSL rule's tail.

2. if the rest variable `R` appears in the MSL rule's head, and `R` is associated with the oid-rest term $\langle oid \rangle$ and the oid-excluded terms $\langle oid \rangle_1, \ldots, \langle oid \rangle_n$ then

   (a) we replace the pattern

   $$<\langle oid \rangle \langle label \rangle \langle type \rangle \ \{\langle exluded\ subobjects \rangle \ | \mathtt{R} :[\{\langle rest \rangle\}]\} >$$

   of the MSL rule tail with

   $$<\langle oid \rangle \langle label \rangle \langle type \rangle \ \{\langle excluded\ subobjects \rangle < \mathtt{R\ L\ T\ V} >[\langle rest \rangle]\} >$$

   where `L`, `T`, and `V` must be invented unique variables.

   (b) we add in the MSL rule tail the following conjunction of external predicates

   $$\mathtt{neqoid}(R, \langle oid \rangle_1) \wedge \ldots \wedge neqoid(R, \langle oid \rangle_n)$$

   where the built-in predicate *neqoid* is satisfied if its two arguments are not identical.

   (c) we replace the object patterns

   $$<\langle oid' \rangle \langle label \rangle \langle type \rangle \ \{\langle subobjects\ list \rangle \ R \ \langle subobjects\ list \rangle'\} >$$

   of the MSL rule head with

   $$<\langle oid' \rangle \langle label \rangle \langle type \rangle \ \{\langle subobjects\ list \rangle \ \$\mathtt{R} \ \langle subobjects\ list \rangle'\} >$$

   (d) we replace the object patterns

   $$<\langle oid' \rangle \langle label \rangle \langle type \rangle \ \mathtt{R} >$$

   of the MSL rule head with

   $$<\langle oid' \rangle \langle label \rangle \langle type \rangle \ \{\$\mathtt{R}\} >$$

   (e) if `R` appears in place of the MSL rule head, we replace it with `$R`.

3. if the value variable `V` that binds only to sets appears in the MSL rule's head then

   (a) we replace the pattern

   $$<\langle oid \rangle \langle label \rangle \langle type \rangle \ \mathtt{V} : \{\langle obj\ cond\ list \rangle\} >$$

   of the MSL rule tail with

   $$<\langle oid \rangle \langle label \rangle \langle type \rangle \ \{< \mathtt{V\ L\ T\ Val} > \langle obj\ cond\ list \rangle\} >$$

   where `L`, `T`, and `Val` must be invented unique variables.

[14] In order to keep simple the syntax of Figure A.6 we allow only variables to follow the `$` symbol. If $\langle oid \rangle$ is not a variable, but it is a constant or a term we should introduce in the body of the rule an equality $V = \langle oid \rangle$ where V is an invented variable. Then we can place V after the `$`

(b) we replace object patterns

$$<\langle oid\rangle'\langle label\rangle\langle type\rangle\ \{\langle subobjects\ list\rangle\ \texttt{V}\ \langle subobjects\ list'\rangle\}>$$

of the MSL rule head with

$$<\langle oid\rangle'\langle label\rangle\langle type\rangle\ \{\langle subobjects\ list\rangle\ \texttt{\$V}\ \langle subobjects\ list'\rangle\}>$$

(c) we replace object patterns

$$<\langle oid'\rangle\langle label\rangle\langle type\rangle\ \texttt{V}>$$

of the MSL rule head with

$$<\langle oid'\rangle\langle label\rangle\langle type\rangle\ \{\texttt{\$V}\}>$$

(d) if `V` appears in place of the MSL rule head, we replace it with `$V`.

Finally, for all value variables that bind to atomic constants only, while they do not appear

1. in objects preceded by `isatom`
2. in objects with a type field that is a constant other that `set`
3. in arguments of predicates that appear in the tail

we precede the object where they appear with `isatom`.

B.4.6 Reduction of 3-field Head Object Patterns to 4-field Patterns

We replace every 3-field object pattern that appears in the MSL rule head with a 4-field pattern, where we have inferred what the type field must be. In particular, we replace every 3-field head object pattern

$$<\langle oid\rangle\langle label\rangle\langle value\rangle>$$

with a 4-field pattern

$$<\langle oid\rangle\langle label\rangle\langle inferred\ type\rangle\langle value\rangle>$$

where the *inferred type* field is computed by the following rules:

1. if $\langle value\rangle$ is an integer constant, i.e. a number that does not contain a decimal point, then the $\langle inferred\ type\rangle$ is `integer`. Similarily, if $\langle value\rangle$ is a real number, then the $\langle inferred\ type\rangle$ is `real`. Finally, if $\langle value\rangle$ is a string, then the $\langle inferred\ type\rangle$ is `string`.
2. if $\langle value\rangle$ is a subobjects list, i.e. a list of object patterns enclosed by the curly braces {}, then the $\langle inferred\ type\rangle$ is `set`.
3. if $\langle value\rangle$ is a variable V then the $\langle inferred\ type\rangle$ is computed by the following rules:

   (a) if there is an MSL rule tail object pattern

   $$<\langle oid'\rangle\langle label'\rangle\langle type'\rangle\ \texttt{V}>$$

   then the $\langle inferred\ type\rangle$ field is identical to $\langle type'\rangle$.

(b) if there is an MSL rule tail object pattern where V appears on place of the object-id, label, or type field, i.e. if we can find any of the following patterns

$$< \texttt{V} \ \langle label' \rangle \langle type' \rangle \langle value' \rangle >$$
$$< \langle oid' \rangle \ \texttt{V} \ \langle type' \rangle \langle value' \rangle >$$
$$< \langle oid' \rangle \langle label' \rangle \ \texttt{V} \ \langle value' \rangle >$$

then the $\langle inferred\ type \rangle$ field is `string`.

(c) if the MSL rule tail invokes the external predicate

$$pred(\langle arg \rangle_1, \ldots, \langle arg \rangle_{i-1}, V, \langle arg \rangle_{i+1}, \ldots, \langle arg \rangle_n)$$

then there must be an external predicate definition

$$pred(\langle type \rangle_1, \ldots, \langle type \rangle_{i-1}, \langle type_V \rangle, \langle type \rangle_{i+1}, \ldots, \langle type \rangle_n)(\langle bindings\ list \rangle)$$

and we can deduce that the $\langle inferred\ type \rangle$ is $\langle type_V \rangle$.[15]

B.5 Reduction of Normal-Form MSL Rules to Datalog Rules

In this section we describe the reduction of an MSL specification that consists of normal-form MSL rules to a datalog program (that may contain function symbols). Every normal-form MSL rule $R_i^N, 1 \leq i \leq r$, where r is the number of normal-form MSL rules, reduces to a set of datalog rules R_i^D. The concatenation of all the datalog rules of sets R_i^D, together with three generic rules, that are specified in Subsection B.5.3, constitute the datalog program that specifies the semantics of the MSL specification. Note, from now on whenever we write "MSL rule" we will imply "normal-form MSL rule".

For every MSL rule R_i^N of the mediator specification we generate a set R_i^D of datalog rules. R_i^D contains:

1. one rule of the form
$$bind_i(\bar{W}) : -\langle datalog\ condition \rangle$$
where $\bar{W}$ is the set of variables that appear in the MSL rule head. The $\langle datalog\ condition \rangle$ is derived by reducing the MSL rule's tail to a conjunction of datalog literals, following the steps described in Subsection B.5.1. Note, whenever there is no confusion about which rule we are talking about we will refer to $bind_i$ simply as $bind$.

2. l rules of the form
$$head_j(\bar{Y}) : -bind(\bar{W})$$
where the $head_j$ datalog rule heads are derived by reducing the MSL rule's head to datalog, following the steps described in Subsection B.5.2.

B.5.1 Reduction of the MSL Rule's Tail

The MSL rule's tail is translated into a conjunction of datalog literals, i.e. a conjunction of one or more *object*, *member*, *top*, external predicates, and source specific predicates, that specify which are the acceptable bindings for the set $\bar{W}$ of interesting variables. The following rules formalize the reduction of the MSL rule tail into a conjunction of literals, by associating reduction actions with the productions of the MSL syntax (Figure A.6):

[15] The existing MSL specification does not infer types for source specific predicates. See Section B.3

- $\langle$*condition list*$\rangle$: Every condition c_i , $1 \leq i \leq m$ of the condition list of an MSL rule tail results in a conjunction C_i of one or more datalog literals. The tail of the datalog rule is the conjunction of all the literals that appear in all C_i , $1 \leq i \leq m$.

- $\langle$*condition*$\rangle$: An MSL condition c results in a conjunction C of one or more datalog literals, depending on whether the condition is an object condition or a predicate:

  1. if $\langle$*condition*$\rangle$ is a *"top object condition"* (see Figure A.6), i.e. has the form

     `<`$\langle oid\rangle \ldots$`>@`$\langle src\rangle$

     the condition reduces to a conjunction C of the literals resulting from the object condition `<`$\langle oid\rangle \ldots$`>` (see action associated with $\langle$*object condition*$\rangle$) and the literal

     $$top(\langle src\rangle, \langle oid\rangle)$$

  2. if $\langle$*condition*$\rangle$ is an *"external predicate condition"* (see Figure A.6) that has the form

     `pred(`$\langle arg\rangle_1$`, `$\langle arg\rangle_2$`, ..., `$\langle arg\rangle_n$`)`

     where $\langle arg\rangle_i$, $1 \leq i \leq n$ are the arguments of the predicate, the predicate condition results in the datalog literal (that can be seen as a trivial conjunction)

     $$\texttt{ext\_pred}(\langle arg\rangle_1, \langle arg\rangle_2, \ldots, \langle arg\rangle_n)$$

     We changed the name of the predicate from `pred` to `ext_pred` to avoid confusing it with some source predicate with the same name.

  3. if $\langle$*condition*$\rangle$ is a *"source predicate condition"* that has the form

     `pred(`$\langle arg\rangle_1$`, `$\langle arg\rangle_2$`, ..., `$\langle arg\rangle_n$`)@`$\langle src\rangle$

     where $\langle arg\rangle_i$, $1 \leq i \leq n$ are the arguments of the predicate and $\langle src\rangle$ is a simple term that stands for the source, the source predicate reduces to the datalog literal (that can be seen as a trivial conjunction):

     $$\texttt{pred}(\langle src\rangle, \langle arg\rangle_1, \langle arg\rangle_2, \ldots, \langle arg\rangle_n)$$

- $\langle$*object condition*$\rangle$: The reduction of the object condition to a literal conjunction depends on whether the value of the object condition binds to atomic constants only, sets only, or anything:

  1. if the object condition has the form

     `<`$\langle oid\rangle\langle label\rangle\langle type\rangle\langle constant\rangle$`>`

     it reduces to the datalog literal

     $$object(\langle src\rangle, \langle oid\rangle, \langle label\rangle, \langle type\rangle, \langle constant\rangle)$$

     where $\langle src\rangle$ is the term that describes the source against which the object condition is evaluated.

  2. if the object condition has the form

     `isatom:<`$\langle oid\rangle\langle label\rangle\langle type\rangle\langle variable\rangle$`>`

it reduces to the conjunction of datalog literals

$$object(\langle src\rangle, \langle oid\rangle, \langle label\rangle, \langle type\rangle, \langle variable\rangle) \wedge ext\_neqset(\langle variable\rangle)$$

where $\langle src\rangle$ is the term that describes the source against which the object condition is evaluated and *ext_neqset* is a predicate that is satisfied when its argument does not have the value *set*.

3. if the object condition has the form

   `<`$\langle oid\rangle\langle label\rangle\langle type\rangle$ `{<`$\langle oid_1\rangle$`...> <`$\langle oid_2\rangle$`...>` ... `<`$\langle oid_n\rangle$`...>}>`

   then we introduce in the datalog condition

   (a) the literal

   $$object(\langle src\rangle, \langle oid\rangle, \langle label\rangle, set, set)$$

   (b) the literals that describe that the object identified by $\langle oid\rangle$ has subobjects that are identified by $\langle oid\rangle_i$, $1 \leq i \leq n$

   $$\begin{array}{l} member(\langle src\rangle, \langle oid\rangle, \langle oid\rangle_1) \\ member(\langle src\rangle, \langle oid\rangle, \langle oid\rangle_2) \\ \vdots \\ member(\langle src\rangle, \langle oid\rangle, \langle oid\rangle_n) \end{array}$$

   (c) the literals that result from the reduction of the subobject patterns `<`$\langle oid_1\rangle$`...>` ... `<`$\langle oid_n\rangle$`...>` to conjunctions of datalog literals.

B.5.2 Reduction of MSL Rule's Head

We reduce every MSL rule's head to a set of datalog rules heads and we attach to each datalog rule head the tail $bind(\bar{W})$ thus getting a set of datalog rules that correspond to the MSL rule.

- if the head is an "exported predicate" (see Figure A.6) that has the form

  $$\texttt{pred}(\langle arg_1\rangle, \ldots, \langle arg_n\rangle)$$

  where $\langle arg_1\rangle, \ldots, \langle arg_n\rangle$ are the arguments, we create the datalog rule

  $$\texttt{pred}(\langle medname\rangle, \langle arg\rangle_1, \ldots, \langle arg_2\rangle) : -bind(\bar{W})$$

  where $\langle medname\rangle$ is the name of the mediator we specify.

- if the head has the form

  $$\$\langle oid\rangle$$

  then we introduce the datalog rules

  $$\begin{array}{l} top(\langle oid\rangle) : -bind(\bar{W}) \\ pick(\langle medname\rangle, \langle src\rangle, \langle oid\rangle) : -bind(\bar{W}) \end{array}$$

  where $\langle src\rangle$ is a simple term (variable or constant) that describes the source of the objects identified by $\langle oid\rangle$ ($\langle src\rangle$ can be trivially found by inspecting the MSL rule's tail). The predicate *pick* fetches the object identified by $\langle oid\rangle$ together with all its subobjects.

- if the head is a "generated object" that has the form

  $\mathtt{<}\langle \mathit{oid}\rangle\langle \mathit{label}\rangle\langle \mathit{type}\rangle\ \langle \mathit{generated\ value}\rangle\mathtt{>}$

  and results in one or more datalog rules depending on the form and properties of the $\langle \mathit{generated\ value}\rangle$:

  – if $\langle \mathit{generated\ value}\rangle$ is a constant `c` or a variable `V` (that, by now, we can be sure that it binds to atomic constants only) then we introduce the datalog rule

  $$\mathit{object}(\langle \mathit{medname}\rangle, \langle \mathit{oid}\rangle, \langle \mathit{label}\rangle, \langle \mathit{generated\ value}\rangle) : -\mathit{bind}(\bar{W})$$

  – if $\langle \mathit{generated\ value}\rangle$ is a subobjects list that has the form

  $$\{<\langle \mathit{oid}_1\rangle \ldots>\ \ldots\ <\langle \mathit{oid}_n\rangle \ldots>\ \$\langle \mathit{refoid}_1\rangle\ \ldots\ \$\langle \mathit{refoid}_m\rangle\}$$

  [16] we introduce:

  1. the datalog rule
  $$\mathit{object}(\langle \mathit{medname}\rangle, \langle \mathit{oid}\rangle, \langle \mathit{label}\rangle, \mathit{set}, \mathit{set})$$
  2. the datalog rules that specify that the objects identified by
  $$\langle \mathit{oid}_1\rangle, \ldots, \langle \mathit{oid}_n\rangle, \langle \mathit{refoid}_1\rangle, \ldots, \langle \mathit{refoid}_m\rangle$$
  are subobjects of the object identified by $\langle \mathit{oid}\rangle$
  $$\mathit{member}(\langle \mathit{medname}\rangle, \langle \mathit{oid}\rangle, \langle \mathit{oid}_i\rangle) : -\mathit{bind}(\bar{W})$$
  $$\mathit{member}(\langle \mathit{medname}\rangle, \langle \mathit{oid}\rangle, \langle \mathit{refoid}_j\rangle) : -\mathit{bind}(\bar{W})$$
  where $1 \leq i \leq n$ and $1 \leq j \leq m$.
  3. the datalog rules that result from the reduction of the head object patterns
  $$<\langle \mathit{oid}_1\rangle \ldots> \ldots <\langle \mathit{oid}_n\rangle \ldots>$$
  to datalog conditions.
  4. the datalog rules that "copy" the objects identified by $\langle \mathit{refoid}_1\rangle, \ldots, \langle \mathit{refoid}_m\rangle$ from $\langle \mathit{src}\rangle$ to $\langle \mathit{medname}\rangle$
  $$\mathit{pick}(\langle \mathit{medname}\rangle, \langle \mathit{src}\rangle, \langle \mathit{refoid}_j\rangle) : -\mathit{bind}(\bar{W})$$
  where $1 \leq j \leq m$.

B.5.3 Copying Objects from Source to Destination

The following three generic rules are attached to every datalog program and are used for copying an object together with its subobjects from a source S to a mediator M:

$$\mathit{object}(D, O, L, T, V) : -\mathit{pick}(D, S, O) \wedge \mathit{object}(S, O, L, T, V)$$
$$\mathit{member}(D, O_1, O_2) : -\mathit{pick}(D, S, O_1) \wedge \mathit{member}(S, O_1, O_2)$$
$$\mathit{pick}(D, S, O_2) : -\mathit{pick}(D, S, O_1) \wedge \mathit{member}(S, O_1, O_2)$$

[16] Without loss of the generality we assume that the `$` patterns follow the `<...>` patterns.

B.6 Evaluable External Predicates

Rules that contain external predicates may not be evaluable. For example, consider the external predicate `decompose_name` that was described in Section 2. We may not have a rule such as

```
<name {<last_name LN> <first_name FN>}> :- decompose_name(N, LN, FN)
```

because the predicate `decompose_name` can not produce bindings for `LN` and `LN` without first being given a binding for `N`.

We formalize the notion of evaluable rules that contain external predicates, as follows: A rule is evaluable if there is a permutation of the conditions of the rule tail, such that if the external predicate p appears as the i-th condition of the tail, then there is an implementation of p such that in the place of bound arguments appear terms that contain constants and variables that appear in the conditions that precede p.