Merging Ontologies in the Context Mediation Framework

Befekadu Ayenew

Working Paper CISL# 2002-09

May 2002

Composite Information Systems Laboratory (CISL)
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02142

Merging Ontologies in The Context Mediation Framework
By
Befekadu Ayenew

Submitted to the
Department of Electrical Engineering and Computer Science

May 24, 2002

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Context Mediation integrates heterogeneous data sources by comparing the contexts associated with the sources and resolving any semantic conflicts. Presently, context mediation can be done on data sources only if they subscribe to the same ontology. We propose a merging strategy that will allow us to extend mediation to sources that elevate to differing ontologies. This strategy takes a divide-and-conquer approach by breaking down the problem into smaller mediation problems that can be processed by the current mediation system. These sub-problems will be determined through an alignment of the concerned ontologies. The rules of these alignments will be dictated by the conflicts that need to be resolved. Once the sub-problems have been solved, the complete solution to the mediation problem is constructed by merging each output under the conditions that were imposed in the original mediation problem.

Thesis Supervisor: Stuart Madnick
Title: John Norris Maguire Professor of Information Technology
and Professor of Engineering Systems

Table of Contents

Introduction

Often times, it is necessary for applications to retrieve related data from more than one data source. A good example of this would be aggregation sites whose main purpose is comparing the prices offered by different vendors. These sites collect the prices for a particular product from different vendors, compare these prices and present the results to the user in a coherent manner. However, it is not always clear whether or not the data from these data sources is in the same context. For instance, different vendors could list their prices in different currencies; if there is indeed contextual disparity between sources, it is necessary to make the data coherent with one another before doing the comparisons.

1.1 Problem Description

Let us consider how we would merge the following two tables containing financial information on companies. Our goal is to use the two tables and obtain the total assets from one table and the net income from the other table.

1.1.1 Identical Contexts

First, let us consider the case where the contexts of the two relations are identical. In other words, the data from these two tables is coherent and can be compared without any further mediation.

DstreamAF	
NAME_COMPANY	**TOTAL_ASSETS**
DAIMLER-BENZ	103548992
NTT	7037243392

DiscAF	
NAME	**NET_INCOME**
DAIMLER-BENZ	615000
NTT	83396

Table 0.1 DstreamAF and DiscAF in the same context

In this case, we can get the result we want simply by performing a simple JOIN on the two tables:

```
select DStreamAF.NAME, DStreamAF.TOTAL_ASSETS,  DiscAF.NET_INCOME
        from Dstream, DiscAF where DstreamAF.NAME = DiscAF.COMPANY_NAME
```

1.1.2 Identical views of the world

Now, let us suppose the two tables espouse a common view of the world. In other words, the people who constructed the two tables share an identical way of modeling the data. For instance, both tables agree that a company has a name and this name can vary depending on the naming format being used. In addition both tables are restricted to a number of naming conventions that are familiar to both data sources. We will assume that we have procedures that allow us to convert the names from one format into another. We have just described two tables that share a common view of the world and yet can have different contexts. Figure 1.2 contains an example of two such tables.

DStreamAF	
COMPANY_NAME	**TOTAL_ASSETS**
DAIMLER-BENZ	103548992
NTT	7037243392

DiscAF	
NAME	**NET_INCOME**
DAIMLER BENZ CORP	615000000
NIPPON TELEGRAPH & TELEPHONE CORP	83396000

Table 0.2 DstreamAF and DiscAF in different contexts

Using our conversion procedures, we can determine that 'DAIMLER-BENZ' and 'NTT' in DstreamAF context are equivalent to 'DAIMLER BENZ CORP' and 'NIPPON TELEGRAPH & TELEPHONE CORP' in DiscAF context. Therefore, we can reconcile the contextual disparity between DstreamAF.COMPANY_NAME and DiscAF.NAME and simply join the two tables.

1.1.3 Different views of the world

So far we have discussed how to merge the two tables if they have identical contexts or if they at least have a common view of the world. The last and perhaps the most difficult case pertains to tables with different views of the world. In other words, the data in the two tables is not modeled in the same way. Hence, the two tables do not pick their contexts from a common set of contexts, and they can have their own representations for contexts that are otherwise identical.

For instance, let us take the two tables in Figure 1.1. We will once again assume that the two tables are using the same naming conventions. In the first case, the two tables had the same view of the world; therefore, the contexts for the two tables were exactly identical. If the two tables had different views of the world, the contexts will have different representations even if they were equivalent. A simple example would be the names of the naming conventions. Although they both represent the same naming format, both views of the world might have their own terms for the same naming format. Therefore, even though the contexts from the two tables are equivalent, they are still incoherent with one another and mediation is still not possible.

Now, let us consider the two tables in Figure 1.2. Once again, we will not be able to do the required conversions since the two tables have different views of the world. The DstreamAF context does not recognize the terms used to refer to the naming conventions used by the DiscAF context and vice versa. Therefore, routine context mediation is not sufficient to reconcile these two naming formats and make the company names from the two tables coherent with each other.

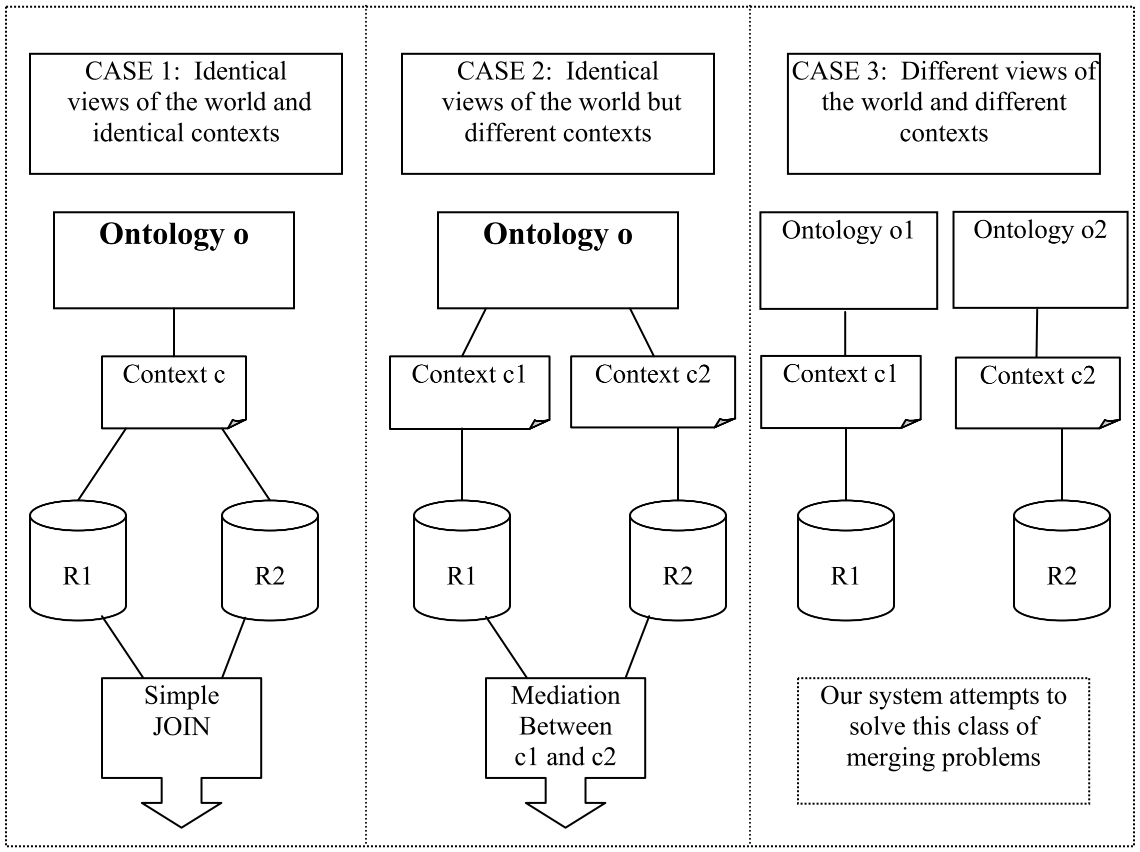

CASE 1: Identical views of the world and identical contexts
Ontology o
Context c
R1
R2
Simple JOIN
CASE 2: Identical views of the world but different contexts
Ontology o
Context c1
Context c2
R1
R2
Mediation Between c1 and c2
CASE 3: Different views of the world and different contexts
Ontology o1
Ontology o2
Context c1
Context c2
R1
R2
Our system attempts to solve this class of merging problems

Figure 1.0 A summary of the three classes of merging problems

The formal representation of the context for each data source is derived from the object model the data source subscribes to. This object model is known as an ontology, and it denotes a particular way of viewing the world. All data sources that subscribe to the same ontology have a common view of the world. The first two scenarios we considered in 1.1.1 and 1.1.2 were both examples of this class of context problems; we were able to resolve these problems making use of the fact that they all had contexts that were derived from identical ontologies.

In the third case, we have two data sources that subscribe to two different ontologies. Since contexts are based on ontologies, the problem of reconciling contexts becomes one of reconciling ontologies. Hence, the major challenge is resolving any semantic conflicts between the ontologies and making them coherent with one another. Figure 1.1 illustrates these three classes of merging problems.

The purpose of this thesis is to propose a solution to this problem and to implement a mechanism, which will let users issue queries that use data sources from multiple ontologies. The proposed solution will still comply with the specifications of the current context mediation framework, and it will attempt to use as much as possible of the current implementation.

1.2 Related Work

The problem of ontology heterogeneity is not new. Extensive research has already been done in the area of ontology management. Various tools for merging and aligning have been developed over the last few years. Although several tools have been built, most of them have been marginally successful because they have failed to deal with the semantic mismatches effectively. In fact, most of the earlier ontology alignment and merging tools have opted to focus on resolving only syntactic conflicts, and that has rendered them useless in handling domains with conceptual heterogeneity.

A good example for this is Ontomorph, a tool that sought to resolve mostly language level mismatches by converting ontologies into a common format [5]. Since resolving semantic conflicts usually requires human knowledge, most of the semantic matching enabled tools seek the assistance of domain experts in decision making on merging and aligning operations.

A rare exception to this is ONIONS (Ontological Integration of Naive Sources), which attempted to integrate ontologies using a stratified design of an ontology library system [5]. This system contained formalized generic ontology, which were classified based on description logic. Furthermore, this system had intermediate modules that contained most of the general conceptualizations of a domain. Ontologies were matched with ontologies in this library system during the alignment and merging processes. Quite predictably, this system was limited to ontology on medical terminology, and it was of little or no use for domains with no relevant ontology in the library system.

One of the earliest tools designed to resolve conceptual heterogeneity with human assistance was Chimaera, a tool that provided limited support to the user by pointing to possible alignments and merging [12]. Chimaera could resolve some semantic differences, and it even had a conflict detection utility that alerted the user whenever an illegal merging or alignment operation was performed. Although Chimaera was mostly inaccurate, it laid the grounds for the birth of the much improved and more advanced Smart.

Smart, which later became Prompt, is a system that provides a semi-automatic approach to ontology merging and alignment by involving the user in conflict resolution [12]. It also supplies the user with more suggestions than Chimaera, and it has sharper conflict detection. Its biggest improvement, however, is the specificity of its suggestions and its ability to read the concept hierarchy in the ontology more effectively. Based on its understanding of how closely concepts are related, it can suggest possible alignment and merging operations more successfully. In fact, studies have shown that, on average, Prompt gives 30% more correct suggestions than Chimaera [12].

The merging strategy presented in this thesis will use a somewhat similar approach to reconcile the ontologies and resolve any semantic conflicts between ontologies. By the time the user tries to execute a query, all the concerned ontologies will be coherent with one another and ready for mediation.

Although a large variety of semantic conflicts can arise between ontologies, this mechanism will attempt to resolve only mismatches caused by the use of synonyms to refer to semantically equivalent objects in different ontologies. These are a very common type of conflict between ontologies, and we will use them to demonstrate how the key components of the mechanism will deal with any type of semantic heterogeneity. A more powerful ontology alignment strategy that can solve a larger number of semantic mismatches will be presented in the later chapters as a possible extension to this implementation.

In the next chapter, we will provide an overview of GCMS, the current implementation of the context mediation system. In the following two chapters, we will present the design of our proposed system and give a description of its implementation. In chapter 5, we will use a more detailed motivational example to demonstrate how the system works. The final chapter will contain the design for a more effective approach to the problem of ontology heterogeneity, other possible extensions and concluding remarks.

2 Global Context Mediation System (GCMS)

The Context Interchange (COIN) group at the Sloan School of Management does research on the integration of heterogeneous data sources using context mediation. Context mediation is a strategy in which semantic conflicts among heterogeneous systems are not identified a priori, but are detected and reconciled by a *Context Mediator* through comparison of the contexts associated with any two systems engaged in data exchange [7]. Context mediation allows the execution of a query over multiple data sources with different contexts, and this powerful capability lends itself to an effective implementation of *an application* or a group of heterogeneous data sources tied together by a common ontology.

The most recent implementation of COIN, the Global Context Mediation System (GCMS) supports this notion of applications. GCMS is a complete end-to-end system, which is capable of taking in a query, doing the mediation and query execution before finally returning the appropriate result. Following, we will briefly discuss the features and capabilities of the current GCMS system in order to lay grounds for the ensuing discussion on the necessary amendments to the system.

GCMS consists of three major components:

1. applications
2. the abduction engine
3. query planner & executioner.

2.1 Applications

A GCMS application is a group of data sources subscribing to a common ontology. An application is made up of the following elements [7]:

1. Ontology
2. Context Definitions
3. Data Sources
4. Elevations
5. Conversion Functions

2.1.1 Ontology

A COIN ontology, like any other data model, represents a particular conceptualization of a domain. An ontology is made up of semantic types, attributes and modifiers. Semantic types are objects that denote independent class types in the ontology. The attributes of a semantic type are other semantic types that are used to define the properties of this type. For instance, in the excerpt of the financial ontology in the figure below, *Company Name* is an attribute of the *Company* semantic type. Modifiers are specialized attributes whose values can vary from one context to another. For example *Format Modifier* is a modifier of *Company Name*. In other words, until we are given a context that defines the *Format*

Modifier, it is not immediately obvious from the ontology what format *Company Name* is in.

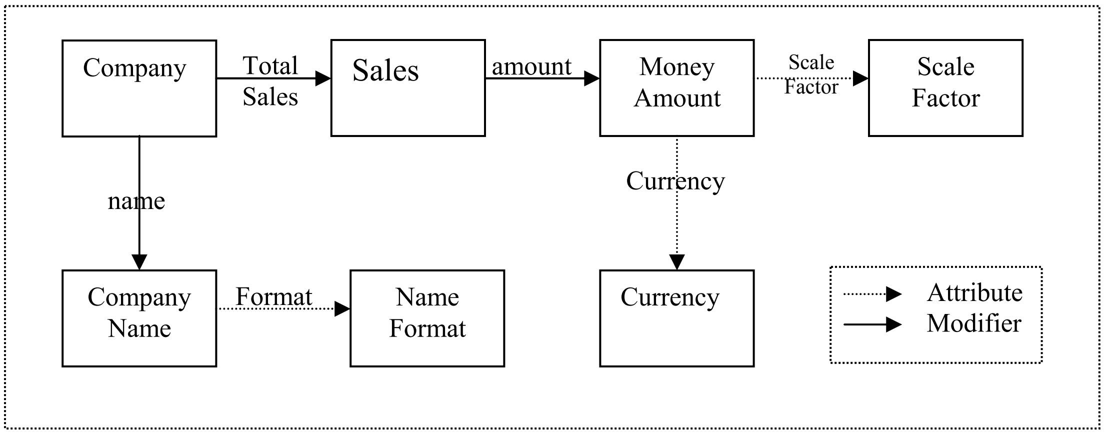


Figure 2.1 An excerpt of a financial ontology

2.1.2 Context Definitions

COIN ontologies tackle contextual heterogeneity by using the concept of modifiers. For instance, if *moneyAmount* is a semantic type, which has *currency* and *scaleFactor,* depending on the context, we define these two attributes as modifiers of *moneyAmount*. And as such, *currency* and *scaleFactor* will be assigned values for every context. Hence, an American context may define *currency* to be “US Dollars” and *scaleFactor* to be 1, whereas an Italian context might choose to assign “Lire” and 1,000, 000 to the two values. In a more general sense, a context definition is simply an assignment of particular values to the modifiers in the system.

2.1.3 Data Sources

The main purpose of GCMS is performing context mediation between different data sources. Therefore, any GCMS application needs data sources, and each of these data sources needs to subscribe to the application’s ontology. This relationship between the sources and the ontology is formally expressed through what are known as *elevations*.

2.1.4 Elevations

So far, we have described how ontology and contexts are defined in the COIN framework. Although GCMS uses only an abstract model for context mediation, the whole purpose of this system is querying physical data sources. Therefore, we need a way of tying up the physical model or schema with the more abstract representation or the ontology. Elevations do just that. An elevation *elevates* an attribute in a physical relation to an ontology object, turning relation contents from raw data to context-aware information.

Let's once again consider the excerpt of the financial ontology from Figure 2.1. Based on this figure, a Company object has two attributes: *name* represented by a *CompanyName* object and *total sales* represented by a *Sales* object. A *CompanyName* object, in turn, has a modifier called *Format*, which is represented by a *NameFormat* object. A *Sales* object has an attribute *amount*, which is of object type *moneyAmount* as defined in the earlier example. So this exceprt consists of three modifiers: Format, Currency and scaleFactor.

Now let us consider a data source, which can be elevated to this ontology.

Relation: DStreamAF			
Name	**As_of_date**	**Total_sales**	**Earned_for_ordinary**
DAIMLER-BENZ	05-01-96	103548992	-5674000
NNT	05-01-95	7037243392	76278000

Table 2.1 DStreamAF Table

Table 2.1 shows the excerpt of a relation that contains financial data on companies. The table contains four attributes, but for the purpose of this example, we only need to concern ourselves with Name and Total_sales. The Name column and Total_sales columns contain raw data at the moment but we can make this data context-aware by elevating these two columns to the appropriate semantic types in the ontology from Figure 2.1. Subsequently, Name can be elevated to the *Company* attribute *name* and Total_sales can be elevated to the attribute *TotalSales*. By elevating these two columns to these two semantic types, we allow the characterization of the data in these two columns using any of the contextual elements defined for these types.

We have already specified that both of these semantic types have associated modifiers. Namely, name has a *Format* modifier and although *Sales* does not have its own modifier it is affected by the modifiers of its attribute *amount*. Amounts modifiers *scaleFactor* and *Currency* have direct baring on the contextual significance of *netSales*; consequently, these modifiers are imposed on any column that is elevated to *netSales*.

The data in the two columns is now tied into the abstract model and mediation on queries to this physical source can now be performed based on these elevations. Once the attributes of a relation have been elevated, making the source data context-aware is just a simple matter of defining a context and assigning it to the source. In addition, users will be a able to use these contexts as a receiver's context for defining the contextual environment from which they will be issuing queries.

To summarize, elevations are the physical to abstract mappings that make it possible for the abduction engine [7] to perform context mediation on physical data sources. Before a data source can participate in a mediation process, it needs to be elevated to the ontology of its parent application.

2.1.5 Conversion Functions

Another key component of GCMS that can be defined through the front end tool is the conversion function for a modifier. Since modifiers can take on different values, they also need functions that specify how to convert data between contexts with two different values for the same modifier. For instance, in the case of the *scaleFactor* modifier, to make data in the American context (*scaleFactor* is 1) coherent with similar data in the Italian context (*scaleFactor* is 1,000,000) we will need to multiply or divide by 1,000,000 depending on the order of conversion. A conversion function for the *scaleFactor* modifier will formally express this relationship between two different modifier values and provide the rules for converting relevant data from one context to another. In general, conversion functions are defined per modifier and they are used by the mediator to convert data between contexts.

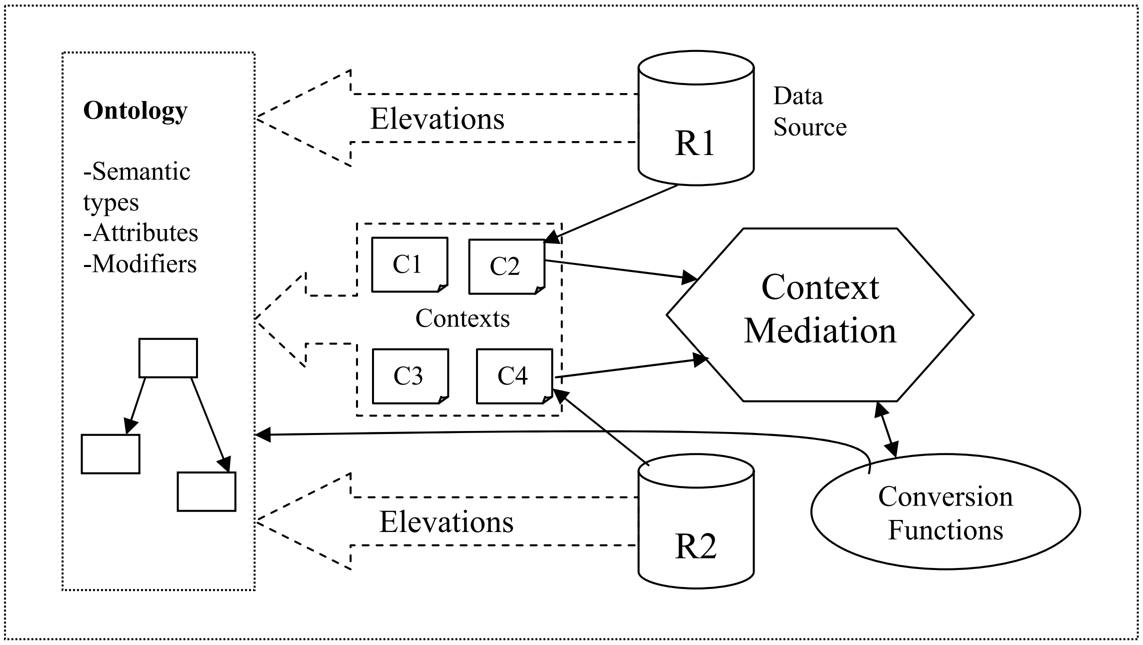


Figure 2.2 The Elements of a GCMS Application

Figure 2.2 shows all the key components of an application and how they are tied to each other. The application's ontology will have a representation that complies with the specifications of the COIN framework [7]. In order to make ontology management easier, GCMS allows users to build and edit ontologies in their graphical form. This function is provided through a ontology with a user-friendly graphical interface. After the user builds an ontology using this interface, the tool stores the new model in a central registry and converts this graphic representation into an internal representation that can be used in mediation. This tool also allows users to edit existing ontologies by re-deriving the graphical representation from model data retrieved from the registry.

The ontology editor is part of a larger front-end application management interface. Another important piece of this front-end interface is the meta-data management tool. The meta-data management mechanism is used to manage the contexts, data sources, conversion functions and elevations of applications. Given this mechanism and a working ontology, a user can build a proper application with all the contextual elements necessary for mediation. For instance, the user can define conversion functions, define a context and a source, do the proper elevations to the source schema and assign the context to the source, and we now have a simple “ready for mediation” application complying with COIN specifications.

The components we have discussed so far are all key to the GCMS implementation, and the front end interface allows us to create and modify these different components. Going back to the notion of applications, the purpose of this front end tool can be summarized as the general maintenance of GCMS applications. Once the application has been built and its key components have all been defined, it is ready for context mediation.

2.2 Abduction Engine

When a query is dispatched to the system, it goes through several stages before it finally returns the results. Arguably the most important stage is mediation and this process is handled by the abduction engine. Before the abduction engine can start mediation, first the query needs to be converted into Datalog. The original query is expressed in SQL, and an SQL to Datalog parser is used to do the necessary conversion. After converting to Datalog, the query is now in Naive Datalog form, or it still does not take into account the receiver’s context or any source contexts.

In the very next stage, this naive Datalog query is upgraded to context sensitive Datalog by including in it the receiver’s context and the source contexts. However, it should be noted that the query is still unaware of possible conflicts between these contexts.

The next two stages are conflict detection and mediation. In these two stage, the abduction engine determines any conflicts between the contexts and formulates a mediated query by applying the necessary conversion functions to resolve these conflicts. This query is still in Datalog form but it is now contextually accurate. In the very next stage, another parser is used to convert this Datalog query back into SQL. This query is now passed to the query planner/executioner [7].

2.3 Query Planner/Executioner

The query planner takes in the mediated SQL query and forms a plan for executing it. The data sources in the query might include databases, web pages and local sources, therefore, the query planner needs to determine the optimal order in which these sources are accessed and the results are put together. This plan is passed to the optimizer for further optimization before it is finally executed to output a result.

Figure 2.3 shows an overview of the GCMS architecture.

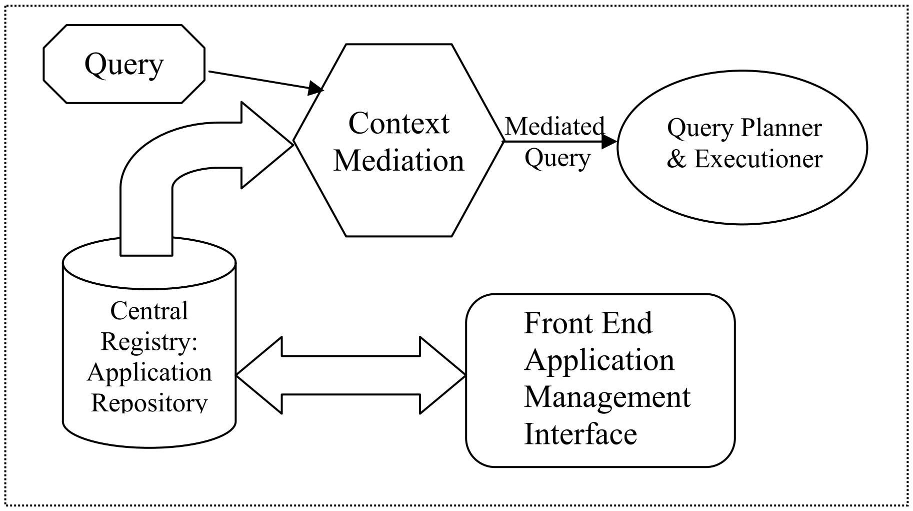

Query
Context Mediation
Mediated Query
Query Planner & Executioner
Central Registry: Application Repository
Front End Application Management Interface

Figure 2.3 A bigger picture of GCMS

Now that the reader has a fairly detailed understanding of GCMS components and processes, we will go into the main purpose of this thesis.

3 System Design

The next few sections contain a detailed description of the design of the system, and we will briefly explain the purposes of the key components of the system. Context mediation on federated queries is performed in three major stages:

1. Query Planning
2. Receiver Context Generation
3. Merging

3.1 Query Planner

At this point, we assume that the reader has some familiarity with the current implementation of GCMS. The purpose of the query planner is to take a federated query and break it down into single application queries that can be handled by GCMS. In other words, if a query contains sources belonging to more than one application, the planner will break the query down per application by grouping same application sources together.

3.1.1 SQL Optimization

However, before the planner even attempts to break down the query it does some optimizations that will speed up the merging process. Let us consider the following query:

```
select WorldcAF.TOTAL_ASSETS, DStreamAF.EARNED_FOR_ORDINARY,
        from WorldcAF, DStreamAF
        where WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'
        and DStreamAF.AS_OF_DATE = '01/05/94'
        and WorldcAF.COMPANY_NAME = DStreamAF.NAME
```

Here, we assume that WorldcAF and DstreamAF are relations belonging to two different applications, WorldScope and DataStream respectively. Although the type of standard SQL optimization that we will be making is already implemented in the GCMS Executioner, we still need to apply it to this federated query because the executioner is able to optimize only the single application queries that are derived from the federated query.

In this case, we can immediately see that the join condition on DStreamAF.NAME and WorldcAF.COMPANY_NAME is redundant since WorldcAF.COMPANY_NAME is already set equal to a constant ‘DAIMLER-BENZ AG’. Hence we can improve this query by replacing the join with another constant to attribute comparison.

```
select WorldcAF.TOTAL_ASSETS, DStreamAF.EARNED_FOR_ORDINARY,
        from WorldcAF, DStreamAF
        where WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'
        and DStreamAF.AS_OF_DATE = '01/05/94'
        and DStreamAF.NAME = 'DAIMLER-BENZ AG'
```

After the query planner breaks down this federated query, we have the following two subqueries for WorldScope and DataStream.

```
DataStream:   select DStreamAF.EARNED_FOR_ORDINARY,  DstreamAF.NAME
                      from DStreamAF
                      where DStreamAF.AS_OF_DATE = '01/05/94'
                      and DstreamAF.NAME = 'DAIMLER-BENZ AG'


WorldScope:   select WorldcAF.TOTAL_ASSETS, WorldcAF.COMPANY_NAME
                      from WorldcAF
                      where WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'
```

Note that DstreamAF.NAME and WorldcAF.COMPANY_NAME have to be selected in the views even if they were not selected in the federated query because the final merging is performed on those two columns. The joining condition for these two sub-queries will be

```
              Select DataStream.Dstream.EARNED_FOR_ORDINARY,
                      WorldScope.WorldcAF.TOTAL_ASSETS
                      From DataStream, WorldScope
                      Where DataStream.DstreamAF.NAME =
                      WorldScope.WorldcAF.COMPANY_NAME
```

Here, DataStream and WorldScope pertain to the views containing the mediated results returned from the two applications. Since we have imposed strong conditions on COMPANY_NAME and NAME in both sub-queries, the join condition that is performed on these two attributes in the merging stage is almost trivial. In fact, in this case, since we have already specified the company name, we expect to get just one row back from each subquery.

However, consider what would have happened if we hadn't made the this optimization. In this case, after breaking down the federated queries, we would have

```
DataStream:   select DStreamAF.EARNED_FOR_ORDINARY,  DstreamAF.NAME
                      from DStreamAF
                      where DStreamAF.AS_OF_DATE = '01/05/94'


WorldScope:   select WorldcAF.TOTAL_ASSETS, WorldcAF.COMPANY_NAME
                      from WorldcAF
                      where WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'
```

The join condition will still be the same, but we now have to do more work during merging because we have not done any pre-mediation optimization on the joined attributes from the two applications. In other words, the DataStream view now contains even rows for which NAME is not 'DAIMLER-BENZ AG'; Hence, we will have to enforce this condition during the merging stage, whereas in the optimized case, we were able to enforce it at the level of the DataStream subquery.

3.1.2 Subquery Generation

Breaking down a query into subqueries involves two steps. The first one is grouping all the sources in the query by application. This can be determined by consulting the registry which contains schema information on all applications. The next step is picking out the relevant conditions for each application and applying the appropriate propositional logic to them. In other words, we want to recover the right disjunctive and conjunctive connectives for each condition without altering the query logic.

Let's consider a slightly more elaborate example of a DataStream-WorldScope federated query after some optimization:

```
select WorldcAF.TOTAL_ASSETS, DStreamAF.EARNED_FOR_ORDINARY,
        from WorldcAF, DStreamAF
        where WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'
        or DStreamAF.AS_OF_DATE = '01/05/94'
        and DStreamAF.NAME = 'DAIMLER-BENZ AG'
        or WorldcAF.sales > 500000
        and WorldcAF.total_assets > 10000000
        and DstreamAF.earned_for_ordinary > 10000000
        and WorldcAF.COMPANY_NAME = DstreamAF.NAME
```

Our goal is to break this query down into two separate queries without altering the propositional logic it would have applied if all the sources had been from the same application. The safest approach here would be to use the cover-up method and treat all the conditions that we don't want in a subquery conditions as true logic statements. In the case of the example, this will translate into the following:

DataStream subquery:

select DStreamAF.EARNED_FOR_ORDINARY from DStreamAF
where ~~WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'~~ true
or DStreamAF.AS_OF_DATE = '01/05/94'
and DStreamAF.NAME = 'DAIMLER-BENZ AG'
or ~~WorldcAF.sales > 500000~~ true
and ~~WorldcAF.total_assets > 10000000~~ true
and DstreamAF.earned_for_ordinary > 10000000
and ~~WorldcAF.COMPANY_NAME = DstreamAF.NAME~~ true

```
select DStreamAF.EARNED_FOR_ORDINARY from DStreamAF
        where true
        or DStreamAF.AS_OF_DATE = '01/05/94'
        and DStreamAF.NAME = 'DAIMLER-BENZ AG'
        or true
        and true
        and DstreamAF.earned_for_ordinary > 10000000
        and true
```

Merging condition:

```
select WorldcAF.TOTAL_ASSETS, DStreamAF.EARNED_FOR_ORDINARY,
        from WorldcAF, DStreamAF
```

```
          where ~~WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'~~ true
          or ~~DStreamAF.AS_OF_DATE = '01/05/94'~~ true
          and ~~DStreamAF.NAME = 'DAIMLER-BENZ AG'~~ true
          or ~~WorldcAF.sales > 500000~~ true
          and ~~WorldcAF.total_assets > 10000000~~ true
          and ~~DstreamAF.earned_for_ordinary > 10000000~~ true
          and WorldcAF.COMPANY_NAME = DstreamAF.NAME

select WorldcAF.TOTAL_ASSETS, DStreamAF.EARNED_FOR_ORDINARY,
          from WorldcAF, DStreamAF
          where true
          or true
          and true
          or true
          and true
          and true
          and WorldcAF.COMPANY_NAME = DstreamAF.NAME
```

We have left out the WorldScope subquery because it uses the same exact approach as the DataStream subquery. The next step is to apply propositional logic to these simplified queries by starting with the conjunctions, followed by the disjunctions, applied from left to right in the order they appear.

The simplification rule we will be using is as follows. If we have two true statements connected through disjunction or conjunction we will just eliminate the first truth statement and the connective following it. If we have a true statement joined with a relational condition, we will remove the true statement and the connective between them.

After all the conjunctions have been eliminated in the DataStream subquery, we have

```
select DStreamAF.EARNED_FOR_ORDINARY from DStreamAF
          where true
          or DStreamAF.AS_OF_DATE = '01/05/94'
          and DStreamAF.NAME = 'DAIMLER-BENZ AG'
          or DstreamAF.earned_for_ordinary > 10000000
```

And after all the disjunctions have been removed, we have

```
select DStreamAF.EARNED_FOR_ORDINARY from DStreamAF
          where DStreamAF.AS_OF_DATE = '01/05/94'
          and DStreamAF.NAME = 'DAIMLER-BENZ AG'
          or DstreamAF.earned_for_ordinary > 10000000
```

Similarly, for the merging condition we'll have the following simplified query after eliminating all the true statements.

```
select WorldcAF.TOTAL_ASSETS, DStreamAF.EARNED_FOR_ORDINARY,
          from WorldcAF, DStreamAF
          where WorldcAF.COMPANY_NAME = DstreamAF.NAME
```

Since merging will be done on the outputs from the individual subqueries, we always need to make sure that the attributes in the join conditions are always selected in the subqueries. For instance, in this example, WorldcAF.COMPANY_NAME and DstreamAF.NAME are not selected in the federated query, and hence neither appears in the generated subqueries. Therefore, we have to inspect the merging condition and make sure that all the joined attributes are selected in the subqueries.

So after enforcing this rule we get the following two subqueries

DataStream:
```
    select DStreamAF.EARNED_FOR_ORDINARY, DstreamAF.NAME
        from DStreamAF
        where DStreamAF.AS_OF_DATE = '01/05/94'
        and DStreamAF.NAME = 'DAIMLER-BENZ AG'
        or DstreamAF.earned_for_ordinary > 10000000
```
WorldScope:
```
    select WorldcAF.TOTAL_ASSETS, WorldcAF.COMPANY_NAME
        from WorldcAF
        where WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'
        or WorldcAF.sales > 500000
        and WorldcAF.total_assets > 10000000
```

3.2 Context Generator

Now we have single application queries that can be handled by the current GCMS system. However, mediation requires a receiver context, and we need to send equivalent receiver contexts with each GCMS subquery in order to make merging possible. In fact, so far we have assumed that the two receiver contexts for the DataStream and WorldScope mediation processes are equivalent because if that were not the case, we would not be able to perform simple SQL type joins on final the views returned from these mediations.

However, we ideally do not want to impose many constraints on the users by requiring them to define equivalent contexts for every application that is involved in the process. Since we are doing a join that spans multiple applications, it is safe to assume that there is some semantic equivalence between objects in the involved applications. In the example we just gave, we know that WorldcAF.COMPANY_NAME and DstreamAF.NAME elevate to at least semantically related (if not semantically equivalent) types in the two applications.

Hence we can also infer that there is a lot of semantic equivalence in the contextual definitions that are relevant to these semantic types. Ideally, we do not want to make the user provide the definitions for both applications. Instead, we want to take just one of the equivalent definitions and derive the rest using alignments that have been made prior to query execution.

For our implementation, we use an example of an ontology alignment that seeks to resolve only synonym mismatches to demonstrate how cross-application alignments can be used to generate equivalent receiver contexts for all involved application and how these contexts can be used with the other components of this system to make cross-application context mediation possible. A more powerful mechanism for aligning ontologies will be presented in the last chapter of this thesis.

3.2.1 Synonym Matching

Synonym Matching resolves only one of several semantic conflicts that arise in ontology merging. Hence it is feasible only in cases where the concerned ontologies are very similar in structure. We are not proposing this as a lasting solution to the problem of ontology heterogeneity but rather as a simple back end partial solution to let us have a complete end-to-end demonstration of the other components of the merging system.

Alignments are made between any two applications and they are transitive between applications. For instance, let us suppose we have two applications with ontology A and B that are already aligned. Now if a third application with ontology C is aligned with A, by the transitive property, all the alignments between A and B are passed on to A and C. Hence, all three applications will be aligned with one another.

Since we are considering only synonym matching in this example, we will simply start by establishing semantic equivalence between like semantic types in the ontologies and work our way down to matching modifiers and modifier values. In other words, once we have aligned two semantic types, we will proceed with the assumption that the modifiers from one semantic type can only be aligned with modifiers from the other semantic type. Once the modifiers have been aligned, we also need to do mappings between equivalent modifier values because all ontologies do not use the same terminology.

Now let us assume that the ontology excerpt we had in Figure 2.1 was from a DataStream world. Let the following figure represent an excerpt from an equivalent ontology in the WorldScope world.

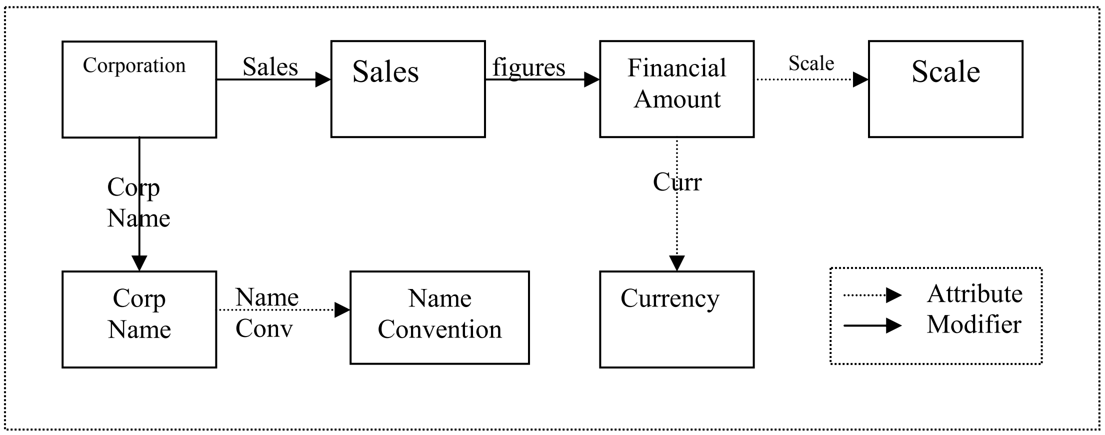


Figure 3.1 An excerpt of the WorldScope financial ontology

Since these two ontologies are structurally similar, we can do simple synonym alignments between them. The following figure shows how we can align sections of the two ontologies by drawing equivalence between semantic types, attributes, modifiers and modifier values in the two ontologies.

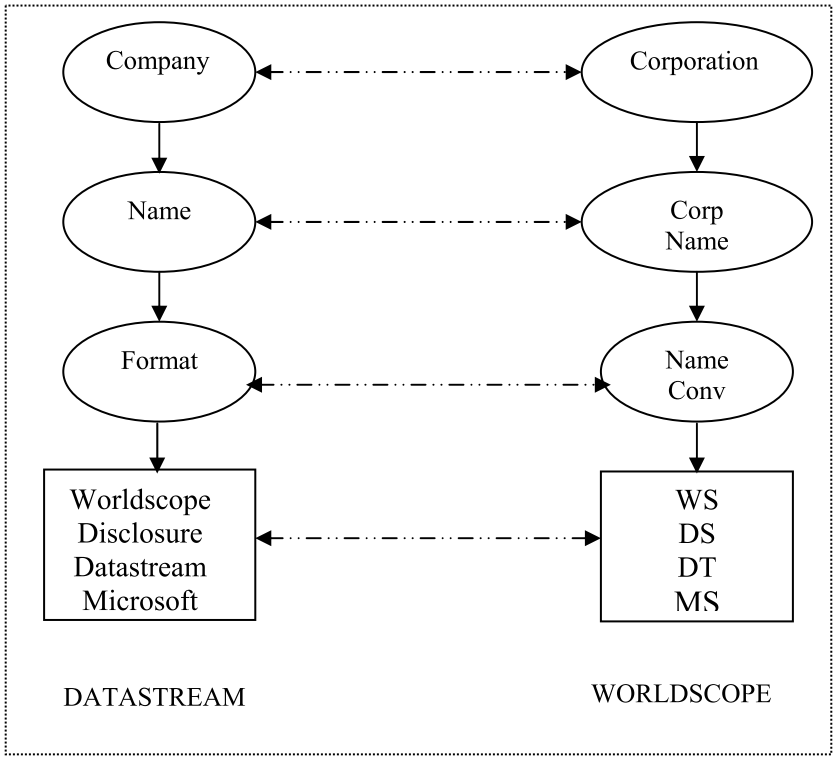


Figure 3.2 Synonym alignments between DataStream and WorldScope ontologies

In this example, the semantic type *Company* in DataStream is aligned with the *Corporation* from WorldScope. By establishing this semantic equivalence, we are affirming that the two semantic types are interchangeable, differences in their attributes and their modifiers notwithstanding. Similarly, we can align *Name* and *Corp Name*.
Next we want to align the modifiers from these two attributes. Once we have aligned the modifiers, we need to do equivalence mappings between every possible values of the aligned modifiers because the terminology is likely to be different even for the modifier values.

Now we have the alignments that are required to define receiver contexts that span multiple applications. First of all, we will define the receiver context on a given number of applications. A single application context in GCMS simply contains modifier definitions for the modifiers in that application. In this case, the receiver context will be able to include modifiers from more than one application. However, any context cannot contain multiple definitions for the same modifier. Therefore, this puts a constraint on all semantically equivalent modifiers because they cannot have conflicting definitions.

We have two options here: the first one is to make the user specify equivalent values for all semantically equivalent modifiers. This approach will require a considerable amount of work during context definitions in determining the right modifier values for each application based on the alignments we have. In the alignment example we just discussed, this would be done by setting both *format* and *name conv* to equivalent modifier values in their respective applications.

The second, and perhaps more feasible approach is to disallow the definition of multiple semantically equivalent modifiers. In other words, the receiver context will contain only one modifier value definition for every class of semantically equivalent modifiers. In this case, we would pick only one of *format* and *name conv* and assign it a value in the context. This will be the approach we will be using in this implementation.

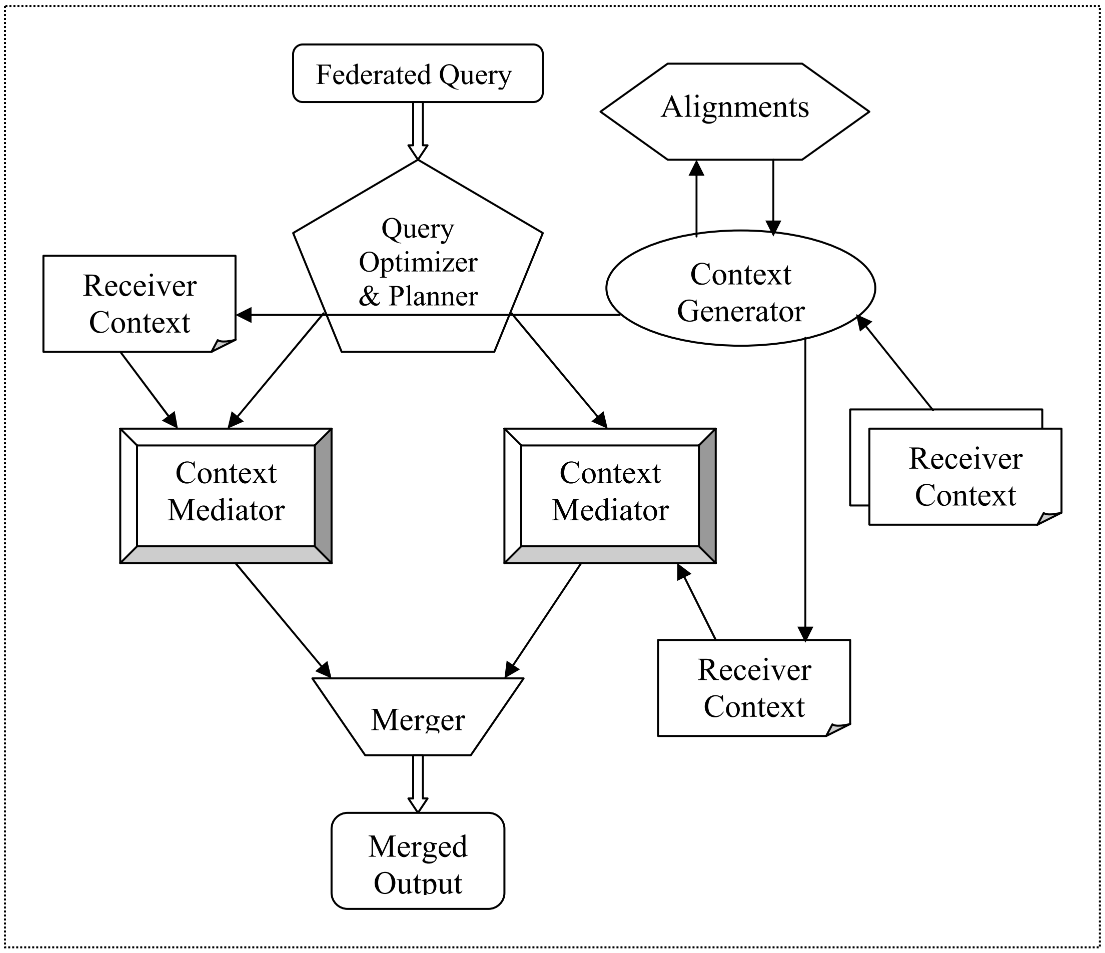


Figure 3.3 Design Architecture

The purpose of the context generator is to take these heterogeneous modifier definitions and derive the corresponding modifier definitions for all the semantically equivalent modifiers in all concerned applications. In other words, given an application, if the receiver context contains modifier definitions for all unaligned modifiers and definitions of semantically equivalent modifiers for all aligned modifiers, the context generator should be able to generate the equivalent homogeneous receiver context for that application.

Since the receiver context is typically built to span more than one application, the number of homogeneous receiver contexts generated will often be more than one. Equipped with these GCMS compatible receiver contexts, we are now ready to go ahead and do routine context mediation on the subqueries that were produced by the query planner. However, in order to do that, we first need to parametrize the current GCMS system to be able to handle multiple mediation processes concurrently.

3.3 Parametrization of GCMS

Context Medation on GCMS is done using an abduction engine whose functions are invoked through an interface provided by the executioner. The abduction engine needs to load all the necessary application files before it can do mediation. These files include internal representations for the application's ontology, contexts, sources, conversion functions and elevations. These internal reps are loaded into a module and this module is kept intact once the abduction engine is initialized.

However, GCMS can currently support only one module at a time. We could load one application after another sequentially into this module and do mediation in series but a more feasible approach is to parametrize the abduction engine and the execution engine to support multiple modules. This allows us to have more than one application loaded at the same time so we can run multiple context mediation processes in parallel. This approach is also time efficient because it does not require the re-initialization of the Eclipse engine every time a new application is loaded into a module [7].

At this point, we have a system that is completely parametrized so we can start with a federated query and get outputs from the individual mediation processes for all concerned applications. The last stage involves taking these output and merging them by the imposing the merging conditions that were derived during the query planning stage.

3.4 Merging

Let us now revisit the DataStream-WorldScope example we used to demonstrate the query planning stage. After breaking the federated query into two subqueries we have the following merging condition to complement them:

```
select WorldcAF.TOTAL_ASSETS, DStreamAF.EARNED_FOR_ORDINARY,
        from WorldcAF, DStreamAF
```

```
          where WorldcAF.COMPANY_NAME = DstreamAF.NAME
```

This merging condition is incomplete right now because it references relations instead of applications. For instance, we could have several relations called WorldcAF and DstreamAF. Therefore, we need to make it clear that this merging condition applies to only the views of these tables that were generated during query planning. To simplify matters, let us refer to the views containing the outputs from query planning as DS (for DataStream) and WS (for WorldScope).

Now we can specify a merging condition that is particular to these views.

```
select WS.WorldcAF.TOTAL_ASSETS, DS.DStreamAF.EARNED_FOR_ORDINARY,
          from WS, DS
          where WS.WorldcAF.COMPANY_NAME = DS.DstreamAF.NAME
```

Executing this query will finally give us a mediated output for the original federated query bringing us to the completion of the mediation and merging process.

4 Implementation

Almost all the implementation in this system is done exclusively in Java, including the propositional logic involved in the query planner. In this chapter, we will present the implementation of the following components

1. Alignment Tool
2. Context Generator
3. Query Builder
4. Query Planner
5. GCMS Parametrization Utility
6. Merger

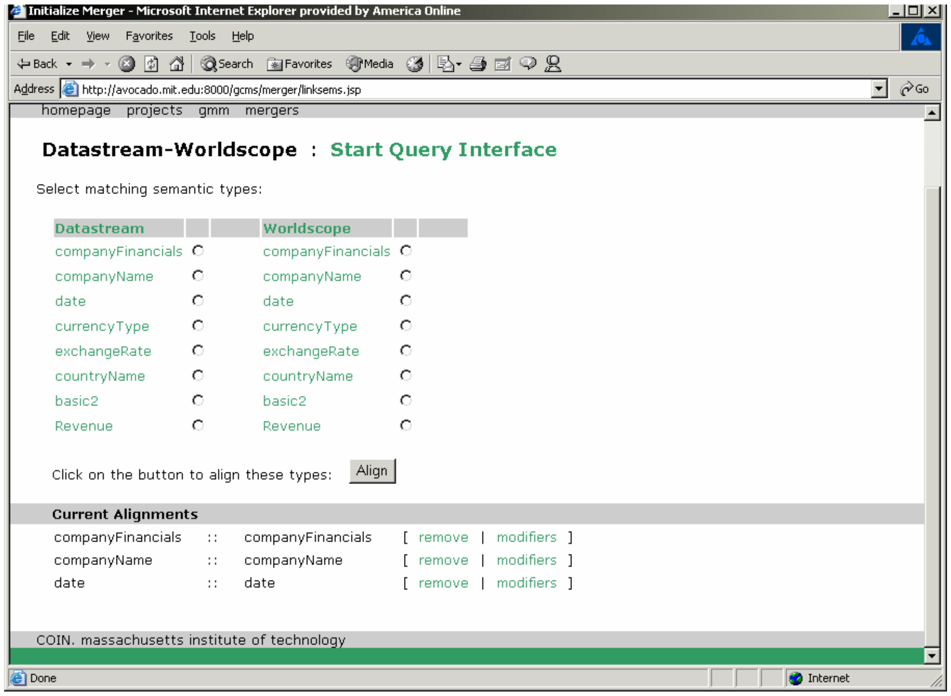
4.1 Alignment Tool

Once again, the purpose of the alignment tool is to establish semantic equivalence between objects in ontologies from different applications. The tool allows users to make alignments between any two applications and it relies on the registry to store these equivalence mappings. In order to make this possible, we have implemented the following data model

```
SQL> desc merger
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------

 MERGER_ID                                          NUMBER(38)
 NAME                                               VARCHAR2(40)
 APP1                                               NUMBER(38)
 APP2                                               NUMBER(38)

SQL> desc semtype_map
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------

 MAPPING_ID                                         NUMBER(38)
 MERGER_ID                                          NUMBER(38)
 SEM1                                               NUMBER(38)
 SEM2                                               NUMBER(38)

SQL> desc modifier_map
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------

 MAPPING_ID                                         NUMBER(38)
 SEMTYPE_MAP                                        NUMBER(38)
 MOD1                                               NUMBER(38)
 MOD2                                               NUMBER(38)

SQL> desc mod_val_map
 Name                                      Null?    Type
 ----------------------------------------- -------- ------------------
```

MAPPING_ID	NUMBER(38)
MOD_MAPPING	NUMBER(38)
VALUE1	VARCHAR2(100)
VALUE2	VARCHAR2(100)

These mapping relations reference each other in the same order the parts of an ontology reference each other. In other words, mappings between two semantic types (semtype_map) belong to a mapping between two ontologies (merger) and mappings between modifiers (modifier_map) belong to a mapping between semantic types (semtype_map).



Initialize Merger - Microsoft Internet Explorer provided by America Online
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://avocado.mit.edu:8000/gcms/merger/linksems.jsp
Go
homepage projects gmm mergers
Align
Done
Internet

Datastream-Worldscope : Start Query Interface

Select matching semantic types:

Datastream		Worldscope	
companyFinancials	○	companyFinancials	○
companyName	○	companyName	○
date	○	date	○
currencyType	○	currencyType	○
exchangeRate	○	exchangeRate	○
countryName	○	countryName	○
basic2	○	basic2	○
Revenue	○	Revenue	○

Click on the button to align these types:

Current Alignments

companyFinancials	::	companyFinancials	[remove | modifiers]
companyName	::	companyName	[remove | modifiers]
date	::	date	[remove | modifiers]

COIN. massachusetts institute of technology

Figure 4.1 A screenshot of the alignment tool

The alignment interface uses JSP pages and follows the same central registry used by GCMS. Users make the alignments in a sequential manner starting from applications and working their way down to modifier values. In other words, alignments on modifiers values cannot be made until the modifiers, their parent semantic types and their parent applications have all been aligned.

Also this implementation does not support one-to-many or many-to-many alignments. The reason for this restriction will become more apparent when we are discussing the implementation of receiver contexts. For modifier values that are identical for two aligned modifiers, the user does not need to make the alignments because the context generator always assumes that missing alignments for modifier values constitute semantic equivalence. Once all the proper alignments have been made, the user can build a receiver context spanning multiple applications.

4.2 Context Generator

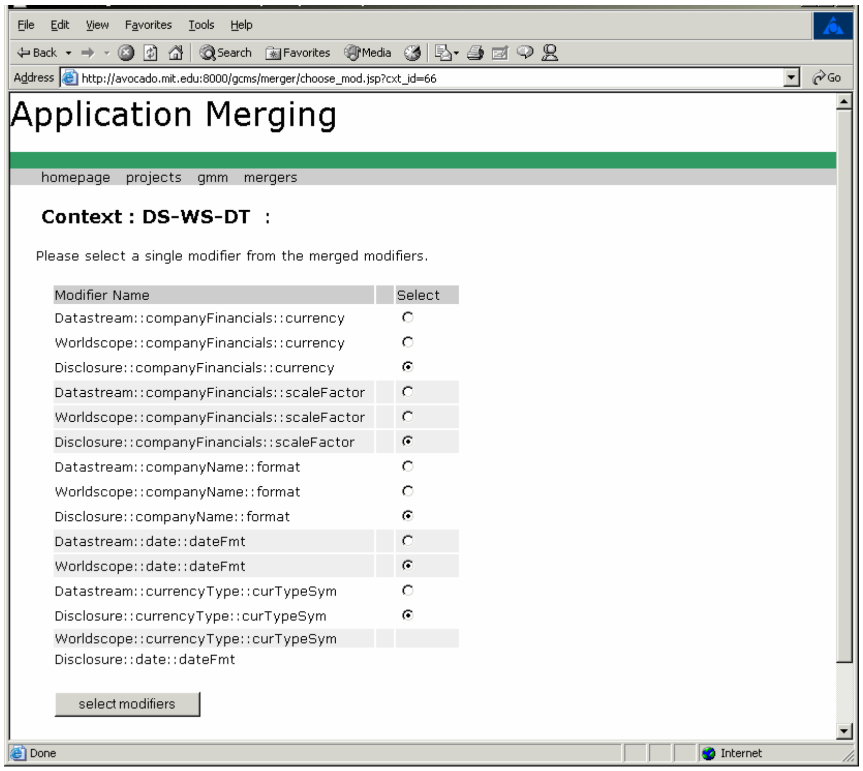
Just like the alignment tool, the context generator prompts users to specify which application to use in building this context. We can classify all the possible scenarios into three groups:

- The federated query will involve certain applications and the receiver context will contain modifiers from more than one of these applications.
- The federated query will involve certain applications and the receiver context will contain modifiers from just one of these applications
- The receiver context will contain modifiers that don't belong to any of the applications involved in the federated query.

In all of these three cases, we require to have the proper alignments between all applications involved in both the federated query and the receiver context.

Although alignments are between pairs of applications, the context mediator is able to transitively deduce alignments that are not explicitly specified by the user. For instance, if the user aligns DataStream with both WorldScope and Disclosure, the context mediator is able to derive the alignments between WorldScope and Disclosure with no further action from the user.

Once the user has specified, which applications to use in this receiver context, the context generator will return a list of all the modifiers from all selected applications. The generator will also look up the alignments between these applications and force the user to choose only one modifier for cases where it finds modifiers that are already aligned. This goes back to the discussion of conflicting modifier definitions in the design section. Therefore, the receiver context template will essentially contain all unaligned modifiers from all applications and one modifier representing every group of aligned modifiers.


File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://avocado.mit.edu:8000/gcms/merger/choose_mod.jsp?cxt_id=66
Go
homepage projects gmm mergers
select modifiers
Done
Internet

Application Merging

Context : DS-WS-DT :

Please select a single modifier from the merged modifiers.

Modifier Name	Select
Datastream::companyFinancials::currency	○
Worldscope::companyFinancials::currency	○
Disclosure::companyFinancials::currency	●
Datastream::companyFinancials::scaleFactor	○
Worldscope::companyFinancials::scaleFactor	○
Disclosure::companyFinancials::scaleFactor	●
Datastream::companyName::format	○
Worldscope::companyName::format	○
Disclosure::companyName::format	●
Datastream::date::dateFmt	○
Worldscope::date::dateFmt	●
Datastream::currencyType::curTypeSym	○
Disclosure::currencyType::curTypeSym	●
Worldscope::currencyType::curTypeSym	
Disclosure::date::dateFmt	

Figure 4.2 A screenshot of the modifier template builder

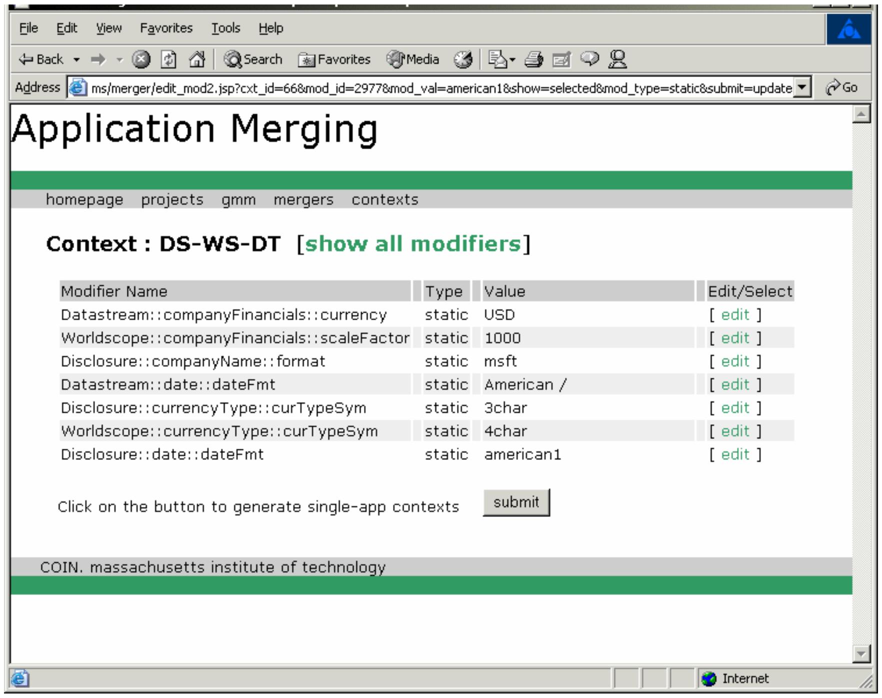
The user chooses which aligned modifiers to use and the template for the receiver context will be displayed showing the selected modifiers. However, users can always deselect and select modifiers and change the template. With the help of a DBA, the user can now specify the context definitions for all the modifiers.

At this point, we have a receiver context, which is compatible with all the applications involved assuming that all the modifier definitions are consistent with the alignments. During mediation, the context generator is invoked by the query planner to build single application contexts which are identical to a given receiver context that meets all the conditions we have discussed here. We are now ready to do mediation on federated queries involving any applications whose modifiers are in the receiver context or aligned with the modifiers in the receiver context.

Receiver contexts and their definitions are stored in the registry using the following data model:

```
SQL> desc merged_context
 Name                                      Null?    Type
 ----------------------------------------- -------- -------------------

 CONTEXT_ID                                          NUMBER
 NAME                                                VARCHAR2(30)
 APPS                                                VARCHAR2(20)
```

```
SQL> desc merged_cxt_values
 Name                                      Null?    Type
 ----------------------------------------- -------- -------------------

 CONTEXT_ID                                          NUMBER
 MODIFIER_ID                                        NUMBER
 MODIFIER_VALUE                                     VARCHAR2(40)
 MODIFIER_TYPE                                      VARCHAR2(10)
 SELECTED                                           CHAR(1)
 MERGING                                            VARCHAR2(15)
```


Address: ms/merger/edit_mod2.jsp?cxt_id=66&mod_id=2977&mod_val=american1&show=selected&mod_type=static&submit=update

Application Merging

homepage projects gmm mergers contexts

Context : DS-WS-DT [show all modifiers]

Modifier Name	Type	Value	Edit/Select
Datastream::companyFinancials::currency	static	USD	[edit]
Worldscope::companyFinancials::scaleFactor	static	1000	[edit]
Disclosure::companyName::format	static	msft	[edit]
Datastream::date::dateFmt	static	American /	[edit]
Disclosure::currencyType::curTypeSym	static	3char	[edit]
Worldscope::currencyType::curTypeSym	static	4char	[edit]
Disclosure::date::dateFmt	static	american1	[edit]

Click on the button to generate single-app contexts [submit]

COIN. massachusetts institute of technology

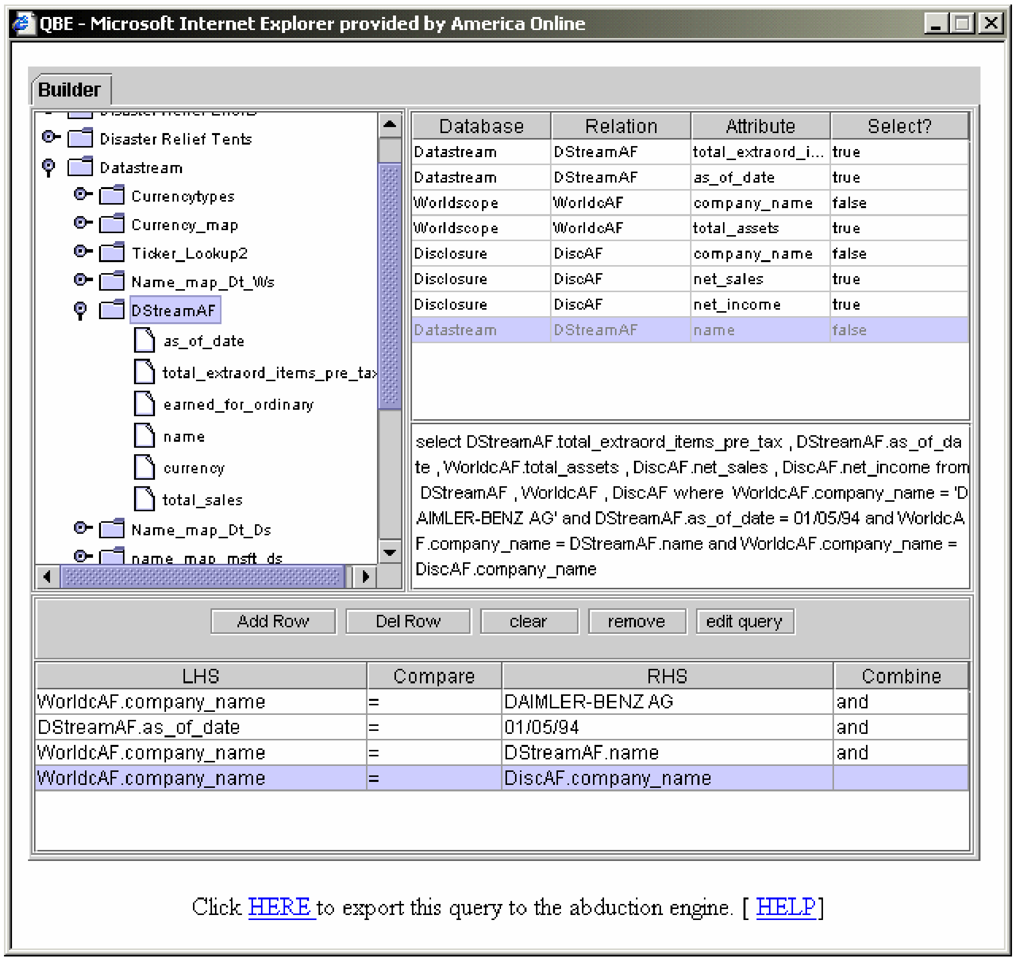
Figure 4.3 Sample Receiver Context

4.3 Query Builder

The query builder is an application that is designed to let the user build federated queries using the schema information stored in the central registry. The screenshot below shows how this tool is used to build fairly complicated queries. The tree like structure on the left hand side of the application imports the schema information on all GCMS applications and users are able to browse through all the relations in all applications and select the attributes they want to use in the query.

When an attribute is selected it appear on the top right hand side of the application. All selected attributes default to ‘TRUE’ for the SELECT? Property so if users do not want to select that column in the query, they have to change the value to ‘FALSE.’ While attributes are being selected and the SELECT? Property is being edited the federated query will be constantly updated and displayed on the frame right under the attribute frame.

Users can also build the 'WHERE' conditions using the lowest frame where they can specify what to compare on the left hand side (LHS) and the right hand side (RHS) and how to compare and combine these conditions with one another.



Database	Relation	Attribute	Select?
Datastream	DStreamAF	total_extraord_i...	true
Datastream	DStreamAF	as_of_date	true
Worldscope	WorldcAF	company_name	false
Worldscope	WorldcAF	total_assets	true
Disclosure	DiscAF	company_name	false
Disclosure	DiscAF	net_sales	true
Disclosure	DiscAF	net_income	true
Datastream	DStreamAF	name	false

```
select DStreamAF.total_extraord_items_pre_tax , DStreamAF.as_of_da
te , WorldcAF.total_assets , DiscAF.net_sales , DiscAF.net_income from
 DStreamAF , WorldcAF , DiscAF where  WorldcAF.company_name = 'D
AIMLER-BENZ AG' and DStreamAF.as_of_date = 01/05/94 and WorldcA
F.company_name = DStreamAF.name and WorldcAF.company_name =
DiscAF.company_name
```

LHS	Compare	RHS	Combine
WorldcAF.company_name	=	DAIMLER-BENZ AG	and
DStreamAF.as_of_date	=	01/05/94	and
WorldcAF.company_name	=	DStreamAF.name	and
WorldcAF.company_name	=	DiscAF.company_name	

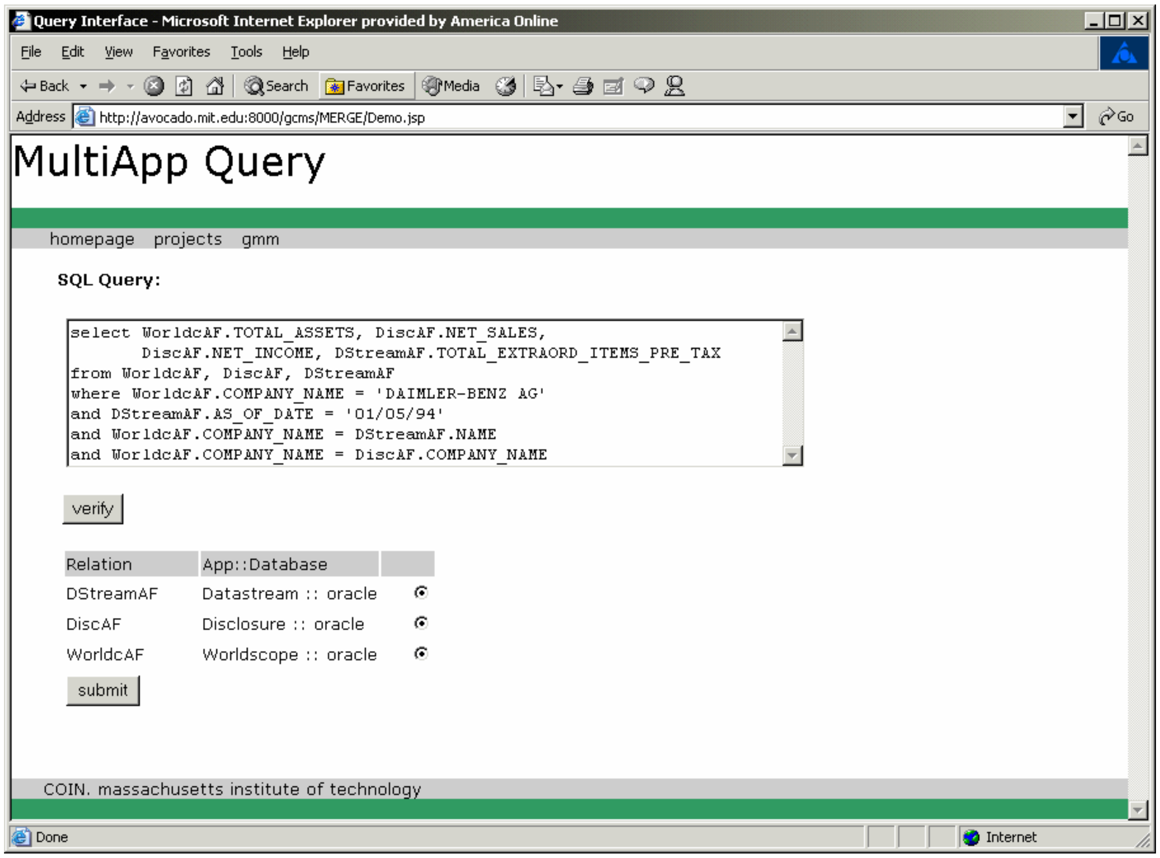
Click HERE to export this query to the abduction engine. [HELP]

Figure 4.4 A screenshot of the query builder

When the federated query is complete, the user can click on the link at the bottom of the screen to export it to the query planner interface.

4.4 Query Planner

When the query gets to the planner, the first thing the user needs to do is specify which application each relation is from. The planner looks up all the relations in the query and finds all the applications that contain relation by those names. In some cases, there will be more than one application per relation and the user will have to specify the right application.


Query Interface - Microsoft Internet Explorer provided by America Online
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://avocado.mit.edu:8000/gcms/MERGE/Demo.jsp
Go

MultiApp Query

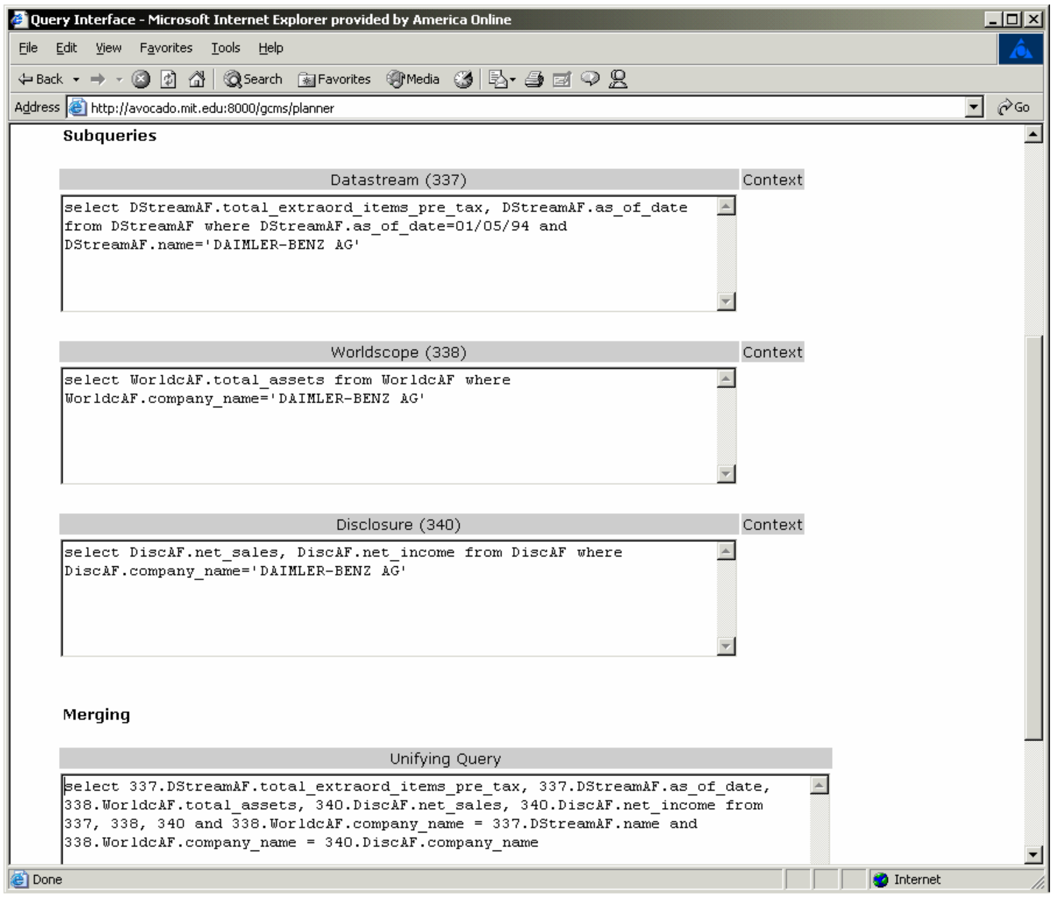
SQL Query:

```
select WorldcAF.TOTAL_ASSETS, DiscAF.NET_SALES,
       DiscAF.NET_INCOME, DStreamAF.TOTAL_EXTRAORD_ITEMS_PRE_TAX
from WorldcAF, DiscAF, DStreamAF
where WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'
and DStreamAF.AS_OF_DATE = '01/05/94'
and WorldcAF.COMPANY_NAME = DStreamAF.NAME
and WorldcAF.COMPANY_NAME = DiscAF.COMPANY_NAME
```

Relation	App::Database	
DStreamAF	Datastream :: oracle	●
DiscAF	Disclosure :: oracle	●
WorldcAF	Worldscope :: oracle	●

Figure 4.5 A screenshot of the query interface

After all the relations have been uniquely identified, the planner prompts the user to choose the receiver context for this query. The users are expected to know which receiver context is compatible with the applications they are trying to merge. Once the receiver context has been selected, the planner optimizes and produces a subquery for every application that was selected in identifying the relations.


Query Interface - Microsoft Internet Explorer provided by America Online
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://avocado.mit.edu:8000/gcms/planner
Go

Subqueries

Datastream (337) Context

```
select DStreamAF.total_extraord_items_pre_tax, DStreamAF.as_of_date
from DStreamAF where DStreamAF.as_of_date=01/05/94 and
DStreamAF.name='DAIMLER-BENZ AG'
```

Worldscope (338) Context

```
select WorldcAF.total_assets from WorldcAF where
WorldcAF.company_name='DAIMLER-BENZ AG'
```

Disclosure (340) Context

```
select DiscAF.net_sales, DiscAF.net_income from DiscAF where
DiscAF.company_name='DAIMLER-BENZ AG'
```

Merging

Unifying Query

```
select 337.DStreamAF.total_extraord_items_pre_tax, 337.DStreamAF.as_of_date,
338.WorldcAF.total_assets, 340.DiscAF.net_sales, 340.DiscAF.net_income from
337, 338, 340 and 338.WorldcAF.company_name = 337.DStreamAF.name and
338.WorldcAF.company_name = 340.DiscAF.company_name
```

Figure 4.6 A screenshot of the subquery generation page

The planner also invokes the context generator on the receiver context to produce a receiver context for every application. Once these single application contexts have been produced, the planner will invoke a GCMS parametrization utility that generates the files that are necessary for each mediation process.

4.5 Parametrization Utility

The purpose of this utility is to generate all the files that are needed by the abduction engine and the executioner for mediation. This includes the internal representations for contexts, ontologies, sources, elevations and conversion functions. These files will be generated for every application that needs a mediation process.

Instead of overwriting the same application files every time mediation is invoked, this utility retains the application files from every application and allows the execution of concurrent mediation processes. The utility also parametrizes the abduction engine by allowing it to start a new module every time a new application needs to be loaded up. Therefore, the multiple mediations necessary for merging are run in parallel, enhancing the scalability and the overall performance of the system.

4.6 Merger

The last component is the merger which is responsible for taking the output from each individual mediation process and merging them under the given merging conditions. The merger turns each output into a temporary view in the central registry and the final query is run on these views. The merging conditions are directly imported from the query planner and the merger simply returns the output meeting these conditions.

5 Motivational Example

As a driving example, let us consider a case where we try to merge three different applications. The three applications will be DataStream, WorldScope and Disclosure. The federated query be querying one relation from each application.

DataStream: DStreamAF
WorldScope: WorldcAF
Disclosure: DiscAF

Federated Query:

```
select WorldcAF.TOTAL_ASSETS, DiscAF.NET_SALES,
        DiscAF.NET_INCOME, DStreamAF.TOTAL_EXTRAORD_ITEMS_PRE_TAX,
        from WorldcAF, DiscAF, DStreamAF
        where WorldcAF.COMPANY_NAME = 'DAIMLER-BENZ AG'
        and DStreamAF.AS_OF_DATE = '01/05/94'
        and WorldcAF.COMPANY_NAME = DStreamAF.NAME
        and WorldcAF.COMPANY_NAME = DiscAF.COMPANY_NAME
```

After optimizing and breaking it down, we get the following three subqueries.

DataStream:

```
select DStreamAF.TOTAL_EXTRAORD_ITEMS_PRE_TAX,
        DStreamAF.NAME
        from DStreamAF
        where DStreamAF.AS_OF_DATE='01/05/94'
        and DStreamAF.NAME='DAIMLER-BENZ AG'
```

WorldScope:

```
select WorldcAF.TOTAL_ASSETS, WorldcAF.COMPANY_NAME
        from WorldcAF
        where WorldcAF.COMPANY_NAME='DAIMLER-BENZ AG'
```

Disclosure:

```
select DiscAF.NET_SALES, DiscAF.NET_INCOME,
        DiscAF.COMPANY_NAME
        from DiscAF
        where DiscAF.COMPANY_NAME='DAIMLER-BENZ AG'
```

And the merging conditions will be specified using the following three views: DT, WS and DS to refer to the outputs for DataStream, WorldScope and Disclosure respectively.

```
select WS.WorldcAF.TOTAL_ASSETS, DS.DiscAF.NET_SALES,
DS.DiscAF.NET_INCOME,
DT.DStreamAF.TOTAL_EXTRAORD_ITEMS_PRE_TAX
from WS, DS, DT
where WS.WorldcAF.COMPANY_NAME = DT.DStreamAF.NAME
and WS.WorldcAF.COMPANY_NAME=DS.DiscAF.COMPANY_NAME
```

Next, we use the alignments between the three applications to generate single application receiver contexts. For our example, the only relevant modifier is the name format modifier which is a modifier of the company name semantic type. We will summarize the modifier names and their values for the three applications in the following table.

	DataStream	**WorldScope**	**Disclosure**
Modifier Name	Name_format	Format	Name_fmt
Modifier Values	Worldscope Disclosure Datastream Microsoft	WS DS DT MS	WS_fmt DS_fmt DT_fmt MS_fmt

Figure 5.1 Modifier value table

Now let us suppose the receiver context was Format := DS. The context generator will now go through this table and determine the equivalent modifier values for DataStream and Disclosure. Consequently, the receiver contexts for the three applications will now contain

DataStream: Name_format := Disclosure
WorldScope: Format := DS
Disclosure: Name_fmt := DS_fmt

The same conversions will be applied to all the other modifiers in the receiver context. If a modifier value cannot be found in the table, the mechanism will assume that a common modifier value is used by the other ontologies. We now have three receiver contexts and we are ready to do context mediation.

After we have run the three mediation processes, we store the mediated outputs from the three applications into the views that were mentioned earlier in this chapter (WS, DS and DT). Finally, we use the merging conditions imposed on the federated query to join these views and get the correct output.

6 Possible Improvements and Extensions

Although we have tried to parametrize GCMS, we have not parametrized every component. For instance, the POE (Planner/Optimizer/Executioner) has yet to be parametrized. Some of the difficulty in parametrizing the POE arises from the fact that some data sources are web sources. Since the executioner has to use wrappers to access these sources, the merger cannot simply take a mediated SQL query containing references to web sources and join it with another mediated SQL query. This deficiency accounts for the extra work the merger has to do in building and querying views formed from the results produced by the POE. Clearly, integrating the merger with the POE and making it capable of merging the mediated subqueries before execution will make the system far more efficient so this is an area that could be improved on.

But the major focus of this chapter will be describing how a more powerful alignment can be established between applications using the notion of global ontologies.

6.1 Global Ontologies

The idea here is to be able to relate ontologies to on another through an overlying model instead of doing pair-wise matching as we have done in this implementation. Consequently, different ontologies will elevate to this model much like data sources elevating to an ontology in GCMS. Once this meta-model has been defined, we can define contexts for each ontology based on the meta-model modifiers and we can do context mediation between different ontologies, much the same way we do mediation on sources that elevate to the same ontology. But before we start describing this merging mechanism, let us briefly go through the most common semantic conflicts that arise between ontologies.

6.2 Ontology Level Conflicts

Let us consider how we would try to resolve the semantic heterogeneity if we wanted to extend our tightly-coupled approach to more semantic conflicts. Namely, let us consider how we would try to resolve the semantic conflicts between the three financial ontologies in Appendix A. The first two ontologies have a reasonable number of conflicts between them; the third ontology, on the other hand, has many similarities with both of these ontologies. Our challenge will be to take to distinct ontologies like the first two and tyr to reconcile them. The conflicts that need to be resolved will include language heterogeneity and ontology level mismatches. For our system, we will assume that all ontologies are written in a common language; therefore, we will be disregarding all language related conflicts.

Ontology level conflicts can generally be grouped into three categories [5]:

1. Terminological Mismatches
2. Conceptualization Mismatches
3. Explication Mismatches

6.2.1 Terminological Mismatches

Terminological mismatches refer to syntactic conflicts that are caused by the terminology that is used in writing the ontologies [5]. Two examples of terminological mismatches are synonym conflicts and homonym conflicts.

6.2.1.1 Synonym Mismatches

Synonym conflicts arise when two words are used to describe the same concept (class) in two ontologies [10]. For instance, *financial amount* and *monetary amount* or *curr* and *currency* could be used to refer to the same object in two different ontologies. We have already resolved synonym conflicts in our implementation; our general approach will be explicitly establishing semantic equivalence between synonymous semantic objects in the ontologies. Hence, we will be elevating *financial amount* and *monetary amount* to *FINANCIALS* and *curr* and *currency* to *CURRENCY*.

Synonym alignments get a little more complicated when we are dealing with modifiers. Unlike semantic types and attributes, modifiers have values that can be ontology specific; therefore, we need to address any semantic disparity that might arise between these values. For instance, when we align *curr* and *currency,* we also need to define a modifier for the currency value format. In other words, we need to specify the format in which we are expressing the value of the *CURRENCY* modifier.

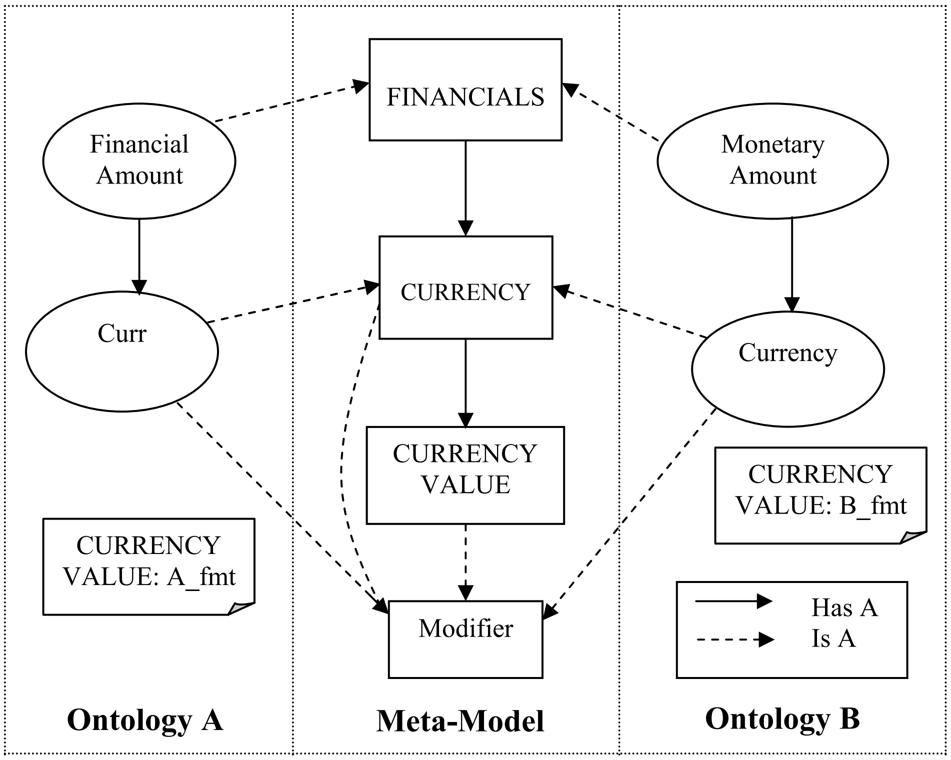


Figure 6.1 Synonym Alignments

For instance, ontology A might refer to US Dollars as *USD* and ontology B might refer to the same value as *USDOL*. Therefore, from the perspective of someone who can see both ontologies only in terms of the meta-model, we need to specify the format that is being used by each ontology. Accordingly, if we named the formats being used by ontology A and B, *A_fmt* and *B_fmt*, we will just have to set the value of the *CURRENCY VALUE* modifier to these values in each respective ontology.

From this point on, we will always assume that a format modifier is added to the meta-model every time two modifiers are aligned.

6.2.1.2 Homonym Mismatches

Homonym conflicts are conflicts caused by words that are used in a different sense in two ontologies [10]. For instance, income could represent a company's income in one financial ontology and the CEO's income in another. In this case, the two types are not semantically related; therefore, we cannot relate the two semantic types.

6.2.2 Conceptualization Mismatches

Conceptualization mismatches arise from differences in the conceptualization of the domain [5]. In other words, two authors can have different conceptualizations of the same domain, and each can represent a domain using different models. The four major conceptualization conflicts are data representation mismatches, scope mismatches, model coverage and granularity mismatches and generalization mismatches.

6.2.2.1 Data Representation Mismatches

Data representation mismatches are caused when two ontologies have two different objects representing the same semantic type [10]. In other words, there is a very explicit relation between the two objects but they are not semantically equivalent. For instance, *income* could be represented as *weekly income* in one financial ontology and *annual income* in another.

In our global ontology approach, we try to resolve this conflict by introducing a modifier for the meta-model which allows us to relate the two semantic types. In this instance, we will elevate both *weekly income* and *annual income* to the same type, *INCOME*, in the metal model and create a *PERIOD* modifier for this income type to distinguish the differences between these different income types. With the introduction of this modifier we now have a new class of contexts, namely ontology contexts. Ontology contexts will be define using meta-model modifiers, and we will be using them to resolve a number of semantic conflicts that can be resolved by defining new modifiers. In this example, we will set the value of *PERIOD* to *weekly* in one ontology and *annual* in the other.

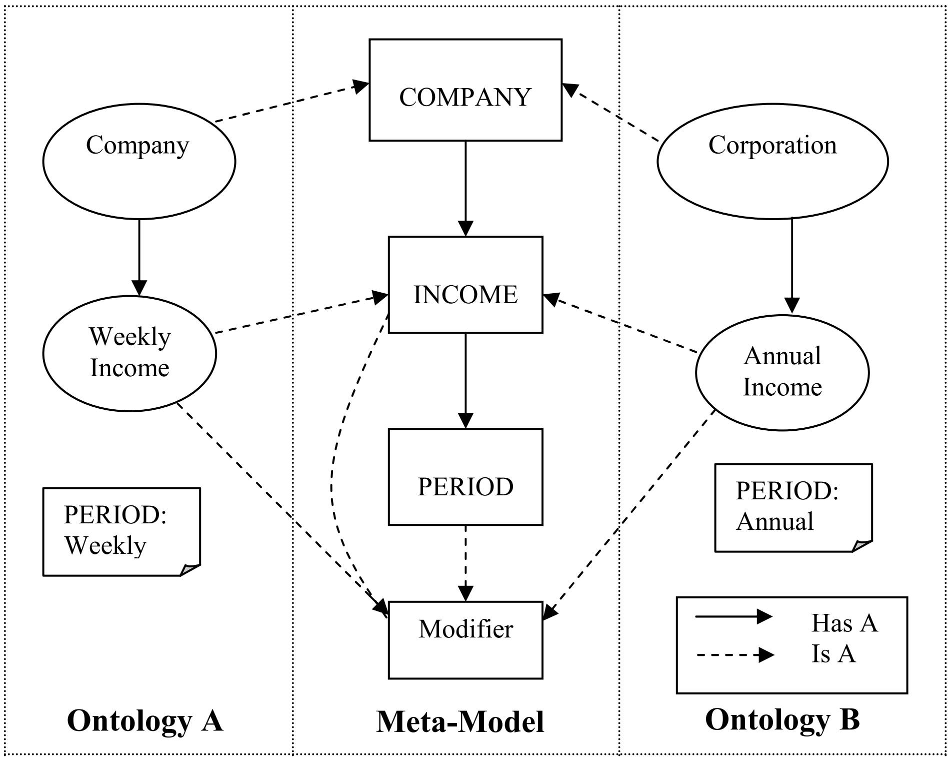


Figure 6.2 Data Representation Alignments

6.2.2.2 Scope Mismatches

Scope mismatches arise when a type is represented in different scopes in two different ontologies [13]. In other words, two ontologies could describe the same semantic type, albeit use different attributes or modifiers. For instance, *report date* might be an attribute for *income* in one financial ontology whereas it might be deemed unnecessary in another. Generally, unless the missing objects are implicitly defined in the ontologies, it is impossible to do any type of alignment. Some cases of scope mismatches overlap with generalization conflicts, and we will be discussing these conflicts shortly.

6.2.2.3 Model Coverage and Granularity Mismatches

Model coverage and granularity mismatches are caused when two ontologies make different levels of distinction between equivalent objects [3]. For instance, one ontology might choose to represent sales as *total sales* whereas another one might choose to go one level further and break it down into *retail sales* and *direct sales*. This would be a typical example of an aggregation conflict.

Although, we will not be going into this in great detail, one possible way of resolving this type of conflict would be defining basic aggregation operators that will be helpful in establishing semantic equivalence using functional definitions. For instance, in this case, we could define a new *TOTAL_SALES* type in the meta-model and elevate *total sales* to this type. We could then align *retail sales* and *direct sales* to *total sales* by applying the aggregation operator to them and elevating them to *TOTAL_SALES*.

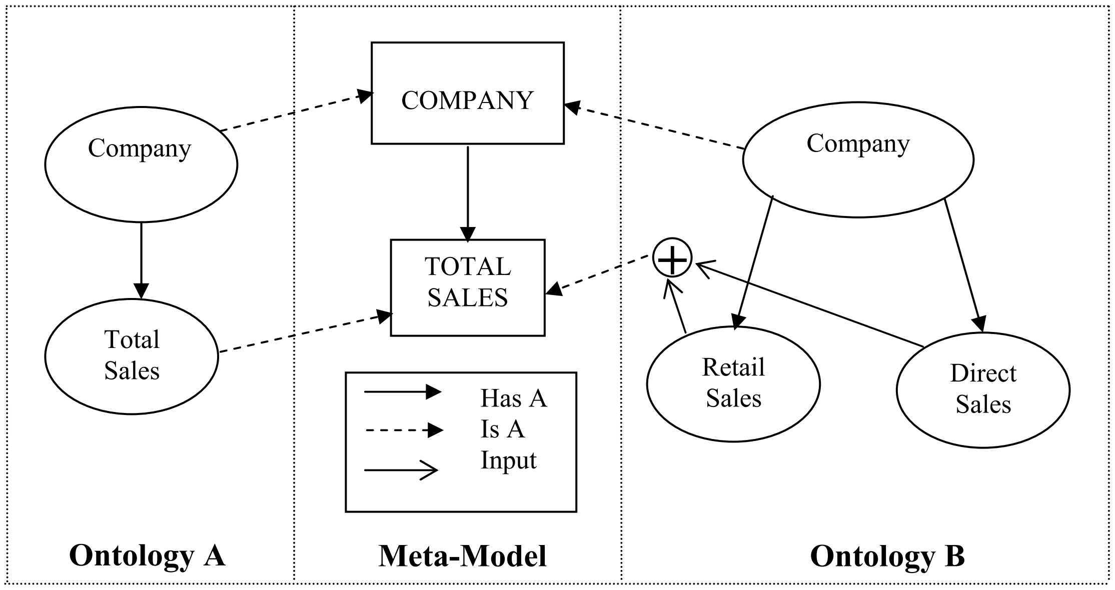


Figure 6.3 Model Coverage and Granularity Mismatches

6.2.2.4 Generalization Conflicts

Generalization conflicts are caused if one ontology has a more general conceptualization of a semantic type [10]. An example of this would be an ontology which has a *location* attribute for *company* and another that assumes that all companies are in the US. This is an example of a generalization mismatch because one ontology is more general and explicitly includes the location whereas the other one already assumes a specific location.

In order to resolve this conflict, we once again introduce a new modifier at the meta-model level. As with most general vs. specific problems, the approach we will be taking is starting with the more specific ontology and attempting to make it coherent with the more general ontology. Accordingly, we start out by elevating the *location* in the more general ontology to a *LOCATION* modifier in the meta-model. We will be elevating this attribute to a modifier because this *LOCATION* is an variable attribute in one ontology and a constant attribute in another.

Now, when we are defining the ontology contexts, we set the value of *LOCATION* to US in the more specific ontology and in the case of the general ontology, the value of the modifier will be inherited from the *location* attribute in the general ontology itself. We can implement this without the introduction of any other modifiers simply by making the value of LOCATION a static type (for the specific ontology) or a dynamic type (for the more general ontology).

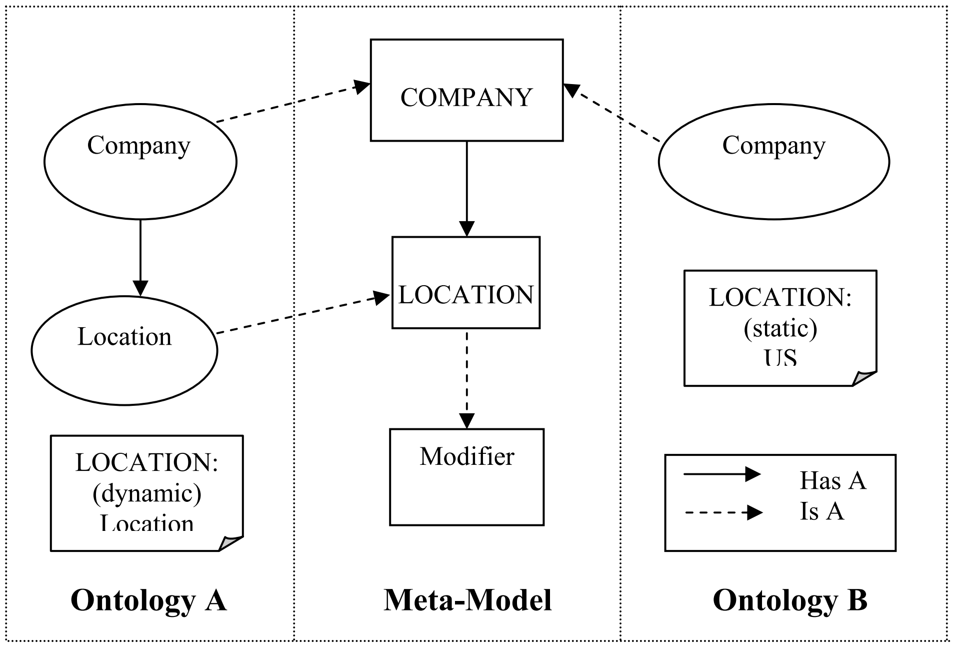


Figure 6.4 Generalization Conflicts

If *location* has modifiers, then the proper modifiers are also created in the meta-model and the values of these modifiers will be included in the ontology contexts. This will be especially important in the case of the more specific ontology because we will need to specify the *LOCATION* context as a static value, using terminology that is coherent with the general ontology.

6.2.3 Explication Mismatches

The last class of ontology level mismatches, explication mismatches, refer to discrepancies that arise from variations in styles of ontology modeling [5]. The three major types of explication mismatches are paradigm conflicts, data precision conflicts and concept description mismatches.

6.2.3.1 Paradigm and Data Precision Mismatches

Paradigm mismatches are caused by the use of different paradigms such as time [3]. An instance of this would be representing stock prices as a *closing value* (point reference) or a daily *high-low range* (interval reference). The same example can also be used for data precision mismatches, which arise when there is one-to-many mapping between semantically related types in two ontologies.

Generally, these types are only semantically related and it is not possible to establish direct semantic equivalence between them without the assistance of an external mechanism (for instance a look up table in this case) which will be used to convert a type in one ontology to its equivalent in another. By *equivalent* we do not mean equal because there is always some approximation involved in resolving data precision mismatches, while going from a general type to a more specific type.

Just like we did with the generalization mismatches, we start with the more precise type and try to express it in terms of the more general type. In other words, we will introduce a new type in the meta-model and elevate both the less precise and more precise types to this type. However, for the more precise type, we will do the elevation under certain conditions that will map the precise value to a less precise value that is coherent with the more general ontology. Hence, if we finally try to do a join involving the two ontologies, we will be able to join only on the level of the more general ontology. In the instance of the stock value, we will be able to compare only daily ranges, as opposed to exact closing values.

6.2.3.2 Concept Description Mismatches

Finally, concept description mismatches are caused by variations in modeling conventions [3]. For instance, a difference between two semantically equivalent classes can be specified using an attribute or using a new class.

Let us consider the company example we had for generalization conflicts. To recap, one *company* object referred to any company whereas the other referred specifically to American companies. To establish a relationship between these two *company* objects in the same ontology we can make *country* an attribute for the more general *company* object and set it to *US* to represent *American company* object. Or equivalently, we can make *American Company* a subclass of *company* using a subsumption relation. To resolve this type of conflict, we will simply elevate both *company* and *American company* to the semantic type and treat it as a typical case of a generalization conflict.

These conflicts can be expressed in terms of other conflicts so we will generally try to reduce the problem into mismatches we have already resolved and proceed with the methodology we have developed for those mismatches.

If we applied this strategy in resolving the conflicts between the first two ontologies in Appendix A, we would end up with a meta-model that strongly resembles the last ontology (C) in that appendix. The following table tries to summarize the conflicts presented here and the approach we are taking in resolving them.

Semantic Conflict	**Resolution**
Synonyms	Direct semantic equivalence

Homonyms	No Resolution due to semantic inequivalence
Data Representation	Definition of a modifier at the meta-model level and use of ontology contexts
Scope	Generally no resolution except for a small class of mismatches that overlap with generalization mismatches
Model Coverage & Granularity	Definition of operators to *relate semantically related* types
Generalization	Definition of a modifier at the meta-model level and use of ontology contexts to reference a general type or a specific value
Paradigm & Data Precision	Semantic equivalence after the conversion of a specific type into a more general type that is coherent with the more general ontology
Concept Description	Reduce to other semantic mismatches and resolve each one separately

Table 6.1 A summary of semantic conflicts and their resolution

6.3 Mediation Using a Global Ontology

Now we know how to resolve a reasonable number of semantic conflicts. The next step is incorporating these conflict resolution techniques in a mechanism that can effectively use them to merge different ontologies. With the global ontology approach, this involves doing mediation between ontologies. Ideally, we want to use GCMS to do this mediation so let us see explore the analogy between data source mediation and ontology mediation. The following table contains the analogous elements from the two mediation processes.

Data Source Mediation	Ontology Mediation
Data Sources	Ontologies
Contexts	Ontology Contexts
Ontology	Meta-model (global ontology)
Elevations	Elevations
Conversion Functions	Conversion Functions

Table 6.2 Analogies between data source and ontology mediation

The only parts of the mediation that are significantly different are the ontology and its meta-model. The contexts, the elevations and the conversion functions are very similar to their data source mediation equivalents; therefore, we will not be spending a lot of time explaining their significance.

Once again, we start the merging process by breaking down the problem into single application mediation sub-problems. We will now be defining the receiver context in terms of the modifiers from the meta-model. Please note that the meta-model will include even modifiers that appear in only one ontology because it is supposed to be a superset of all the underlying ontology. Therefore, the modifiers from the meta-model are a superset of the modifiers from every ontology.

We start out with one receiver context that spans all the concerned applications without any conflicts. We go through each modifier value in this context and try to generate its equivalent in the receiver context for every application. We use the alignments we have established between modifiers to generate these single application receiver contexts. We then proceed to do parallel mediations on each application and we store our results in temporary views.

However, before we can join these views, we need to do mediation on the ontologies and determine how they are related to one another. Our goal is to be able to express all the data in the views in a single ontology receiver context. Therefore, we need to do ontology context mediation on these views. In order to be able to do ontology context medation on these views we first need to elevate them to the meta-model and extend to them the ontology context drafted for their parent ontology. For our receiver ontology context, we choose an arbitrary ontology context or if desired, the user can specify which ontology context to use simply by picking one of the ontologies.

This procedure is slightly different from the routine mediation procedure because it requires two levels of elevation. First we have data sources getting elevated to ontologies, and then we have these ontologies getting elevated to the meta-model. This method bypasses the ontology-to-meta-model elevation and directly elevates data sources to the meta-model. However, it should be noted that the data sources that are being elevated are views containing results from single application mediation processes. In addition, the ontology contexts belonging to the two ontologies are also adopted by the views belonging to each respective application. Hence context mediation is done on the views using the contexts of their parent ontologies. Now we build a query that references these views and contains the same join conditions that were imposed on the original federated query. To get the final result, we pass this query into the ontology context mediation process.

For instance, let us consider the data representation conflict we had in Figure 6.2. Suppose we are trying to merge two tables from the two ontologies by joining the income columns from each table. We first do mediation on each application separately store our mediated results in the temporary views. Now one of these incomes represents weekly income and the other represents annual income; therefore, they are not quite ready for a JOIN yet. In order to resolve this context problem, we now do mediation on these views using a query with the original JOIN conditions.

This approach allows us to resolve a larger number of semantic conflicts, and it also scales better than the pair-wise matching strategy. And finally, it attempts to minimize the amount of additional work by utilizing the existing mediation system.

6.4 Conclusion

We have proposed a strategy that will allow users to query data sources belonging to multiple applications. The strategy employs tightly-coupled ontology merging to resolve the semantic heterogeneity among the ontologies. We then employ a divide and conquer approach to split the federated query into subqueries that are ready for routine mediation. These subqueries will refer to only sources from the same application and the current implementation of GCMS will be used to do mediation on each subquery.
The result from each one of these subqueries will all be returned in the same receiver context. The receiver context for each subquery will be derived from a single receiver context using the context generator. This generator uses synonym alignments made prior to query dispatch to construct identical contexts in each application. Finally, these results are joined using the conditions that were imposed on the federated query.

As we have already mentioned, synonym matching is not a lasting solution to the problem of ontology heterogeneity. The approach proposed in this chapter attempts to give a more complete solution, and yet, there are still conflicts it cannot handle. The next step should be implementing this strategy and extending it to resolve even more semantic conflicts.

References

[1] T. Alatovic. Capabilities Aware Planner/Optimizer/Executioner for COntext INterchange Project. M.Eng Thesis, Massachusetts Institute of Technology, Sloan School of Management, January 2002.

[2] P. A. Bernstein, A. Y. Halevy and R. A. Pottinger. A Vision for Management of Complex Models. Seattle Washington.

[3] H. Chapulsky, E. Hovy and T. Russ. Progress on Automatic Ontology Alignment Methodology. 1997.

[4] B. Czedjo, M. Rusinkiewicz, and D. Embley. An Approach to Schema Integration and Query Formulation in Federated Database Systems. In Proceedings of the 3rd IEEE Conference on Data Engineering, February 1987.

[5] Y. Ding, D. Fensel, M. Klein, and B. Omelayenko, "Ontology management: survey, requirement and directions." *On-To-Knowledge*. Amsterdam, Holland. June 2001.

[6] C. H. Goh, S. Madnick, and M. Siegel. Context interchange: Overcoming the challenges of large-scale interoperable database systems in a dynamic environment. In Proceedings of the Third Int'l Conf on Information and Knowledge Management, Gaithersburg, MD, November 1994.

[7] C. H. Goh. Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Systems. PhD dissertation, Massachusetts Institute of Technology, Sloan School of Management, December 1996.

[8] T. R. Gruber, "A translation approach to portable ontology specifications." *Knowledge Acquisition*, 5(2). 1993.

[9] J. Jannink, P. Mitra, E. Neuhold, S. Pichai, R. Studer, and G. Wiederhold: "An Algebra for Semantic Interoperation of Semistructured Data"; in *1999 IEEE Knowledge and Data Engineering Exchange Workshop (KDEX'99),* Chicago, Nov. 1999.

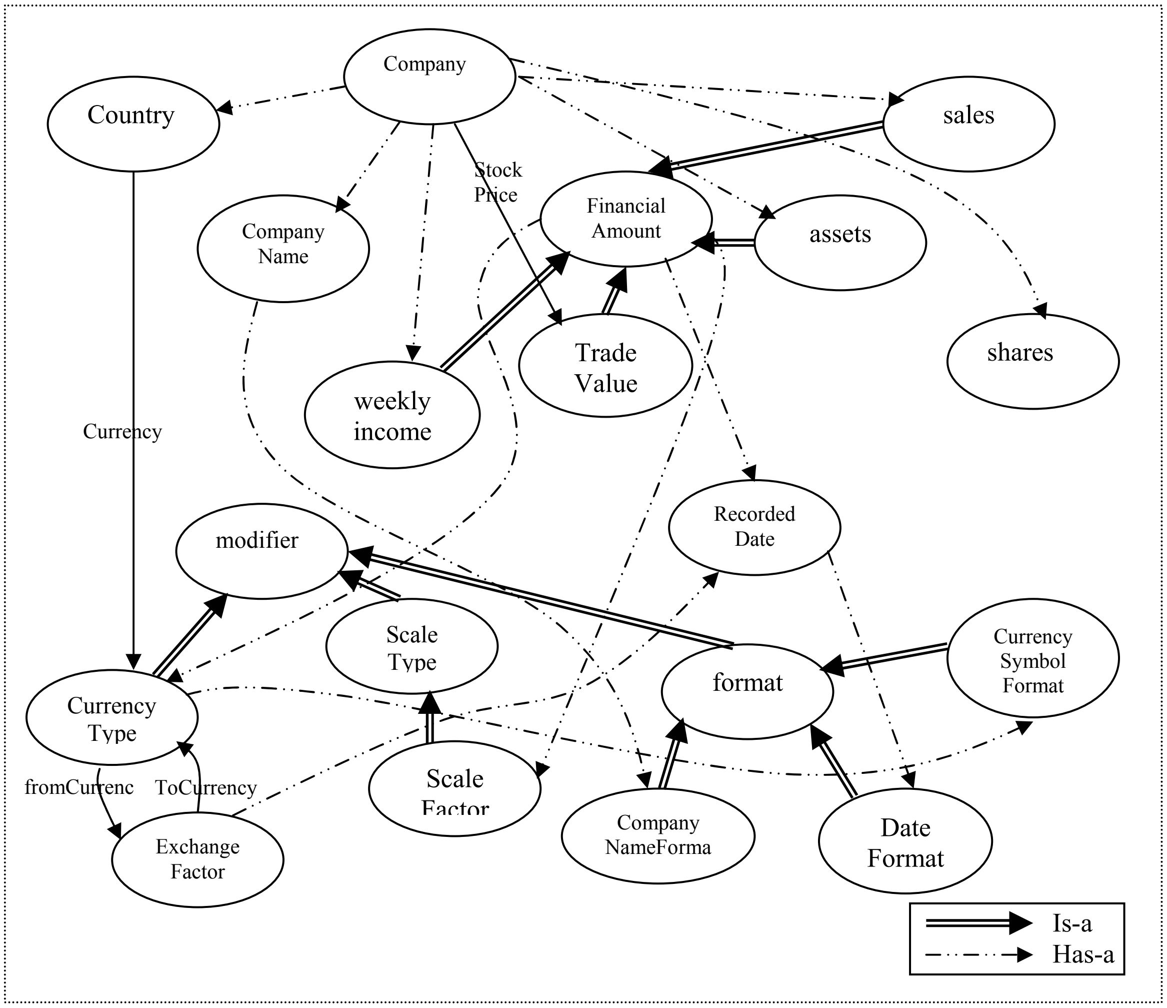
[10] V. Kashyap and A. Sheth. Semantic and Schematic Similarities between Database Objects: A Context-based Approach. In THE VLDB Journal, September 1995.

[11] P. Mitra, G. Wiederhold and J. Jannink: " Semi-automatic Integration of Knowledge Sources"; in *Proceedings of Fusion '99*, Sunnyvale CA, July 1999.

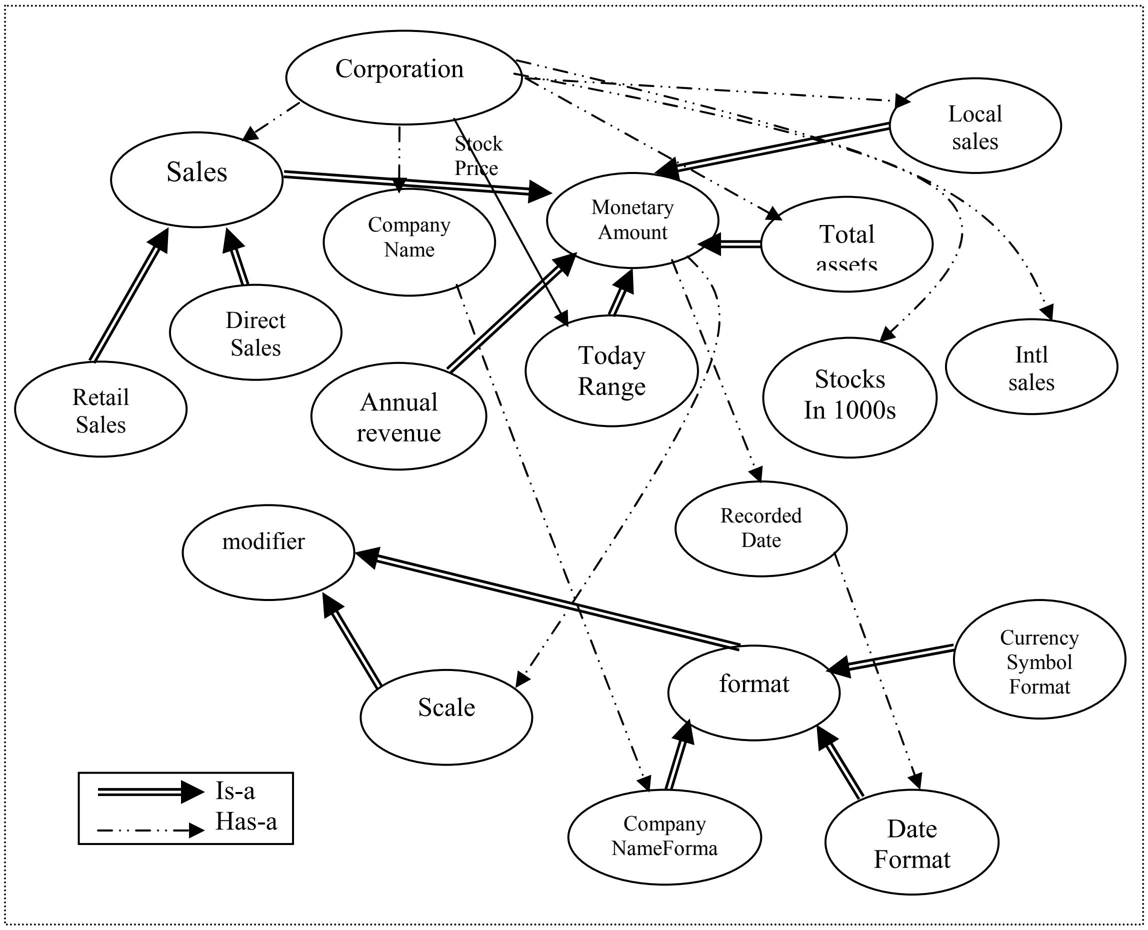
[12] N. F. Noy, M. A. Musen, PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. 2000.

[13] P. R. S. Visser et al. An Analysis of Ontology Mismatches; Heterogeneity Versus Interoperability. In AAAI 1997 Spring Symposium on Ontological Engineering. Liverpool, England. 1997.

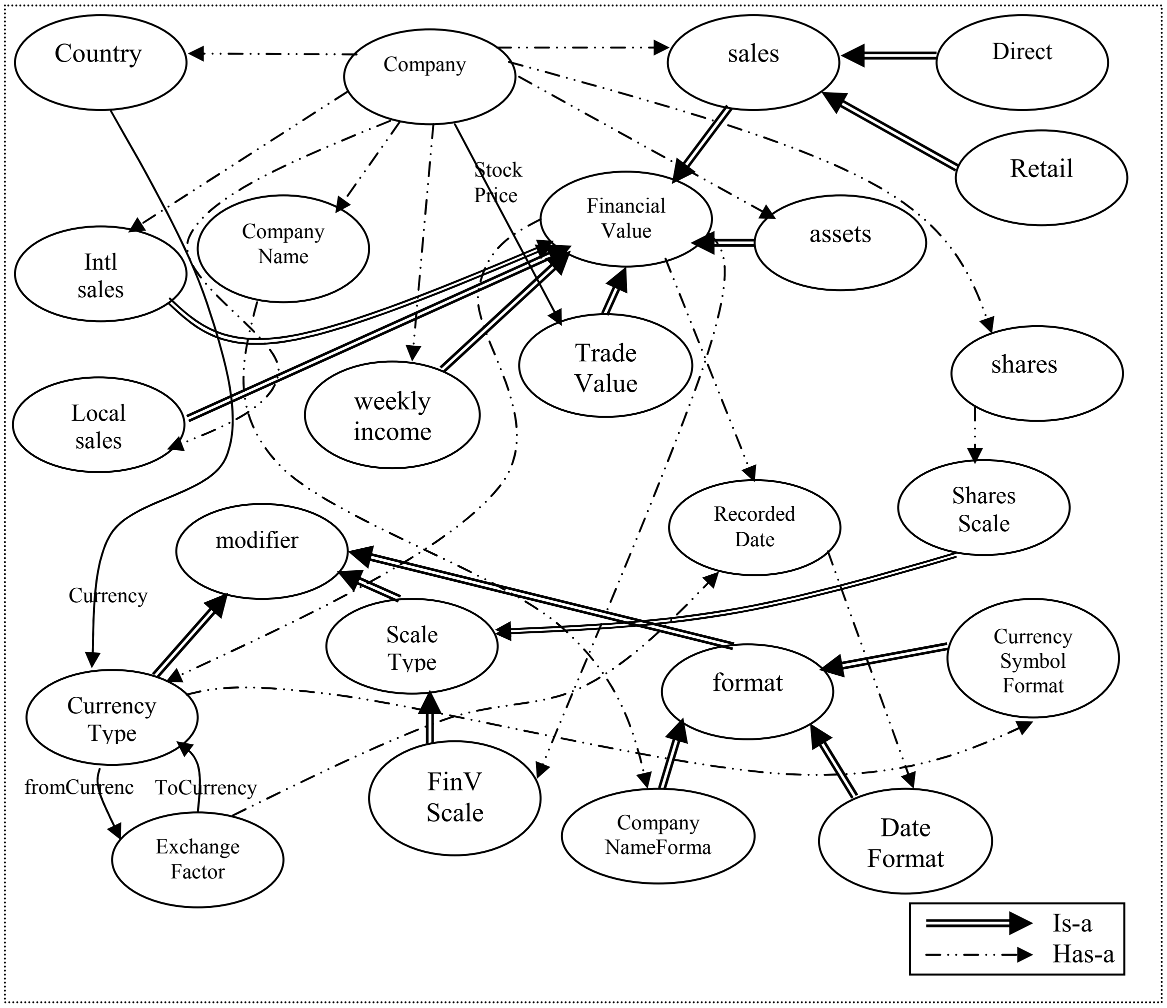
Appendix A



Financial Ontology A



Financial Ontology B



Financial Ontology C