Design and Architecture of the Context Interchange System

by

Syed Ahmed Zaman Shah

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirement of the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 1998

© 1998 M.I.T.
All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute, publicly, paper and electronic copies of this thesis and to grant others the right to do so.

Author

Certified by

Accepted by

Smith

Department of Electrical Engineering and Computer Science

Professor Stuart Madnick
Thesis Supervisor

May 01, 1998

Arthur C.
Chairman, Department Committee on Graduate Theses
Design and Architecture of the Context Interchange System

by

Syed Ahmed Zaman Shah

Submitted to the
Department of Electrical Engineering and Computer Science

May 18, 1998

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

ABSTRACT

The Context Interchange Project (COIN) presents a unique approach to the problem of semantic conflict resolution among multiple heterogeneous data sources. The system presents a uniform view to the user for all the available data sources. The semantic conflicts are automatically detected and reconciled by the system. In this thesis, I describe the COIN system from a both as a user's perspective and an application developer's perspective. Then I discuss the current architecture of the system from the software-engineering point of view and propose modifications to the system that will make the system more modular and easy for the system developers to maintain and enhance the system.

Thesis Supervisor: Professor Stuart Madnick
Title: Jack Norris Maguire Professor of Information Technology
Acknowledgments

I would like to thank many people for their support and guidance throughout my graduate work. First, I would like to thank Professor Stuart Madnick and Dr. Michael Siegel, for their guidance and support, both academic and personal, throughout the last one year. My most sincere thanks to Ste’phane Bressan for his immeasurable help with my work. I cannot thank him enough for putting up with me and my ignorance while all that time keeping me on track and helping me whenever I needed guidance. I would also like to thank my fellow COINers, Allen Moulten, Thom Lee, Ricardo Ambrose and many others who made it a very fun and interesting place to work and learn. My deepest thanks to my family, particularly my mother and my younger brother, Yassir Shah(whose guidance and love are the primary reasons for my academic success), for their support, confidence, and love. Thanks to my friends, particularly, Omar Raza Khan and Srikar Srinath for putting up with me when I needed somebody to talk to. Most of all, I would like to thank Allah for all the opportunities and successes that He granted me over the years.
I dedicate all my work to my father Syed Fida Hussain Shah (1921-1994).
# Table of Contents

1. **INTRODUCTION** ......................................................................................................................... 8

2. **OVERVIEW** ................................................................................................................................. 15

   2.1 THE COIN FRAMEWORK........................................................................................................ 15

       2.1.1 Context Mediator .................................................................................................... 16

       2.1.2 System Perspective ................................................................................................ 16

       2.1.3 Application Domains ........................................................................................... 17

   2.2 TOP LEVEL ARCHITECTURE ............................................................................................ 18

       2.2.1 Client Processes ..................................................................................................... 19

       2.2.2 Server Processes .................................................................................................. 19

       2.2.3 Mediator Processes ............................................................................................. 20

3. **COIN ARCHITECTURE** ........................................................................................................... 21

   3.1 A MOTIVATIONAL EXAMPLE .......................................................................................... 21

   3.2 DOMAIN MODEL AND CONTEXT DEFINITION .................................................................. 24

   3.3 ELEVATION AXIOMS .......................................................................................................... 26

   3.4 MEDIATION SYSTEM .......................................................................................................... 28

       3.4.1 SQL to Datalog Query Compiler ........................................................................ 29

       3.4.2 Mediation Engine .................................................................................................. 30

       3.4.3 Query Planner and optimizer .............................................................................. 33

       3.4.4 Runtime engine ...................................................................................................... 37

   3.5 WEB WRAPPER ..................................................................................................................... 37

       3.5.1 Web wrapper architecture .................................................................................... 39

       3.5.2 Wrapping a web site ............................................................................................... 40

4. **BUILDING AN APPLICATION** .................................................................................................. 42

   4.1 PRE-APPLICATION SETUP ............................................................................................... 43

   4.2 DOMAIN MODEL DEFINITION ......................................................................................... 44

   4.3 IDENTIFICATION OF THE SOURCES AND RELATIONS ..................................................... 45

   4.4 SOURCE AND RELATION DEFINITION ........................................................................... 47

       4.4.1 Inserting the Data Sources .................................................................................. 47

       4.4.2 Inserting the Relations ......................................................................................... 48

   4.5 ELEVATION OF THE DATA SOURCES .............................................................................. 51

       4.5.1 Mapping the Physical Types to Semantic Types ................................................ 52

       4.5.2 Creating the Semantic Relations ........................................................................... 53

       4.5.3 Creating Views to Ease Application Development ............................................. 55

   4.6 ADDITION OF INTEGRITY CONSTRAINTS ............................................................................ 57

   4.7 CONTEXT DEFINITION FOR EACH DATA SOURCE .......................................................... 58

       4.7.1 Recognizing the Semantic Differences ................................................................... 58

       4.7.2 Creating the Contexts ............................................................................................. 60

   4.8 ADDITION OF THE CONVERSION FUNCTIONS .............................................................. 61

   4.9 APPLICATION GENERATION ............................................................................................... 64
Chapter 1

1. Introduction

The only thing more important than a well-designed software system is a well-written set of documentation for that system. Any system that does anything useful has to be accompanied by documentation. The most common form of documentation is a user manual explaining the features of the systems for the users of that system and how to use those features. For a very small system e.g. `ls` command on a UNIX system, that might just be one page listing all the command line flags for that command and a brief explanation of what each flag does. But for large systems e.g. operating systems or database management systems, the documentation might contain many volumes. The Oracle DBMS V7 [19] comes with a documentation set that contains 32 volumes each about 500-700 pages in length. Large software companies spend a significant amount of time and resources trying to produce a coherent set of documentation for the users.

Large software systems usually offer a very rich set of features. This large feature set however comes at the cost of complexity. Therefore in order for the users of these systems to be able to make use of all those features, good documentation plays an important part. No matter how many features a system offers, if they are not well documented and hence cannot be made use of, then they might as well not exist. A good set of documentation goes farther than that. It even saves time and money for the system developers since rather than them having to explain how to use the system to the users over and over again, the system designers and developers can expend one-time effort producing a comprehensive set of documentation once. Once a good set of documents is produced, the users can then simply learn about the features of the system by reading the manuals and figure out how to use the systems. Meanwhile, the system designers can spend their precious time designing the next great version of the system.
The documentation mentioned above is usually referred to as user documentation as it is
directed towards the end users of the system. There is another type of documentation
whose target audience are not the users of the system, but the developers of the system.
This type of documentation is usually called system documentation. While the user
documentation is primarily concerned about how to use the system, the system
documentation in essence captures the knowledge about the system itself. These
documents contain information about how the system was designed. These usually include
the overall architecture of the system, design and in some cases implementation of each
sub-system (e.g. the algorithms used etc.). In addition to that, the interfaces of each sub-
system are explained, which essentially provides the information about how each sub-
system interacts with other sub-systems and how they all fit together to form a complete
system.

There is yet another type of documentation that often accompanies large database systems
and Information Systems. This set of documentation is directed at System Administrators.
This documentation deals with the issues of how to install and configure the system based
on the need of each site and how to manage and administer the system. An example might
be a database management system. In this case, the documentation would discuss various
configurations of systems based on the number of users that are going to use the system
and how big the database is expected to grow. The documentation would list how to
install the various sub-systems, how to configure each one those sub-system, how to
create initial database, issues on how to bring up or shutdown a database. The
documentation would also talk about how to create and maintain user accounts,
permissions on who can access what part of the database and such. This set of
documentation quite important in the above-mentioned systems and is normally called
System Administration Guide.

For most systems however, user document and the design document are still the most
important categories of documentation. While the user documentation is important, almost
necessary for the users to be able to use the system, design documentation is equally
important for a number of reasons. The first and most important use of design document is that it serves as a roadmap for the system designers as to where the systems stand and where it is heading. Since the design document captures the knowledge about the system, the designers, while talking about the future versions of the system, know exactly what features the system currently offers and what are the features that should be added to the system. In addition to that, as the systems grow large, it becomes almost impossible for a single person to know all the features and their design intimately. The design document serves as a central place where the architecture of the whole system is captured, thus making it easy for the designers to analyze how the various design decisions for new features would integrate into the existing system. This allows the system designers to make the design choice for the new features that would best fit with the current architecture of the system.

While the system designers work on the design of the system, system developers are the ones who actually work on the implementation of the system. If the system designers do a good job of writing the design document and system specifications, system developers can then just take the system design and use the specification to go ahead and actually implement the features designed by the system. Of course the system developers will be talking to the system designers in a constant feedback cycle, since the developers while developing the system might come across cases that the system designers either overlooked or forgot to account for which would require revisions of the system design specifications. But if the design documents were poorly written, this would cause a much higher communication overhead, as the system developers would be needlessly asking the system designers for explanations. The more important benefit of the design document is that it allows for modular development. If each subsystem is adequately defined and specified, then the developer who is implementing that particular subsystem need not worry about any other part of the system. As long as she implements her subsystem according to the specified specification, the various subsystems can be developed in parallel and in the end be integrated into the main system with relative ease.
Over the lifetime of large software systems, a great number of developers and designers work on the design and development of the system. Usually with the passage of time, the original system designers move on to do other projects or work on other systems. New people come on board and carry on the work of people who left. As the original designers leave, rather than walking out of the door with all the knowledge of the system, they need to transfer the knowledge of the system that they possess to the new designers that are taking their place. This can be achieved in two ways. The previous designers can hold training sessions where they can transfer the knowledge to the new designers in person by having one-on-one meeting. This method is cumbersome, requires a lot of co-ordination (which is not always possible as the end dates of the leaving designers and the starting dates of the new designers do not always coincide) and can be flawed as the old designers might forget some crucial details of the system design. Now instead of having to train the people who are replacing them by doing a manual brain dump, if the departing designers have been maintaining a good design document, they do not any of the above-mentioned issues to deal with. The retiring designers can leave anytime at their convenience. The new people can then simply pick up the design documentation and come up to speed after reading the documents and become productive very quickly.

During the latter phases of the software lifecycle, the number of new features that are being added to the system goes down and the system slowly goes into maintenance mode. During that time, very few people are working on the system design and system maintainers who simply fix bugs and field customer request do most of the work. A good and well-maintained set of design document becomes very crucial during that phase the software life. As most of the original people who actually developed the system have left by then, the system maintainers who do not normally have intimate knowledge of the workings of the systems have just the design documents to go by. Thus good documents save the maintainers a lot of time and the project administrators a lot of money.

To summarize, documentation is a crucial part of the software systems and its importance cannot be underestimated. There are two types of documentation, user documentation and
design documentation. User documentation usually explains how to install and use the system, hence it concentrates on the features of the systems and how those features can be used. The design documentation, on the other hand, is targeted towards the designers and the developers of the system itself. The design document contains the information about the overall architecture of the system, design and possibly implementation of each subsystem and how each subsystem interacts with the rest of the subsystems to form the complete system. Both sets of documentation are necessary and crucial for the large software systems.

In the recent past, a lot of research has been done by various researchers on the Context Interchange System. Over the last few years, the system has undergone major revision overhaul and a lot of features have been added to the system. In most cases, as individual researchers were working on certain parts of the system, they carried on their work in those subsystems that they were interested in and mostly ignored the rest of the system. They did not give a lot of thought as to how their newly modified subsystem or in some cases, entirely new subsystem would integrate with the rest of the system. Another factor that contributed to the “chaos” was that the Context Interchange group by its nature has a very dynamic environment. Every year students graduate and leave and new students take their place. The graduating students barely spend any time worrying about above-mentioned issues like how the subsystem that they worked on would integrate with the rest of the system. In most cases they work on their subsystem in isolation, get it to work, make some effort to integrate into the rest of the system. If it works, wonderful, if not, well, they leave anyway. It would however be unfair to put all the blame on all these researchers. They are constrained for time and also by the fact that a good set of design documentation that can guide them does not exist. Researchers who are working on some specific pieces of the system either write papers or a thesis and thus preserve that knowledge. However, there have not been a concerted effort by some person to take all that is out there, and come up with a set of documentation for the system.
That is the hole this thesis plugs. In this thesis, I will be providing a set of coherent documentation for the Context Interchange System (COIN). The documentation is broadly divided into two sets of documents. Each set of documentation is directed at a certain task. The two sets are:

- Application Development Tutorial: This document is written to teach the users of the system on how to write applications on top of COIN system for the end users. This document is in the form of a tutorial. We start with an idea of an application and then guide the user step by step on how to build that application.

- System design and architecture document: This document in some sense is the main document as it captures the design and architecture of the whole system. While primarily targeted at designers, who want to know the design and architectures of the COIN system, this document has enough depth that it also serves the purpose of an overall design document.

In addition to the above, the thesis also contains a discussion of the current design of the COIN system. It talks about the various tradeoffs and then based on the current design, proposes modifications to the current design that will improve the modularity of the system and make it ready for the distributed development that we are heading. I will discuss all the advantages and tradeoffs of the proposed modifications as compared to the existing design in an effort to prove that it is indeed beneficial for us to move towards the new design changes.

The rest of the thesis is as follows. Next chapter contains a high level architecture of the system. Chapter 3 discuses each sub component in somewhat using an example scenario. Chapter 4 consists of application development tutorial with the example. Chapter 5 is slightly different in tone from the rest of the chapters in that while the other chapters are primarily system documentation, this chapter departs from the theme and instead critiques the current design of the COIN system from the software-engineering point of view. This chapter talks about some of the shortcoming of the existing system and then proposes
some not too radical modifications to the design that will address those shortcomings. Chapter 6 concludes the thesis. Appendices contain all the source code that is used in building the example application discussed in this thesis.
Chapter 2

2. Overview

The COIntext INterchange (COIN) strategy seeks to address the problem of semantic interoperability by consolidating distributed data sources and providing a unified view over them. COIN technology presents all data sources as SQL databases by providing generic wrappers for them. The underlying integration strategy, called the coin model, defines a novel approach for mediated [9] data access in which semantic conflicts among heterogeneous systems are automatically detected and reconciled by the Context Mediator. Thus the COIN approach integrates disparate data sources by providing semantic interoperability (the ability to exchange data meaningfully) among them.

2.1 The COIN Framework

The COIN framework is composed of both a data model and a logical language, COINL [11], derived from the family of F-Logic [10]. The data model and language are used to define the domain model of the receiver and data source and the context [12] associated with them. The data model contains the definitions for the “types” of information units (called semantic types) that constitute a common vocabulary for capturing the semantics of data in disparate systems. Contexts, associated with both information sources and receivers, are collections of statements defining how data should be interpreted and how potential conflicts (differences in the interpretation) should be resolved. Concepts such as semantic-objects, attributes, modifiers, and conversion functions define the semantics of data inside and across contexts. Together with the deductive and object-oriented features inherited from F-Logic, the COIN data model and COINL constitute an appropriate and expressive framework for representing semantic knowledge and reasoning about semantic heterogeneity.
2.1.1 Context Mediator

The Context Mediator is the heart of the COIN project. It is the unit that provides mediation for user queries. Mediation is the process of rewriting queries posed in the receiver's context into a set of mediated queries where all potential conflicts are explicitly solved. This process is based on an abduction [13] procedure which determines what information is needed to answer the query and how conflicts should be resolved by using the axioms in the different contexts involved. Answers generated by the mediation unit can be both extensional and intentional. Extensional answers correspond to the actual data retrieved from the various sources involved. Intentional answers, on the other hand, provide only a characterization of the extensional answer without actually retrieving data from the data sources. In addition, the mediation process supports queries on the semantics of data that are implicit in the different systems. There are referred to as knowledge-level queries as opposed to data-level queries that are enquires on the factual data present in the data sources. Finally, integrity knowledge on one source or across sources can be naturally involved in the mediation process to improve the quality and information content of the mediated queries and ultimately aid in the optimization of the data access.

2.1.2 System Perspective

From the systems perspective, the COIN strategy combines the best features of the loose- and tight-coupling approaches to semantic interoperability [14] among autonomous and heterogeneous systems. Its modular design and implementation funnels the complexity of the system into manageable chunks, enables sources and receivers to remain loosely-coupled to one another, and sustains an infrastructure for data integration. This modularity, both in the components and the protocol, also keeps our infrastructure scalable, extensible, and accessible [2]. By scalability, we mean that the complexity of creating and administering the mediation services does not increase exponentially with the number of participating sources and receivers. Extensibility refers to the ability to incorporate changes into the system in a graceful manner; in particular, local changes do
not have adverse effects on other parts of the system. Finally, accessibility refers to how a user in terms of its ease-of-use perceives the system and flexibility in supporting a variety of queries.

2.1.3 Application Domains

The COIN technology can be applied to a variety of scenarios where information needs to be shared amongst heterogeneous sources and receivers. The need for this novel technology in the integration of disparate data sources can be readily seen in the following examples.

One very obvious and useful application of the COIN technology is in the context of financial domain. This technology can be successfully used to assist financial analysts in conducting research and valuing and comparing companies in this global economy where companies follow different business practices and accounting principals depending on the country they operate in. When the financial analysts analyze different companies that might not be based in the same countries, and compare those companies, they rely on the financial data provided by various information providers. There are a number of information providers that provide historical data and other research both to institutions (investment banks, brokerages) as well as individual investors. Most of the time this information is presented in different formats and must be interpreted with different rules. Some examples are scale-factors and currency of monetary figures. Mismatches among these assumptions across sources of inside one source can be critical in the process of financial decision making.

In the domain of manufacturing inventory control, the ability to access design, engineering, manufacturing and inventory data pertaining to all parts, components, and assemblies are vital to any large manufacturing process. Typically, thousands of contractors play roles and each contractor tends to set up its data in its own individualistic manner. Managers may need to reconcile inputs received from various contractors, compare various tenders submitted by different contractors for a specific order in order to
determine the best bid among all the bids made in order to optimize inventory levels and ensure overall productivity and effectiveness. COIN technology can play an important role in standardizing various formats the contractors put their bid in thus facilitating the job of the managers while choosing the best bid.

Finally, the modern health care enterprise lies at the nexus of several different industries and institutions. Within a single hospital, different departments (e.g. internal medicine, medical records, pharmacy, admitting, and billing) maintain separate information systems yet must share data in order to ensure high levels of care. Medical centers and local clinics not only collaborate with one another but also with state and federal regulators, insurance companies, and other payer institutions. This sharing requires reconciling differences such as those of procedure codes, medical supplies, classification schemes, and patient records.

2.2 Top Level Architecture

Figure 1: COIN System Overview
The feasibility and features of the proposed strategy of using context mediation to solve semantic differences between various heterogeneous data sources has been concretely demonstrated in a working system that provides mediated access to both on-line structured databases and semi-structured data sources such as web sites. This demonstration system implements most of the important concepts of context interchange strategy and is called the COIN System. This section introduces the COIN System and gives a very high level architectural view of the system. The infrastructure leverages on the World Wide Web in a number of ways. First, we rely on the hypertext transfer protocol for the physical connectivity among sources and receivers and the different mediation components and services. Second, we employ the hypertext markup Language and Java for the development of portable user interfaces. Figure 1 shows the architecture of the COIN system. It consists of three distinct groups of processes.

2.2.1 Client Processes

The client processes provide the interaction with receivers and route all database requests to the Context Mediator. An example of a client process is the multi-database browser [15], which provides a point-and-click interface for formulating queries to multiple sources and for displaying the answers obtained. Specifically, any application program that posts queries to one or more sources can be considered a client process. This can include all the programs (e.g. spread sheet software programs like Excel) that can communicate using the ODBC bridge to send SQL queries and receive results. It does not matter to the backend system where the requests are coming from or how as long as they are well-formatted SQL queries.

2.2.2 Server Processes

Server processes refer to database gateways and wrappers. Database gateways provide physical connectivity to database on a network. The goal is to insulate the Mediator Process from the idiosyncrasies of different database management systems by providing a uniform protocol for database access as well as canonical query language (and data model)
for formulating the queries. Wrappers, on the other hand, provide richer functionality by allowing semi-structured documents on the World Wide Web to be queried as if they were relational databases. This is accomplished by defining an *export schema* for each of these web sites and describing how attribute-values can be extracted from a web site using pattern matching [16].

### 2.2.3 Mediator Processes

The Mediator processes refer to the system components that collectively provide the mediation services. These include SQL-to-datalog compiler, context mediator, and query planner/optimizer and multi-database executioner. SQL-to-Datalog compiler translates a SQL query into its corresponding datalog format. Context mediator rewrites the user-provided query into a mediated query with all the conflicts resolved. The planner/optimizer produces a query evaluation plan based on the mediated query. The multi-database executioner executes the query plan generated by the planner. It dispatches subqueries to the server processes, collates the intermediary results, and returns the final answer to the client processes.

Of these three distinct groups of processes, the most relevant to our discussion is the group of mediator processes that run the mediation system as those are the processes that comprise the COIN backend system. In the next chapter, we discuss the mediation process and all the sub components that make up the mediation process. We first start by a scenario of a stock analyst who needs to gather information from a collection of sources each having different assumptions about the data. We will explain how mediation can make the analyst's job much easier. After that, we expand on that example and first explain the notion of a domain model and then move on to talk about each subsystem in detail.
In this section, we use an example of a financial analyst doing research on Daimler Benz. She needs to find out the net income, net sales, and total assets of Daimler Benz Corporation for the year ending 1993. In addition to that, she needs to know the closing stock price of Daimler Benz. She normally uses the financial data stored in the database Worldscope, but for this assignment, Worldscope does not have all the information that she needs. She recalls Jill, her technophile coworker telling her about two new databases, Datastream and Disclosure and how they contained all the information that Jill needed. So she sets off to get the information that she needs. She starts off with the Worldscope database. She knows that Worldscope has total assets for all the companies. She logs into the Oracle, which contains the Worldscope data and issues a query:

```
select company_name, total_assets from worldscope
where company_name = 'DAIMLER-BENZ AG';
```
She immediately gets back the result:

**DAIMLER-BENZ AG 5659478**

Satisfied, she moves on. After looking at the data information for the new databases, she figures out that she can get the data on net income from *Disclosure* and net sales from *Datastream*. For net sales, she issues the query:

```sql
select company_name, net_income from disclosure
where company_name = 'DAIMLER-BENZ AG';
```

The query does not return any records. Puzzled, she checks for typos and tries again. She knows that the information exists. She tries one more time this time entering partial name for DAIMLER BENZ.

```sql
select company_name, net_income from disclosure
where company_name like 'DAIMLER%';
```

She gets the record back:

**DAIMLER BENZ CORP 615000000**

She now realizes that the data sources do not conform to the same standards, as it becomes obvious from the fact that the two data sources store the name of the same company differently. Cautious, she presses on and issues the third query:

```sql
select name, total_sales from datastream
where name like 'DAIMLER%';
```

She gets the result:

**DAIMLER-BENZ 9773092**

As she is putting the results together, she realizes that there a number of things unusual about this data set. First of all, the total sales are twice as much as the total assets of the company, which is highly unlikely for a company like Daimler Benz. What is even more
disturbing is that net income is almost ten times as much as net income. She immediately realizes something is wrong and grudgingly opens up the fact sheet that came with the databases. Upon studying them, she finds out some interesting facts about the data that she was using so gaily. She finds out that Datastream has a scale factor of 1000 for all the financial amounts, while Disclosure uses a scale factor of one. In addition, both Disclosure and Datastream use the country of incorporation for the currency. She knew that Worldscope used a scale factor of 1000 but at least everything was in U.S Dollars. Now she has to reconcile all the information by finding a data source (possibly on the web) that contains the historical currency exchange rates. In addition to that, she still has to somehow find another data source and get the latest stock price for Daimler Benz. For that, she knows she will first have to find out the ticker symbol for Daimler Benz and then look it up using one of the many stock quote servers on the web.

Enter context mediation. The context mediation system automatically detects and resolves all the semantic conflicts between all the data sources being used and presents the results to the user in the format that the user is familiar with. In the above example, if the analyst was using context mediation system instead, all she had to do was formulate and ask only one query without having to worry about the underlying differences between the data after stating that she wanted the results back in the Worldscope context:

```sql
select worldscope.total_assets, datastream.total_sales, disclosure.net_income, quotes.Last
from worldscope, datastream, disclosure, quotes where
worldscope.company_name = "DAIMLER-BENZ AG" and
datastream.as_of_date = "01/05/94" and
worldscope.company_name = datastreama.name and
worldscope.company_name = disclosure.company_name and
worldscope.company_name = quotes.cname ;
```

The system would then have detected all the conflicts that the analyst had to deal with when she was not using the context mediation system, and would have resolved those conflicts without the analyst’s explicit intervention.
In the next few sections, we will use the above scenario to discuss the concept of a domain model and how a domain model is defined and created. We will also explain the various subsystems and how they work.

### 3.2 Domain Model and Context definition

The first thing that we need to do before we can start using context mediation to have concept and definition of the domain model in the application domain that we are working in. A domain model specifies the semantics of the "types" of information units, which constitutes a common vocabulary used in capturing the semantics of data in disparate sources. In other words it defines the ontology which will be used. The various semantic types, the type hierarchy, and the type signatures (for attributes and modifiers) are all defined in the domain model. Types in the generalized hierarchy are rooted to system types, i.e. types native to the underlying system such as integers, strings, real numbers etc.

![Financial Domain Model](image)

**Figure 3: Financial Domain Model**

Figure 3 is lists the domain model that is used in our example. In the domain model described, there are three kinds of relationships expressed.

**Inheritance:** This is the classic type inheritance relationship. All semantic types inherit from basic system types. In the domain model, type `companyFinancials` inherits from basic type `string`.
**Attributes:** In COIN [17], objects have two forms of properties, those which are structural properties of the underlying data source and those that encapsulate the underlying assumptions about a particular piece of data. *Attributes* access structural properties of the semantic object in question. For instance, the semantic type companyFinancials has two attributes, company and fyEnding. Intuitively, these attributes define a relationship between objects of the corresponding semantic types. Here, the relationship formed by the company attribute states that for any company financial in question, there must be corresponding company to which that company financial belongs. Similarly, the fyEnding attribute states that every company financial object has a date when it was recorded.

**Modifiers:** These define a relationship between semantic objects of the corresponding semantic types. The difference though is that the values of the semantic objects defined by the modifiers have varying interpretations depending on the context. Looking once again at the domain model, the semantic type companyFinancials defines two modifiers, scaleFactor and currency. The value of the object returned by the modifier scaleFactor depends on a given context.

Once we have defined the domain model, we need to define the contexts for all the sources. In our case, we have three explicit data sources with the assumptions about their data in Figure 4. A simplified view of how part of a context is included. A complete discussion of what each of these statements mean is in Chapter 4.

```prolog
modifier(companyFinancials, 0, scaleFactor, c_ws, M):-
    cste(basic, M, c_ws, 1000).

modifier(companyFinancials, 0, currency, c_ws, M):-
    cste(currencyType, M, c_ws, "USD").

modifier(date, 0, dateFmt, c_ws, M):-
    cste(basic, M, c_ws, "American Style /").
```
One last thing that needs to be provided as part of a context is the set of conversion functions between different contexts. An example is the conversion between scale factors in different contexts. Following is the conversion routine that is used when scale factors are not equal:

cvt(companyFinancials, _O, scaleFactor, Ctxt, Mvs, Vs, Mvt, Vt) :-
    Ratio is Mvs / Mvt,
    Vt is Vs * Ratio.

<table>
<thead>
<tr>
<th>Datasource</th>
<th>Scale Factor</th>
<th>Currency</th>
<th>Date Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worldscope</td>
<td>1000</td>
<td>USD</td>
<td>American “/”</td>
</tr>
<tr>
<td>Disclosure</td>
<td>1</td>
<td>Local</td>
<td>American “/”</td>
</tr>
<tr>
<td>datastream</td>
<td>1000</td>
<td>Local</td>
<td>European “/”</td>
</tr>
<tr>
<td>Olsen</td>
<td>1</td>
<td>Local</td>
<td>European “/”</td>
</tr>
<tr>
<td>Quote</td>
<td>1</td>
<td>USD</td>
<td>American “/”</td>
</tr>
</tbody>
</table>

Figure 4

3.3 Elevation Axioms

The mapping of data and data-relationships from the sources to the domain model is accomplished via the elevation axioms. There are three distinct operations that define the elevation axioms:

- Define a virtual semantic relation corresponding to each extensional relation.
- Assign to each semantic object defined its value in the context of the source.
Map the semantic objects in the semantic relation to semantic types defined in the domain model and make explicit any implicit links (attribute initialization) represented by the semantic relation.

We will use the example of the relation Worldscope to show how the relation is elevated. The Worldscope relation is a table in oracle database and has the following columns:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPANY_NAME</td>
<td>VARCHAR2(80)</td>
</tr>
<tr>
<td>LATEST_ANNUAL_FINANCIAL_DATE</td>
<td>VARCHAR2(10)</td>
</tr>
<tr>
<td>CURRENT_OUTSTANDING_SHARES</td>
<td>NUMBER</td>
</tr>
<tr>
<td>NET_INCOME</td>
<td>NUMBER</td>
</tr>
<tr>
<td>SALES</td>
<td>NUMBER</td>
</tr>
<tr>
<td>COUNTRY_OF_INCORP</td>
<td>VARCHAR2(40)</td>
</tr>
<tr>
<td>TOTAL_ASSETS</td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

And here is part of how the elevated relation looks like:

'WorldcAF_p' (  
    skolem(companyName, Name, c_ws, 1, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),  
    skolem(date, FYEnd, c_ws, 2, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),  
    skolem(basic, Shares, c_ws, 3, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),  
    skolem(companyFinancials, Income, c_ws, 4, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),  
    skolem(companyFinancials, Sales, c_ws, 5, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),  
    skolem(companyFinancials, Assets, c_ws, 6, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),  
    skolem(countryName, Incorp, c_ws, 7, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp))  
) :- 'WorldcAF'(Name, FYEnd, Shares, Income, Sales, Assets, Incorp).

27
We first define a semantic relation for *Worldscope*. A semantic relation is then defined on the semantic objects in the corresponding relation attributes. The data elements derived from the extensional relation are mapped to semantic objects. These semantic objects define a unique object-id for each data element. In the example above each skolem term defines a unique semantic object corresponding to each attribute of the extensional relation. In addition to mapping each physical relation to a corresponding semantic object, we also define and initialize other relations defined in the domain model. The relations that come under this category are attribute and modifiers.

### 3.4 Mediation System

In the following sections, we will describe each mediation subsystem in somewhat more detail. We will use the application scenario we used earlier about the financial analyst trying to gather information about Daimler Benz Corp. We will start with the query, and then we will describe the application as it is programmed, explaining domain and how the context information for various sources is specified. Then we will follow the query as it passes through each subsystem.

We know the query that we are going to ask the system is:

```sql
SELECT worldscope.total_assets, datastream.total_sales, disclosure.net_income, quotes.Last
FROM worldscope, datastream, disclosure, quotes
WHERE worldscope.company_name = 'DAIMLER-BENZ AG' AND
datastream.as_of_date = '01/05/94' AND
worldscope.company_name = datastreama.name AND
worldscope.company_name = disclosure.company_name AND
worldscope.company_name = quotes.cname;
```

We will be asking the query in the *Worldscope* context. What this means is that once the mediation system has processed all parts of the query, the system then converts the result into the *Worldscope* context and returns the converted results back to the user.
3.4.1 SQL to Datalog Query Compiler

The SQL query as it is passed in to the system, is fed to the SQL query compiler. The query compiler takes in the SQL query and first parses the SQL into its corresponding datalog form while at the same time using the elevation axioms it elevates the data sources into its corresponding elevated data objects. The corresponding datalog query for the SQL query above is:

\[
\text{answer(total_assets, total_sales, net_income, last) :-}
\]
\[
\text{WorldcAF_p(V27, V26, V25, V24, V23, V22, V21),}
\]
\[
\text{DiscAF_p(V20, V19, V18, V17, V16, V15, V14),}
\]
\[
\text{DStreamAF_p(V13, V12, V11, V10, V9, V8),}
\]
\[
\text{quotes_p(V7, q_last),}
\]
\[
\text{Value(V27, c_ws, V5),}
\]
\[
\text{V5 = "DAIMLER-BENZ AG"},
\]
\[
\text{Value(V13, c_ws, V4),}
\]
\[
\text{V4 = "01/05/94"},
\]
\[
\text{Value(V12, c_ws, V3),}
\]
\[
\text{V5 = V3},
\]
\[
\text{Value(V20, c_ws, V2),}
\]
\[
\text{V5 = V2},
\]
\[
\text{Value(V7, c_ws, V1),}
\]
\[
\text{V5 = V1},
\]
\[
\text{Value(V22, c_ws, total_assets),}
\]
\[
\text{Value(V17, c_ws, total_sales),}
\]
\[
\text{Value(V11, c_ws, net_income),}
\]
\[
\text{Value(q_last, c_ws, last).}
\]

As can be seen, the query now contains elevated data sources along with a set of predicates that map each attribute to its value in the corresponding context. Since the user asked the query in the Worldscope context (denoted by c_ws), the last four predicates in the translated query ascertain that the actual values returned as the solution of the query need to be in the Worldscope context. The resulting unmediated datalog query is then fed to the mediation engine.
3.4.2 Mediation Engine

The mediation engine in some respects is the heart of the system. This is the part of the system that detects and resolves possible semantic conflicts. In essence, the mediation is a query rewriting process. The actual mechanism of mediation is based on an abduction engine [13]. The engine takes a datalog query and a set of domain model axioms and computes a set of abducted queries such that the abducted queries have all the differences resolved. The system does that by incrementally testing for potential semantic conflicts and introducing conversion functions for the resolution of those conflicts. The mediation engine as its output produces a set of queries that take into account all the possible cases given the various conflicts. Using the same example from above, if the mediation engine were fed the datalog query output by the SQL compiler, along with the domain model and contexts stated above, we would get the following set of abducted queries:

answer(V108, V107, V106, V105) :-
    datexform(V104, "European Style -", "01/05/94",
              "American Style /"),
    Name_map_Dt_Ws(V103, "DAIMLER-BENZ AG"),
    Name_map_Ds_Ws(V102, "DAIMLER-BENZ AG"),
    Ticker_Lookup2("DAIMLER-BENZ AG", V101, V100),
    WorldcAF("DAIMLER-BENZ AG", V99, V98, V97, V96,
             V108, V95),
    DiscAF(V102, V94, V93, V92, V91, V90, V89),
    V107 is V92 * 0.001,
    Currencytypes(V89, USD),
    DStreamAF(V104, V103, V106, V88, V87, V86),
    Currency_map(USD, V86),
    quotes(V101, V105).

answer(V85, V84, V83, V82) :-
    datexform(V81, "European Style -", "01/05/94",
              "American Style /"),
    Name_map_Dt_Ws(V80, "DAIMLER-BENZ AG"),
Name_map_Ds_Ws(V79, "DAIMLER-BENZ AG"),
Ticker_Lookup2("DAIMLER-BENZ AG", V78, V77),
WorldcAF("DAIMLER-BENZ AG", V76, V75, V74, V73, V85, V72),
DiscAF(V79, V71, V70, V69, V68, V67, V66),
V84 is V69 * 0.001,
Currencytypes(V66, USD),
DStreamAF(V81, V80, V65, V64, V63, V62),
Currency_map(V61, V62),
<>(V61, USD),
datexform(V60, "European Style /", "01/05/94", "American Style /")},
olsen(V61, USD, V59, V60),
V83 is V65 * V59,
quotes(V78, V82).

answer(V58, V57, V56, V55) :-
  datexform(V54, "European Style -", "01/05/94", "American Style -"),
  Name_map_Dt_Ws(V53, "DAIMLER-BENZ AG"),
  Name_map_Ds_Ws(V52, "DAIMLER-BENZ AG"),
  Ticker_Lookup2("DAIMLER-BENZ AG", V51, V50),
  WorldcAF("DAIMLER-BENZ AG", V49, V48, V47, V46, V58, V45),
  DiscAF(V52, V44, V43, V42, V41, V40, V39),
  V38 is V42 * 0.001,
  Currencytypes(V39, V37),
 <>(V37, USD),
  datexform(V36, "European Style /", V44, "American Style /"),
  olsen(V37, USD, V35, V36),
  V57 is V38 * V35,
  DStreamAF(V54, V53, V56, V34, V33, V32),
  Currency_map(USD, V32),
As it can be seen in the result produced above by the mediation engine, the mediated query contains four sub-queries. Each of the above sub-queries accounts for a potential semantic conflict. For example, the first sub-query deals with the case when there is no currency conversion conflict. While the second sub-query is pretty much the same query except that now the second query takes into account the possibility of currency conversion. Resolving the conflicts like the above currency conflict may sometime require introducing
intermediate data sources. Figure 4 lists some of the context differences in the various data sources that we use for our example. Looking at the table, we observe that some one of the possible conflicts is different data sources using different currencies to represent the financial data that they contain. In order to resolve that difference, the mediation engine has to introduce an intermediary data source. The source used for this purpose is a currency conversion web site (http://www.oanda.com) and is referred to as olsen. The currency conflict is resolved in the second sub-query where the olsen source is used to convert the currency to correctly represent data in the currency specified in the specified context.

### 3.4.3 Query Planner and optimizer

The query planner module takes the set of datalog queries produced by the mediation engine and produces a query plan. It ensures that an executable plan exists which will produce a result that satisfies the initial query. This is necessitated by the fact that there are some sources that impose restrictions on the type of queries that they can service. In particular, some sources may require that some of the attributes must always be bounded while making queries to those sources. Another limitation sources might have is the kinds of operators that they can handle. Possible example are some web sources that do not always export an interface that does not support all the SQL operators, or they might sometime require that some attributes in queries made to them be always bound. Once the planner ensures than an executable plan exists, it generates a set of constraints on the order in which the different sub-queries can be executed. Under these constraints, the optimizer applies standard optimization heuristics to generate the query execution plan.

The query execution plan is an algebraic operator tree in which each operation is represented by a node. There are two types of nodes:

- **Access Nodes**: Access nodes represent access to remote data sources. Two subtypes of access nodes are:
  - *sfw Nodes*: These nodes represent access to data sources that do not require input bindings from other sources in the query.
• **join-sfw Node:** These nodes have a dependency in that they require input from other data sources in the query. Thus these nodes have to come after the nodes that they depend on while traversing the query plan tree.

• **Local Nodes:** These nodes represent local operations in local execution engine. There are four subtypes of local nodes.
  - **Join Node:** Joins two trees
  - **Select Node:** This node is used to apply conditions to intermediate results.
  - **CVT Node:** This node is used to apply conversion functions to intermediate query result.
  - **Union Node:** This node represents a union of the results obtained by executing the sub-nodes.

Each node carries additional information about what data-source to access (if it is an access node) and other information that is used by the runtime engine. Some of the information that is carried in each node is a list of attributes in the source and their relative position, list of condition operations and any literals and other information like the conversion formula in the case of a conversion node. The simplified query plan for the first sub-query of the mediated query has been included below:

```
SELECT
  ProjL: [att(1, 5), att(1, 4), att(1, 3), att(1, 2)]
CondL: [att(1, 1) = "01/05/94"]
JOIN-SFW-NODE DateXform
  ProjL: [att(1, 3), att(2, 4), att(2, 3), att(2, 2), att(2, 1)]
CondL: [att(1, 1) = att(2, 5), att(1, 2) = "European Style -", att(1, 4) = "American Style /"]
CVT-NODE 'V18' is 'V17' * 0.001
  ProjL: [att(2, 1), att(1, 1), att(2, 2), att(2, 3), att(2, 4)]
CondL: ['V17' = att(2, 5)]
JOIN-SFW-NODE quotes
  ProjL: [att(2, 1), att(2, 2), att(1, 2), att(2, 3), att(2, 1)]
```


att(2, 4)]
CondL: [att(1, 1) = att(2, 5)]

JOIN-NODE
ProjL: [att(2, 1), att(2, 2), att(2, 3), att(2, 4),
att(2, 5)]
CondL: [att(1, 1) = att(2, 6)]

SELECT
ProjL: [att(1, 2)]
CondL: [att(1, 1) = 'USD']

SFW-NODE Currency_map
ProjL: [att(1, 1), att(1, 2)]
CondL: []

JOIN-NODE
ProjL: [att(2, 1), att(1, 2), att(1, 3), att(2, 2),
att(2, 3), att(1, 4)]
CondL: [att(1, 1) = att(2, 4)]

SFW-NODE Datastream
ProjL: [att(1, 2), att(1, 3), att(1, 1), att(1, 6)]
CondL: []

JOIN-NODE
ProjL: [att(2, 1), att(2, 2), att(2, 3), att(2, 4)]
CondL: [att(1, 1) = att(2, 5)]

SELECT
ProjL: [att(1, 2)]
CondL: [att(1, 1) = 'USD']

SFW-NODE Currencytypes
ProjL: [att(1, 2), att(1, 1)]
CondL: []

JOIN-NODE
ProjL: [att(2, 1), att(1, 2), att(2, 2), att(2, 3),
att(2, 4), att(2, 5)]
CondL: [att(1, 1) = att(2, 6)]

SELECT
ProjL: [att(1, 2)]
CondL: [att(1, 1) = 'USD']

SFW-NODE Currencytypes
ProjL: [att(1, 2), att(1, 1)]
CondL: []

JOIN-NODE
ProjL: [att(2, 1), att(1, 2), att(2, 2), att(2, 3),
att(2, 4), att(2, 5)]
CondL: [att(1, 1) = att(2, 6)]

SELECT
ProjL: [att(1, 2)]
CondL: [att(1, 1) = 'USD']

SFW-NODE Currencytypes
ProjL: [att(1, 2), att(1, 1)]
CondL: []
att(1, 3)]
CondL: [att(1, 1) = att(2, 4)]

SFW-NODE Disclosure
ProjL: [att(1, 1), att(1, 4), att(1, 7)]
CondL: []

JOIN-NODE
ProjL: [att(1, 1), att(2, 1), att(2, 2), att(2, 3)]
CondL: []

SELECT
ProjL: [att(1, 2)]
CondL: [att(1, 1) = "DAIMLER-BENZ AG"]

SFW-NODE Worldscope
ProjL: [att(1, 1), att(1, 6)]
CondL: []

JOIN-NODE
ProjL: [att(1, 1), att(2, 1), att(2, 2)]
CondL: []

SELECT
ProjL: [att(1, 2)]
CondL: [att(1, 1) = "DAIMLER-BENZ AG"]

SFW-NODE Ticker.Lookup2
ProjL: [att(1, 1), att(1, 2)]
CondL: []

JOIN-NODE
ProjL: [att(2, 1), att(1, 1)]
CondL: []

SELECT
ProjL: [att(1, 2)]
CondL: [att(1, 1) = "DAIMLER-BENZ AG"]

SFW-NODE Name_map_Ds_WS
ProjL: [att(1, 2), att(1, 1)]
The query plan that is produced by the planner is then forwarded to the runtime engine.

3.4.4 Runtime engine

The runtime execution engine executes the query plan. Given a query plan, the execution engine simply traverses the query plan tree in a depth-first manner starting from the root node. At each node, it computes the sub-trees for that node and then applies the operation specified for that node. For each sub-tree the engine recursively descends down the tree until it encounters an access node. For that access node, it composes a SQL query and sends it off to the remote source. The results of that query are then stored in the local store. Once all the sub-trees have been executed and the all the results available in the local store, then the operation associated with that node is executed and the results collected. This operation continues until the root of the query is reach. At this point the execution engine has the required set of results corresponding to the original query. These results are then sent back to the user and the whole process is completed.

3.5 Web Wrapper

The original query used in our example, among other sources, contained access to a quote server to get the most recent quotes for the company in question, i.e. Daimler-Benz. As opposed to the most of the other sources, the quote server that we used is a web quote service that can be accessed by anyone on the net. In order to access the web sources such as this one, we have developed a technology that lets users treat web sites as relational data sources. The users then issue SQL queries just as they would to any relation in the a
relational database, thus combining multiple sources and creating queries as the one above. This technology is called web wrapping and we have an implementation for this technology that we call web wrapping engine [18]. Using the web wrapper engine (web wrapper for short) the application developers can very rapidly wrap a structured or semi-structured web site and export the schema for the users to query against.
3.5.1 Web wrapper architecture

Figure 5 shows the architecture of the web wrapper. The system takes the SQL query as input. It parses the query along with the specifications for the given web site. A query plan is then constituted. The query plan contains a detailed list of web sites to send HTTP requests, the order of those requests and also the list of documents that will be fetched from those web sites. The executioner then executes the plan. Once the pages are fetched,
the executioner then extracts the required information from the pages and presents the collated results to the user.

3.5.2 Wrapping a web site

For our query, the relation *quote* is a web quote server that we access using our web wrapper. In this section we will briefly talk about how a web site is wrapped. In order to *wrap* a site, you need to create a specification file. This file is plain text file and contains information like the exported schema, the URL of the web site to access and a regular expression that will be used to extract the actual information from the web page. In our example we use the *CNN* quote server to get quotes. A simplified specification file is included below:

```plaintext
#HEADER
#RELATION=quotes
#HREF=GET http://qs.cnnfn.com
#EXPORT= quotes.Cname quotes.Last
#ENDHEADER
#BODY
#PAGE
#HREF = POST http://qs.cnnfn.com/cgibin/stockquote? \\
    symbols=##quotes.Cname##
#CONTENT=last:&nbsp \\
    </font><FONTSIZE=+1><b>##quotes.Last##</FONT></TD>
#ENDPAGE
#ENDBODY
```

The specification has two parts, *Header* and *Body*. The *Header* part specifies information about the name of the relation and the exported schema. In the above case, the schema that we decided to export has two attributes, *Cname*, the ticket of the company and *Last* the latest quote. The *Body* Portion of the file specifies how to actually access the page (as defined in the *HREF* field) and what regular expression to use (as defined in the *CONTENT* field).
Once the specification file is written and placed where the web wrapper can read it, we are ready to use the system. We can start making queries against the new relation that we just created.
Chapter 4

4. Building an Application

In this chapter we talk about how to build an application for the COIN system. We use the scenario built in the previous chapter of a stock analyst doing research on some security. She has four data sources from where she explicitly needs to access information. The stock analyst, as a user, is unaware of the fact that there are cases when the system accesses data sources implicitly without her ever specifying them in her query. An example is the currency exchange data source. The user never specifies if or when or even what data source the system should use. In fact she is not even aware of the fact that a currency exchange data source is used to convert currencies. This is handled by the system. When the system detects a conflict that involves currency conversion, the system automatically calls the intermediary currency conversion data source to resolve the conflicts. While the user is unaware of these intermediary data sources, the application developer needs to be aware of these sources and needs to program the system to handle those conflicts.

The process of developing a COIN application can be broken down into various steps.

- Pre-application setup.
- Domain model definition
- Identification of the sources.
- Source and Relation Definition.
- Elevation of the data sources.
- Addition of integrity constraints.
- Context definition for each data source.
- Addition of conversion functions.
- Application generation.
Once all the above steps are completed and the program debugged, we are ready to run the system. In the following pages, we will discuss each step in detail as we build a COIN application using the example scenario from of a stock analyst gathering information for her report that we mentioned in the previous chapter.

4.1 Pre-application Setup

In this stage we set up the structure where we are going to build the application. The first thing that we do is come up with a unique name for the application. In our case, we choose to call the application *tasc*. Once we have decided on a unique name, we create a new directory where we will place all the application files. The name of the directory, by convention is the same as the name of the application. We need to make sure that this directory has enough disk space to hold all the application files plus the application binaries created by the system. In addition to the above, during query processing, the system uses this directory to keep the intermediate results. The exact size requirement varies from application to application and the type of query and the size of data set being queried, a reasonable size is upwards of ten megabytes.

One more things that needs to be done as part of pre-application setup is copying the file "basics.chr" from the examples directory into the current directory in which the application is being built. The file "basics.char" contains basic rules for constraints that are applicable to all applications. For a detailed discussion on integrity constraints, please look at section 4.6. Once the file is copied into the newly created directory, the pre-application setup is completed and we can move to the next step.

As we walk through the application definition process, we create a set of files that contains all the application definition code. However, if the user needs to define some additional prolog code that they want to use in the system, they can define that in the file *userdefined.coin*. The code defined in the file userdefined.coin is becomes part of the system. It is accessible to the user-application and is executed at run-time.
4.2 Domain Model Definition

Before we start writing a new application, we need to define the domain model that will be used for that application. Figure 6 describes the domain model used in the example application that we are going to build. We call this financial domain model since the stock analyst is trying to gather financial information about a company. In the figure below, we have two types of objects, basic objects (denoted by squares) and semantic objects (denoted by ellipses). Basic objects have scalar types. Examples are numbers and strings.

The semantic objects are the ones that are used in the system as they carry rich semantic information about the objects in a given domain. In addition, there are two types of relations specified between the objects. These are:

- Attribute Relation
- Modifier Relation

These relations are explained in Chapter 4. One thing to note here is that in Chapter 4, in addition to the two types of relations mentioned above, we had a third type of relation called inheritance relation. That relation does not show up during application development phase as this relation is more of a conceptual relation and is handled by higher-level design tools. By the time we reach at the level of application development we are at, the inheritance relationship is flattened out by the higher level tools.

![Figure 6: Financial Domain Model](image)

Using Figure 6 as a guide, we define the domain model for the COIN system. In a file called `dm.coin`, we define each object and declare its type. In addition to the object
definition and type declaration, we define all the relations for each object. To demonstrate how to define an object and create the relations for that object, we will use the example of the semantic type *companyFinancials* as an example. The particular object has two modifiers, *company* and *fyEnding* and two modifiers, *scaleFactor* and *currency*. We first need to define *companyFinancials* semantic type. The construct to define a semantic object is:

\[
\text{semanticType}(\text{companyFinancials}).
\]

Once we have defined the object, we then go ahead and define other relations. The attributes and modifiers for *companyFinancials* are defined as:

\[
\text{attributes}(\text{companyFinancials}, \{\text{company}, \text{fyEnding}\}). \quad \text{Modifiers}(\text{companyFinancials}, \{\text{scaleFactor}, \text{currency}\}).
\]

The complete listing of the file *dm.coin* for the domain model is included in Appendix A.

### 4.3 Identification of the Sources and Relations

This section explains how we identify the sources. There are two types of relations:
- **Imported Relations**
- **Exported Relations**

Imported relations are accessed through the network and they usually correspond to the data sources from external sources. Exported relations are the sources that visible to the users and can be explicitly referenced by the users in their queries. Exported source can either be imported relations or views defined on other imported relations. For the above example, we notice that we have four data sources that the application explicitly refers to. All these sources need to be defined as exported relations in the application. However, on looking closer at the data sources, we notice that not all data sources define their currency amounts in the same currency. As an example, *Worldscope* data source always has the currency of the financial amounts in US Dollars. *Datastream*, on the other hand, lists the amount in the currency of the country that the security was incorporated in. The assets of
Daimler-Benz Corporation would be listed in US Dollars in *Worldscope* data source, but in *Datastream* data source, they would be listed in German Marks as the company is incorporated in Germany. The assets of IBM (an American company) on the other hand would be listed in US Dollars both in *Datastream* and *Worldscope*. To resolve this conflict, we need to bring a data source that can perform currency conversion. To help us solve our problem we use Olsen currency converter [22]. This is a web site that offers historical information on currency conversion rates for almost all the traded currencies. This source however will stay transparent to the user, as they will issue the query without having to explicitly mention the currency conversion. The system will automatically detect the currency conflicts based on the context definitions of the data sources in question and perform the conversion as needed. The *olsen* data source, in this particular example, was used as in import relation while processing the query even though the user did not explicitly mentioned the source in the query. But in this particular case, we decided to specify *olsen* as both import and export relation thus allowing the users to explicitly query the *olsen* source if they want to. This was purely a choice made by us as the application developer. Another example of a purely import data source is Name to ticker converter. In the Quote data source, the company name is specified in the form of the ticker symbol representing the common ticker for that company. The user however need not know about the ticker name, all they want to do is just ask the query and give the company name. For this case, we use a relation that can perform the conversion. The list below lists all the data sources, both export and import that we need for our example and a brief explanation of what each data source is meant for.

- Disclosure—Import/Export data source.
- Worldscope—Import/Export data source.
- Datastream—Import/Export data source.
- Quotes—Import/Export data source
- Olsen—Import/Export data source, needed to do currency conversion.
- Ticker_Lookup2—Import, performs company name to ticker symbol conversion and vice-versa.
• Name_map_Ds_Ws—Import, performs company name conversions between the Disclosure and Worldscope data sources and vice-versa.

• Name_map Dt_Ds—Import, performs company name conversions between the Datastream and Disclosure data sources and vice-versa.

• Name_map Dt_Ws—Import, performs company name conversions between the Datastream and Worldscope data sources and vice-versa.

• Currency_map—Import, given a country name, returns the currency for that country and vice-versa.

• Currencytypes—Import, converts two letter currency codes to three letter codes and vice-versa (e.g., US to USD).

• DateXform—Import, converts date from one format to another (e.g., mm/dd/yy to dd-mm-yy).

4.4 Source and Relation Definition

Once all the data sources have been identified, the next step involves actually defining the data sources and the relations that each data source provided. This is achieved in two steps. The first step is to create an entry for each data source and specify how to actually access that data source. The second step involves creating a relation in the COIN system and linking it to the previously created link for the actual location of the physical source that we just created. The above definitions are placed in file called sources.coin. We will use the example of Quotes data source and show how to first define the actual source in the database and then how to create a relation based on the actual source. The complete code of sources.coin for inserting all the sources and their corresponding relation is included in Appendix B.

4.4.1 Inserting the Data Sources

As we described in the previous section, the first step that we need to perform is to define the Quote source. The general clause that this used for this purpose is of the form:

:- source/3
A source as defined here, does not refer to a data source as described in the text. While in the text, a data source refers to what would be a relation in the relational world, in the present context, the source refers to a gateway (or a server process as shown in figure 2.) An example would be the case where we want to access data sources stored in an oracle relational database. Although the oracle database contains more than one data source, all the data sources are accessed from the same server process named oracle in our definition file. Hence in order to access all the data sources that are stored in the oracle database, we will have to define that server process. The following part, taken from the definition file defines two data sources. The first statement defines the oracle data source and the second statement defines a web data source with the name tasc.

```prolog
:-source(oracle, db, 'http://context.mit.edu/cgi-bin/oracle-wrapper.cgi).
:-source(tasc, web, 'http://context.mit.edu/cgi-bin/prolog-wrapper.cgi).
```

The above clause takes three arguments. The first argument is the name of the source. The second is the type of the source. The type could be a relational database (e.g. oracle database) as it is in our example above, or it could be a web source (the argument would then be 'web'). The third argument is actually the path of the program that COIN would call when it wants to access that source. In the above example, the specified path points to a gateway program that in turn connects to the oracle database and allows for passing SQL queries and returns the results to the users of the gateway program.

### 4.4.2 Inserting the Relations

Once the sources have been defined, we need to actually create all the relations that actually correspond to the data source used in the application. This involves defining the relations, setting up all the attributes of the relations, and describing the capabilities of the relations. The clause that we use for that purpose is:
:-relation/5

The above clause takes five arguments. The first argument can either be the name of the source as it appears in the source clause or it can be a view based on the other relations already defined. If the relation that is being inserted is a view, then the keyword 'view' is passed as the first argument. The first argument specifies the external source the given relation will come from.

The second argument is the name of the relation as it will appear in the COIN system. This argument binds the external relation to the relation in the COIN system.

The third argument specifies the type of the relation being defined. A relation can be one of import (denoted by 'i'), export (denoted by 'e') or import/export (denoted by 'ie'). The type import means that the relation has been imported from an external source and that this relation can only be used by the system internally and is not exposed to the users. The export type refers to the relations that are exported to the users to be used in the user queries. An exported relation can either be relation directly imported from an external source or it could be a COIN view formed by already existing sources. As the last type, a relation can both be of type import and export. This happens when an external source is incorporated into the COIN system and then exported to the users in their queries as is. The major difference between the types 'e' and 'ie' is that the relations of the former type are mostly views based on other, existing relations.

The fourth argument is the schema of the relation. It is actually a list that contains a set of two-tuples. Each tuple corresponds to an attribute in the relation. The first part of the tuple is the attribute name and the second part of the tuple is the data type of that attribute. In our example, quote data source has two attributes, Cname, which is of data type 'string' and Last, which is of data type 'real'.
The last argument (also called capability record since it specifies the capabilities of the given relation) is actually a list of two elements each of which again a list. The first sub-list is a bit-vector that lists the direction of the attributes of the relation. A bit represents each attribute in the list, which is either 1 or 0. For that attributes that have the bit turned on (bit value is 1), the direction of those attribute direction is in (in other words, the attribute needs to be bound when making the query for that relation). Alternately for the attributes whose value is 0 in the bit vector, the direction of those attribute is out (the attribute is returned as a result from the query). The first sub-list can in turn contain sub-lists if the relation can be called with different direction for the attributes in different scenario. The second sub-list contains all the operators not supported by the relation.

This last argument is needed for cases where the relation that we are inserting is not a full relational source. A web source can be a good example. In our case, the quote source cannot perform relational operations like less than or greater than and so on. So for relations like these, we have to specify while inserting the relations what type of operations the relation supports. Now for the quote data source, the actual relation clause would look like this:

```
:-relation (tasc,
    quotes,
    ie,
    [['Cname', string], ['Last', real]],
    cap([[1,0]], ['<', '>', '<>', '>=', '<='])).
```

We see that the relation quote is accessed from the ‘tasc’ source and that the name of the relation is the same as it is in COIN. This relation is of type ‘ie’ which means that this relation is imported from an external source and also that this relation can be used by the users in their queries. The schema of this relation has two attributes. The first attribute is called Cname and is of the type ‘string’. The second attribute is called Last and is of the type ‘real’. The capability record for this relation indicates that the direction of the attribute Cname is in which means that while querying, this attribute needs to be supplied
by the user. The direction of the attribute Last is out (this is passed back as the result of
the query). In addition, the second part of the last argument states that the above relation
does not support any of the operators specified in the list.

One thing to note here is that even though the export relations have capability records, the
capability record for those sources is not really needed because the capability record only
makes sense for the sources that are of the type import. The whole purpose of the
capability records is to specify the capabilities of an external relation that is being imported
into the COIN system. As discussed above, for the sources that are not fully relational, the
capability records are used as hints for the COIN system to realize those limitations of the
sources being incorporated. However, for sources that are of type export, this does not
make sense, as those sources do not have any of the issues encountered in the case of
import sources. This observation is valid and the capability records for the export relations
are not really used. For the export sources, they are there just for uniformity.

4.5 Elevation of the Data Sources

At this point we have defined all the sources for the COIN system and have created the
COIN relations based on the physical relations. In this section we will look at how we can
use the relations to actually start building the application.

Once the domain has been specified, and data sources identified and defined, and we can
finally go ahead and perform the elevation of the data sources into the COIN system. We
call this stage elevation of the data sources, as each relation is elevated from its physical
interpretation into this new domain that the application is being written for. In this
process, each relation is moved from its physical interpretation into this higher, and
semantically rich framework of the chosen domain model where each relation all its
attributes are now a part of the defined domain. Each attribute of the relation in this new
domain is no longer a scalar entity, but an object that is semantically rich. After this step is
performed, all the data sources can then be used by the system to perform mediation in the
given domain.
In more concrete terms, for each relation that we defined in the previous sections, we create a corresponding relation defined in terms of the types created in the given domain model. All the definitions for the elevated data sources are defined in file elevation.coin. For our example, we will use the Worldscope relation and describe how that relation is elevated. This is a two step process. In the first step, we map the physical relations to the semantic types according to the domain and in the second step, we then create the new semantic relations.

4.5.1 Mapping the Physical Types to Semantic Types

Going back to the previous section, we know that the data source Worldscope has the following attributes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPANY_NAME</td>
<td>VARCHAR2(80)</td>
</tr>
<tr>
<td>LATEST_ANNUAL_FINANCIAL_DATE</td>
<td>VARCHAR2(10)</td>
</tr>
<tr>
<td>CURRENT_OUTSTANDING_SHARES</td>
<td>NUMBER</td>
</tr>
<tr>
<td>NET_INCOME</td>
<td>NUMBER</td>
</tr>
<tr>
<td>SALES</td>
<td>NUMBER</td>
</tr>
<tr>
<td>COUNTRY_OF_INCORP</td>
<td>VARCHAR2(40)</td>
</tr>
<tr>
<td>TOTAL_ASSETS</td>
<td>NUMBER</td>
</tr>
</tbody>
</table>

Now if we compare the physical attributes of the table with the domain model in figure 6, we see the following mapping:

<table>
<thead>
<tr>
<th>Domain Type</th>
<th>Physical Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompanyName</td>
<td>VARCHAR2(80)</td>
</tr>
<tr>
<td>Date</td>
<td>VARCHAR2(10)</td>
</tr>
<tr>
<td>Basic</td>
<td>NUMBER</td>
</tr>
<tr>
<td>CompanyFinancials</td>
<td>NUMBER</td>
</tr>
<tr>
<td>CompanyFinancials</td>
<td>NUMBER</td>
</tr>
<tr>
<td>CompanyFinancials</td>
<td>VARCHAR2(40)</td>
</tr>
<tr>
<td>CountryName</td>
<td>NUMBER</td>
</tr>
</tbody>
</table>
The type basic in domain definitions corresponds to basic types like integers, reals or strings.

4.5.2 Creating the Semantic Relations

Once we have mapped the types, we are ready to create the relations. Creating the semantic relations is somewhat similar to creating the actual relations in COIN with a few important differences. Since each attribute of the semantic object corresponds to a relation between two semantic objects, we will define the attributes using a clause that links the two objects in that attribute relation. We will see how this is done shortly. In addition to that, we will have to specify the definitions of all the attribute relations that each semantic type needs to have separately. From the example above, and also by observing the domain model, we observe that for each companyFinancial semantic type, we will have to specify all the attributes for the type companyFinancial. In this case, we will have to specify the company and fyEnding for each object of type companyFinancial. Now without further ado, we will list the listing of elevated relation for Worldscope data source. The complete listing of the elevated data sources and relations is included in the file elevation.coin and can be found in Appendix C.

'WorldcAF_p'(
    skolem(companyName, Name, c_ws, 1,
        'WorldcAF'( Name, FYEnd, Shares, Income,
            Sales, Assets, Incorp)),
    skolem(date, FYEnd, c_ws, 2,
        'WorldcAF'( Name, FYEnd, Shares, Income,
            Sales, Assets, Incorp)),
    skolem(basic, Shares, c_ws, 3,
        'WorldcAF'( Name, FYEnd, Shares, Income,
            Sales, Assets, Incorp)),
    skolem(companyFinancials, Income, c_ws, 4,
        'WorldcAF'( Name, FYEnd, Shares,
            Income, Sales, Assets, Incorp)),

skolem(companyFinancials, Sales, c_ws, 5, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),
skolem(companyFinancials, Assets, c_ws, 6, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),
skolem(countryName, Incorp, c_ws, 7, 'WorldcAF'( Name, FYEnd, Shares, Income, Sales, Assets, Incorp))
)
:- 'WorldcAF'(Name, FYEnd, Shares, Income, Sales, Assets, Incorp).

attr(X,countryIncorp,Y) :- 'WorldcAF_p'(X,_,_,_,_,_,Y).
attr(X,company,Y) :- 'WorldcAF_p'(Y,_,_,X,_,_,_).
attr(X,fyEnding,Y) :- 'WorldcAF_p'(_,Y,_,X,_,_,_).

attr(X,company,Y) :- 'WorldcAF_p'(Y,_,_,_,X,_,_).
attr(X,fyEnding,Y) :- 'WorldcAF_p'(_,Y,_,_,Y,_,_).
attr(X,company,Y) :- 'WorldcAF_p'(Y,_,_,_,_,X,_,_).
attr(X,fyEnding,Y) :- 'WorldcAF_p'(_,Y,_,_,_,X,_,_).

attr(X,officialCurrency,Y) :- 'Currencytypes_p'(Z,Y), 'WorldcAF_p'(_,_,_,_,_,_,X),
value(X, c_ws, U),
value(Z, c_ws, U).

The relation for Worldscope is named WorldcAF_p and defined as a statement that has seven arguments. Each argument corresponds to a skolem term and defines an attribute of the Worldscope relation. Since the original Worldscope relation had seven attributes when
it was imported, we have seven skolem terms each corresponding to an attribute of the
original relation. Each of these clauses is of the standard term called skolem term. The
general form of a skolem term is as follows:

\[
\text{skolem}(\text{Type}, \text{Object}, \text{Context}, \text{position}, \text{source-relation})
\]

The first argument of the term is the type of the semantic object corresponding to that
attribute. The second argument is the actual COIN object corresponding to that attribute.
The third argument is the name of the context the object resides in. We will define the
actual contexts later. The fourth argument is the absolute position of the physical type in
the actual COIN relation corresponding to the given semantic type. The final argument is
the COIN relation itself listed in its entirety.

The second part of the definition involves defining the attribute relations corresponding to
each semantic type that requires them as defined in the domain model. Looking back at the
example again along with the domain model, we see that the type \textit{companyName} requires
an attribute of type \textit{countryIncorp}. We define that attribute by the following statement:

\[
\text{attr}(X, \text{countryIncorp}, Y) : \text{Worl}dc\text{AF}_p(X, _, _, _, _, Y).
\]

The above states that the attribute relation named \textit{countryIncorp} for the object \(X\) is \(Y\),
where \(X\) is the first argument of the relation \textit{Worldscope} which is the object of type
\textit{companyName} and \(Y\) is the last argument of the semantic relation \textit{Worldscope}, which
happens to be of type \textit{countryName}. Similarly for all the objects of the type
\textit{companyFinancials}, we specify their corresponding attributes in order to complete the
semantic type network as specified in the domain model.

4.5.3 Creating Views to Ease Application Development

While building an application, sometimes it may be convenient to define a view based on
already existing sources and work with that view in the application. Views can make an
application code clean and easy to maintain and also give a more organized structure to the whole application and if defined properly, make writing parts of application extremely simple. While views are not required while building an application, if it is at all possible, views should be created and taken advantage of. Here I will use one example view, name_map. This view is based on the observation that in each of the three Worldscope, Datastream and Disclosure data sources, each source had the same company name specified in a slightly different format. In order to resolve the differences, we introduced a set of implicit data sources that performed conversions. These tables were:

- Name_map_Dt_Ds(DtName, DsName)
- Name_map_Ds_Ws(DsName, WsName)
- Name_map_Dt_Ws(DtName, WsName)

We can now define a view on top of them and call it name_map. This view has the following definition:

Name_map(SrcVal, SrcCtx, TgtCtx, TgtVal).

This view maps the SrcVal as specified in SrcCtx to TgtVal as specified in TgtCtx. The values SrcCtx and TgtCtx are actually the names of the contexts that refer to the respective data sources. We then define the view as following. Here we list a few cases. The complete listing of the view is in the file elevation.coin, which is included as Appendix C.

name_map(Val, "ds_name", "ws_name", V) :-
    'Name_map_Ds_Ws'(Val, V).

name_map(Val, "ws_name", "ds_name", V) :-
    'Name_map_Ds_Ws'(V, Val).

name_map(Val, "dt_name", "ds_name", V) :-
    'Name_map_Dt_Ds'(Val, V).
name_map(Val, "ds_name", "dt_name", V) :-
    'Name_map_Dt_Ds'(V, Val).

Another scenario where a view can come in handy is when we want to escape to the underlying prolog system. In a situation like this, we create a view that is based on a prolog system function. An example of that is `currentDate` relation. This relation gets the date from the underlying prolog system, formats it and returns it to the COIN engine. The code for that view and also the elevated source is defined below.

```prolog
currentDate_p (skolem(date, V, c_ws, 1, currentDate(V))) :-
currentDate(V).

currentDate(Date) :-
    {date(D),
     substring(D, 5, 3, Month),
     substring(D, 9, 2, Day),
     substring(D, 23, 2, Year)},
    month(Month, NumMonth),
    {concat_string([NumMonth, "/", Day, "/", Year], Date)}.
```

In this case, the prolog code is evaluated at the mediation time when the user makes a query that requires an access to the above data source.

### 4.6 Addition of Integrity Constraints

In order to maintain the integrity of the data, we define and enforce a set of integrity constraints on the relations that we define [23]. For this purpose we use ECLiPSe's integrity constraints package that allows easy definitions of constraints. For the above quote data source, we want to define the notion of unique key which in our case is the Cname. The constraint, as it is written, is as follows:

```prolog
Quotes_icl @ '?quotes(X, Q1), '?quotes(X, Q2) #=> Q1=Q2.
```
The above is stating that if there are two rows from the quotes data source and the first attribute (Cname) of both the rows is the same, then it is necessary that the second attribute must match. This constraint ensures the uniqueness of the rows. While the above integrity constraint is used for specifying functional dependencies, integrity constraints can be used to provide any additional information the user might have about the data sources. The more information the user can provide about the system, the more it will help the system make informed decisions and creating better queries for that data sources. As an example, consider the case of a relation R that has two attributes, company name, and annual revenue. If we know that all the companies in this relation have revenue larger than 25 Million, we can specify that as the following constraint.

\[
R_{icl} \ @ \ ? \ R(name, \ revenue) \ \rightarrow \ revenue > 25000000
\]

Once the above constraint is specified, this will help the mediation system generate optimal queries. In the above case, the system will not access the relation R if the revenue is less than 25000000, thus saving an unnecessary source access.

All the constraints are written in the file ic.coin, which is included in its entirety as Appendix D.

### 4.7 Context definition for Each Data Source

Once we have elevated all the data sources into the domain model that we are going to work in, we can go ahead and start defining the contexts local to each of the data sources and then creating the contexts for each one of the data sources in COIN.

#### 4.7.1 Recognizing the Semantic Differences

Let us look at the table describing various semantic differences between the data sources once again.
Each row corresponds to a data source for which we have to specify a context in the above table and lists all the characteristics of that data source. We also need to give a unique name to each context that we are going to create for each of the data sources. For illustration, we will be using the *Worldscope* data source, and we will name \( c_{-}ws \) as the name of the context for the above data source. We will define the above context name in the domain model definition file along with the definition of the rest of the contexts. The context is defined as follows:

\[
\text{context}(c_{-}ws) .
\]

The declaration of the contexts is done in the file *dm.coin*. The complete listing of the context declaration for all the data sources is included as Appendix A.

Once we have specified and named the contexts, we need to go ahead and create the actual contexts. For that, we look at the context table again. Looking at the row for the *Worldscope* data source, each column of the table refers to a semantic difference present in the source that needs to be resolved. The column scale factor, for example, refers to the way financial amounts are stored in the give data sources. For example in the *Worldscope* data source, in order to get the actual amount for assets of a company, one will have to multiply the result returned from the data source by 1000. The currency column refers to
the type of currency the financial amounts are stored in. In Worldscope all the amounts are
stored in US dollars, while in Datastream data source, the amounts are stored in the
currency of the country the company was incorporated in. Date format refers to the way
dates are represented in the respective data source. “American /” means that the months
are stored first and the months, days and years are delimited by a “/”. An example would
be “8/28/1970”. Similarly “European -” means the days are stored first and the months,
days and years are delimited by a “-”. An example of the above format would be “28-8-
1970”. Currency type refers to the code used by the data source. Some data source use
two character currency codes (e.g. US for US dollars and FF for French Francs) while
other data sources use three character data sources (e.g. USD for US Dollars and FFR for
French Francs). The last column refers to the format the data source stores the company
name in. Each data source stores the names of the companies in slightly different manner.
An example is Daimler Benz. Worldscope stores the name as DAIMLER-BENZ AG.
Disclosure chooses to store it as DAIMLER BENZ CORP and Datastream data source
remembers it as DAIMLER-BENZ. They are all the name of the same company, however
unless it is known in advance exactly how the name is stored in a given data source, the
query might falsely return no results.

4.7.2 Creating the Contexts

Once we have recognized the potential semantic conflicts, creating the contexts is pretty
straightforward. We just need to code the context information and define all the contexts
in the file context.coin. We will use the example of Worldscope data source and explain
how to write the context for that data source. The complete listing of context.coin that
contains all the contexts is included as Appendix E. Corresponding to each column in
Figure 4, we write a corresponding prolog statement. The context for Worldscope data
source is as follows:

modifier(companyFinancials, 0, scaleFactor, c_ws, M):-
cste(basic, M, c_ws, 1000).
modifier(companyFinancials, O, currency, c_ws, M):-  
cste(currencyType, M, c_ws, "USD").

modifier(date, O, dateFmt, c_ws, M):-  
cste(basic, M, c_ws, "American Style /").

modifier(currencyType, O, curTypeSym, c_ws, M):-  
cste(basic, M, c_ws, "3char").

modifier(companyName, O, format, c_ws, M):-  
cste(basic, M, c_ws, "ws_name").

In the above example, each statement refers to a potential conflict that needs to be resolved by the system. Alternately, each statement refers to a column in the context table shown in Figure 7. Yet another way to look at the above is that each statement corresponds to a modifier relation in the actual domain model. From the domain model listed in Figure 6, we notice that the object CompanyFinancials has two modifiers, scaleFactor and currency. Correspondingly, the first two statements define these two modifiers. Looking at the context table in Figure 7, we notice that the value of the scaleFactor in the Worldscope context is 1000. The first statement represents that fact. It is stating that the modifier scaleFactor for the object O of type companyFinancials in the context c_ws is the object M where (the second line) the object M is a constant (cste) of type basic and has a value of 1000 in the context c_ws. Similarly, the last statement explained in plain English is as follows. The modifier format for the object O of type companyName in the context c_ws is the object M where (the second line) the object M is a constant (cste) of type basic and has a value of ws_name in the context c_ws. As explained earlier, the type basic refers to all the scalar types, numbers, reals and strings.

4.8 Addition of the Conversion Functions

The last step before we are finally finished with the application is to write conversion functions for the contexts. The conversion functions are used during context mediation
when the system needs to convert objects between different contexts. The context writer needs to provide the conversion functions for all the modifiers of all the objects defined in the domain by including them in the file `cvt.coin`. In our example, all the columns of the table directly correspond to the modifiers in the domain model. We will use the example of the `companyFinancials` object from the above domain model. The file `cvt.coin` that contains complete set of conversion functions for all the modifiers is listed in Appendix F. We observe from the domain model that the `companyFinancials` object has two modifiers, `scaleFactor` and `currency`. Hence we will need to write two conversion functions for the `companyFinancials` object. The first function will help us perform the conversion of `scaleFactor` and we will use the second function to perform the conversion for the `currency`.

For semantic objects like `companyFinancials` that have more than one modifier and thus require more than one conversion functions, the system assumes that the conversion functions for each of these modifiers are orthogonal. In other words, the functions do not depend on one another and are both commutative and associate. The reason we have this requirement is that since the system pretty much independently picks up the conversion functions for each object and builds a list, a specific order cannot be guaranteed. The system just creates a list of conversion functions and performs all the functions in the list one by one on the semantic object for which the functions have been specified.

Below is the conversion function for the `scaleFactor` modifier.

```
cvt(companyFinancials, _O, scaleFactor, Ctxt, Mvs, Vs, Mvt, Vt) :-
    Ratio is Mvs / Mvt,
    Vt is Vs * Ratio.
```

The above function states that in order to perform conversion of the modifier `scaleFactor` for the object `_O` of semantic type `companyFinancials` in the context `Ctxt` where the modifier value in source is `Mvs` and the object `_O`'s value in the source context is `Vs` and the modifier value in the target context is `Mvt` and the object `_O`'s value in the target
context is \( V_t \), we first find out the \textit{Ratio} between the modifier value in the source context and the modifier value in the target context. We then determine the Object's value in the target context by multiplying its value in the source context with the \textit{Ratio}. \( V_t \) now contains the appropriately scaled value for the object \( _O \) in the target context. One thing that should be mentioned is that if the modifier value is the same in both the source context and the target context, the system is smart enough to detect that and does not invoke the conversion function for that modifier.

One thing to notice in the above example for the scale conversion is that the variable \( Ctxt \) is the target context and itself is a variable. What this means is that the above conversion function will work in all the contexts. If for some reason, the user wants to define a conversion function for a modifier that is specific to a particular context, they will need to specialize that conversion function and write a separate conversion function for each of the contexts that exist in the current application.

The code for performing currency conversion is also worth looking at since we need to use an auxiliary data source (\textit{olsen}) for performing currency conversion. Since this conversion is different from the straightforward conversion as was the case in the above example, this conversion function is a little more complex. We will list the whole function and then explain it.

\begin{verbatim}
cvt(companyFinancials, O, currency, Ctxt, Mvs, Vs, Mvt, Vt) :-
    attr(O, fyEnding, FyDate),
    value(FyDate, Ctxt, DateValue),
    olsen_p(Fc, Tc, Rate, TxnDate),
    value(Fc, Ctxt, Mvs),
    value(Tc, Ctxt, Mvt),
    value(TxnDate, Ctxt, DateValue),
    value(Rate, Ctxt, Rv),
    Vt is Vs * Rv.
\end{verbatim}
In this case, the problem we have is that the modifier currency for the object `companyFinancials` points to another semantic object whose value depends on an external variable exchange rate and is thus not a static value or a constant. Hence in the above function, we are using the `olsen` source to get the exchange rate and then use that rate to do the actual conversion. In order to call `olsen`, we need additional information like the date for which we want the conversion, the currency from which we are converting and the currency to which are converting. We get the date by using the `fyEnding` attribute of the object `O` that is of type `companyFinancials` and has the attribute that we need to perform this type of conversion. We get the source currency type by looking up the value of the modifier in the source, which happens to be the currency type in the source context. The target currency comes from the modifier value in the target context. Once we have all the information we need in order to call the `olsen` source, we make the call and get the conversion rate in the variable `Rate`. Once we have the rate, we finally calculate the value of the object in the target context by multiplying the value of the object in the source object after we have looked up the value of `Rate` in the desired context.

4.9 Application Generation

Once we are finished with all the above tasks, we are ready to generate the application. At this stage we need to run the mint compiler that takes in all the application files and generates the application. The program that compiles the application source files and generates a binary is called `mintc` and takes one argument that is the full path of the directory, which contains all the application files. The mint compiler is called from the command line, and generates an executable that is the newly created application or an error if there was an error and the application could not be created. For our example, the call to the mint compile would like this.

```
mintc $DEV_DIR/tasc
```
where $\text{DEV\_DIR}$ is the path of the directory where the rest of the coin system resides. After we run the mint compiler and generate the application, we are finished with the application and it is ready to run.
Chapter 5

5. COIN System: Present and Future

In this chapter, I will discuss some system design aspects of the COIN. I will start by a brief description of exactly how the COIN server is implemented.

5.1 The Current System

The current implementation of COIN backend is written entirely in Prolog [20]. That includes the SQL-to-datalog compiler, abduction engine, query planner and optimizer, and multi-database executioner. The prolog engine used by the COIN is called ECLiPSe [21]. ECLiPSe is a prolog engine with built in constraint logic capabilities and can be downloaded from the net for free. ECLiPSe runs on UNIX platforms. The current version of COIN uses UNIX sockets protocols. The server binds to a port, listens for incoming requests on that specified ports and serves the incoming requests. The requests are in the form of SQL statements in plain text. The design is very simple and is shown in the figure below.

![Figure 8: COIN Driver](image)

There is a driver module that handles the communication to and from the port. The driver module waits until it gets a request from the net. Once the request arrives, it collects the request from the port in a buffer, calls the appropriate sub modules in the correct order, gets back the results in another buffer and sends the results back to where the request came from. A pseudo code of the algorithm used by the driver is shown below:
Driver() {
    While (TRUE) {
        ReadFromSocket(buffer);
        SQL = Get_sql(buffer);
        Context = Get_context(buffer);
        Datalog = sql2datalog(SQL, context);
        Mediated = MediateQuery(Datalog, context);
        Plan = Generate_plan(Mediated);
        Result = multidb_execute(Mediated);
        WriteToSocket(Result);
    }
}

5.2 Analysis of the Design

In this section, I will talk about a few of the classic software design tradeoffs and how they apply to COIN system. The hope is that by the end of this section we will have a good sense of which of those tradeoffs we should make.

5.2.1 How simple is too simple?

As we can see from the above code, the interface to the system is very simple and straightforward. The server just takes a SQL query and sends the results back along with the trace. There is no confusion as to what flags the server expects or how the results will get passed back. It is all very simple. All the burden of processing and filtering the result is placed on the client. If the client does not want to show the trace, it can strip away the unnecessary information and just show the results. If the client does not want to get the actual results of the query, but instead just wants a query plan, again, they still send the SQL query to the server, get all the results back and simply throw away the parts that are not needed. This approach is very simple and works beautifully. The downside with this approach is that it is too simple. It has no flexibility. One can think of many cases where
the user might not want to perform all the stages of context mediation. Sometimes for a very complex query, the user might just want to look at the mediated query and not the results. In other cases, the user might want to proceed in stages, just performing the mediation in the first stage caching the results and later on, calling the system again, passing the already mediated query to get the results. This can save the users a lot of unnecessary waste of time having to do things over and over again when in fact it is not necessary.

5.2.2 Tight Integration at the cost of flexibility

The other thing that jumps out of the current design is its very tight integration of subsystems. If we go back and take a look at the original simplified system driver, we will immediately notice that all the subsystems are implemented as calls to procedures. Since all the subsystems are implemented in prolog, all of the subsystems are called one after another as plain procedure calls and the output of one subsystem is passed in to the input of the next subsystem. This type of tight integration was ideal in the case of COIN since all the subsystems were implemented in prolog and there was no issue of interface conflict or the argument type mismatch. However, this can very quickly become an issue when the systems start to get large and the number of implementation languages becomes greater than one. Using the example of COIN, we notice that the inter-subsystem communication is carried out in the form of prolog data structures. As an example, the query planner and optimizer subsystem, as its output produces a query plan. This plan is in the form of a prolog data structure. This output from the query planner then acts as an input to the multi-database executioner. Now in our case, since both the subsystems are implemented in prolog, there is no problem and both the subsystems talk to each other in perfect harmony. Now imagine if you will that one of the subsystems, the multi-database engine for example was written in some other language, say Java. All of sudden we would have a problem. Here we have, our query planner subsystem creating wonderful and fast query plans and our execution engine cannot use them because the query planner is generating the plans as a prolog data structure, which our Java execution engine cannot execute. This particular example of a Java execution engine is not just an example. There is work going
on in the group to produce an execution engine that is not written in prolog, but in Java. So in all likelihood we might have to face this situation sooner than later.

5.2.3 Modularity—Enough Said

One of the most fundamental and probably the most important factor in designing large software systems is how to design modular systems. As the systems grow large, modularity is used to control complexity and the overall system coherency. Large systems are designed to be a collection of subsystem that come together to become whole. Each subsystem in turn is broken down into further sub-subsystems and this process continues until we are down to reasonable sized, manageable subsystems that have a well-defined boundary. Splitting a system up into smaller pieces is easy. The hard part is to split it up such that each subsystem is a coherent system in itself with a well-defined boundary. As the system is decomposed into smaller subsystems, each subsystem needs to specify an interface through which that particular subsystem can be used. Interface design is another very important task that needs to be performed very carefully. Since the subsystems communicate with each other only via interfaces and do not know anything else about each other, a subsystem is only as useful as its interface. If a subsystem has the most impressive features but they are not exposed properly via the interface, they might as well not exist. Ideal interface is not too narrow (just one procedure call for example) or too wide (hundred and hundreds of unnecessary calls). It has to have the right width. For precisely that reason good interface design is sometimes considered an art form.

The COIN system is well architected with the system subdivided into three main subsystems. The current architecture has served the system well given the requirements. But now as the requirements change, some changes will need to be made to the system. We visited the issue of subsystem interoperability in the previous section. Here we will look at it a little more closely. The current COIN subsystems have a procedural interface where the inputs and the outputs are mostly in the form of prolog data structures. These interfaces, even though they being clean and tidy, are strongly interdependent and are very strongly tied to each other. These types of systems, while easy to develop, can very
quickly become hard to maintain if developers start using other technologies as noted in
the previous section. In addition to that, the other more important details are whether it
makes sense to make each subsystem into its own server process and thus allow more
distributed and quite possibly more robust deployment of the server.

5.3 Suggested Design Modifications

In this section, I list two sets of proposals that can be used to improve the system
architecture such that it improves both the usability and the architecture of the system. The
first suggestion immediately increases the capabilities by redesigning the driver of the
COIN system. The second proposal in reality does not add any new features to the
system. Instead, it deals with formalizing the interfaces between the subsystems and
making the system truly modular. This will help later when developers are experimenting
with the system, these changes will make it very easy for them to just take out one
subsystem and slide in a newer, better version of the subsystem. The new subsystem does
not have to be in the same language or even the same process space.

5.3.1 More Flexible Interface to the COIN Server

In this section, I will talk about the proposed modifications to the driver module. In
section 5.1, I showed the architecture of the driver module and we saw that the only way
to communicate with it was to simply send the SQL query and the context and the driver
sent back the results after calling the appropriate COIN modules. In this architecture,
there is no flexibility if the user does not want to perform all the steps of mediation. I now
propose a new model where the user has more control over what parts the system gets
called. This gives the user ability to call the driver and tell it to perform just mediation for
example, or pass in a query plan and just tell it to execute the plan, or alternately do the
whole query. The new driver takes one more argument stage, and based on that argument,
calls the appropriate modules and sends back the results. The values stage can take along
with their meanings are listed below.

- All – Perform all the mediation steps from SQL compilation to execution.
- Compile – Just do the SQL to datalog compilation.
• Mediate – Just perform Mediation. The input is compiled datalog query.
• Plan – Just generate a query plan. The input is a mediated query plan.
• Execute – Execute the query. The input is a query plan produced by the planner.

The new modified architecture is listed below.

```
Network Access

Driver

COIN server
```

**Figure 9: Modified COIN Driver**

The pseudo code for the modified driver is listed below.

```
Driver() {
While (TRUE) {
    ReadFromSocket(buffer);
    Result = Get_request(buffer);
    Context = Get_context(buffer);
    Stage = Get_STAGE(buffer);
    If (Includes(Stage, Compile))
        Result = sql2datalog(Result, context);
    If (Includes(Stage, Mediate))
        Result = MediateQuery(Result, context);
    If (Includes(Stage, Plan))
        Result = Generate_plan(Result);
    If (Includes(Stage, Execute))
        Result = multidb_execute(Result);
    WriteToSocket(Result);
}
}
```

The new driver can now communicate with all the subsystems of the COIN server and depending on the user needs, the driver can just call the appropriate subsystems. This
flexibility goes a long way in scenarios where the user is either running a very complex query and she does not want to perform the whole mediation process in one go. She can just do piecewise mediation as and when she wants. Another scenario where this can flexibility can be extremely handy is for the users who have a static set of query that they run regularly. Using this new driver, they can run the system once and cache the query plan. From then onwards, they can just call the execution engine via the driver and pass in the cached query plan.

5.3.2 Formalizing the Interfaces

In the previous section, we talked about the top-level interface to the COIN server and how it could be modified to improve the functionality of the server. In this section, I will talk about some of the interfaces that exist between the COIN subsystems. As we noticed before, all the subsystems are written in prolog and they support interfaces that are based on prolog. For each subsystem, the input and the output is a prolog data structure. This approach works as long as all the subsystems are implemented in prolog, but breaks immediately if one of the subsystems is implemented in a language other than prolog. In order to fix this problem, we will need to loosen the existing level of integration and also introduce some standard format for the data such that different subsystem can easily communicate with each other even if they are implemented in different languages. The COIN system currently has a well-founded format of data as it gets passed between subsystems during the various stages of mediation. The SQL to Datalog compiler accepts a valid SQL string and produces a set of datalog statements that represent the original query in datalog. The Mediation subsystem takes the datalog query and produces another set of datalog statement that represent a query with all the conflicts resolved. The query planner takes the query in datalog and produces a query plan in the format that is discussed earlier. The execution engine takes the query plan and returns the results after executing the query. At each stage, we do have well-defined format for the input and output. The problem is that each of the subsystems currently accepts those inputs in the form of parsed prolog data structures. In order to address that problem, we need to modify the interfaces of the existing subsystems such that instead of taking the input as
prolog structures, they take the arguments as strings and then perform an extra step of parsing those inputs according to their internal data structures. Similarly when producing outputs, all the subsystems will need to convert their output into string which can then either be sent back to the user if they did not request the whole mediation or on to the next subsystem. A diagram of the modified COIN system is shown below.

**Figure 10: Modified COIN Server**

The normalizers perform the job of converting the arguments and results into standard format. The introduction of normalizers allows the flexibility of easily replacing an existing subsystem with a new version of compatible subsystem along with the normalizers for that subsystem and be able to run the COIN system with the new subsystem installed. It does not matter how the new subsystem is implemented or even what language the subsystem is implemented in. As long as the normalizer converts the input and the outputs in the specified format, we should be able to run the system. This allows us to collaborate with other research groups who might be geographically away. Various groups can conduct research independently regardless of the language of choice or the environment. Once some group has developed a new version of a subsystem and they want other researchers to test the new features in their newly developed subsystem, all they need to do is write the appropriate normalizers for that subsystem and make the new subsystem publicly available. The rest of the COIN community can then pick up the new subsystem, replace
their current subsystem with the new version and start testing the system with the new subsystem installed.

5.4 The new Architecture

The design of the COIN mediation discussed in section 5.1 is being modified to make it more robust and more modular. As explained in section 5.1, currently, the COIN server interacts with the clients through TCP/IP sockets. This method however requires that every time a client needs to process a query, the COIN server has to be manually started which then binds to a port and services the requests. Every time the COIN server crashes, the client processes lose the connection to the server and have to wait until the server process is started again manually. This approach is particularly impractical if the clients are accessing the server remotely and do not have ready access to the server administrator.

We have changed the old COIN architecture and modified the system to automate the server startup and rather than manually starting the server and waiting for the queries, start the server on demand. In the new architecture, the whole application is compiled into an executable that can be called from the operating system. This makes it very easy to write drivers that service client requests as they arrive. Every time a request arrives, the COIN system is started, application loaded, request serviced and the server shutdown. In the preliminary tests, we have discovered that the server startup and application load time is extremely fast and almost negligible. There is a possibility that the startup time might suffer for large application. However we believe this might not be the case, as when we start up the server, we are not compiling the application. We are loading the already compiled application. The application is compiled at application create time.

The top-level interface to the COIN system has also been modified to make it more flexible. We have enhanced the system such that now it is possible to query the system and rather than getting the query results all the time, now the clients can specify if they want
an intermediate result. For example, if the clients choose to just get a query plan and not run the multi-database execution engine, it is possible to do that now. Essentially, we have implemented the design we proposed in section 5.3.1. to allow the users to ask the queries in stages and thus the used do not have to always wait for the whole of the query to finish process if all they want is just the SQL-datalog conversion for example. The ability to call the COIN system in stage along with the automatic system startup and shutdown on demand has increased the utility of the significantly. This feature is especially useful where now people can use their favorite web browsers to ask a query to the COIN system and get back the result without having to first contact someone in the COIN group and get the COIN server starated.

As a demonstration of the new system, we are converting the existing tasc application that we build earlier in this dissertation to use the new architecture.

5.5 The Next Step

The area of context mediation is relatively young and currently a lot of research is being conducted into various aspects of the projects. From the software-engineering point of view, the COIN system is still in its nascent stages. As more scientists conduct research into the field of context mediation, the COIN system will only grow larger and more complex. There will be more subsystems, implemented in varied languages ranging from prolog to perl to Java. Some of that has already started to happen. There is work going on right now to develop an execution engine in Java. This is however, just the beginning. As time goes on, more and more such projects will surface. The COIN system design is already pretty robust and work is going on to make the COIN system even more modular. The challenge for the COIN however lies ahead. As the COIN system acquires more features and gets more complex, will the architecture scale? I am confident that it will.
Chapter 6

6. Conclusions

In this dissertation, I have described and explained the Context Interchange Network, a system that automatically resolves semantic differences between data sources by performing mediation based on the pre-defined contexts for the given data sources. I have explained the COIN system at a conceptual level describing the domain model the idea of resolving semantic differences using mediation. At the systems level, I have talked about various processes that comprise the whole COIN network and how the data flows in the network. At the architectural level, I have explained each subsystem and how they interact with each other. I have also included a thorough tutorial on how to develop applications for the COIN system. I hope that this dissertation will work as a guideline for researchers who follow me on how to use the system, how to build applications for the system and how to extend the system.

This however, is just the beginning. Now that a coherent set of documentation has been produced, I hope that more and more researchers come forward and start working on the system both at the system level and at the application level. Now that a tutorial on how to write an application has been written, I hope that the researches use that tutorial as a guideline to build other more useful applications. On the system side, there are many areas of research. There is a need for application development tools. We have purposely designed the syntax of the application files to be simple and plain. It is not impossible to create a high-level visual application development environment that helps users graphically design new domains and develop applications. The system could conceivably generate the application code which can then compiled to create applications. The opportunities are endless.
References


[18] Bressan, S., Bonnet, P. "Extraction and Integration of Data from Semi-structured Documents into Business Applications". To be published.


[23] Bressan, S., Goh, C. “Semantic Integration of Disparate Information sources over the Internet using Constraint Propagation”. To be Published.
Appendix A: Domain Model

%%--------------------------------------------------------
%% Domain model (dm.coin)
%%--------------------------------------------------------

semanticType(companyFinancials).
semanticType(companyName).
semanticType(date).
semanticType(currencyType).
semanticType(exchangeRate).
semanticType(countryName).
semanticType(basic).

modifiers(companyFinancials, [scaleFactor,currency]).
modifiers(companyName, [format]).
modifiers(date, [dateFmt]).
modifiers(currencyType, [curTypeSym]).
modifiers(exchangeRate, []).
modifiers(countryName, []).
modifiers(basic, []).

attributes(companyFinancials, [company,fyEnding]).
attributes(companyName, [countryIncorp]).
attributes(date, []).
attributes(currencyType, [curTypeSym]).
attributes(exchangeRate, [txnDate,fromCur,toCur]).
attributes(countryName, [officialCurrency]).
attributes(basic, []).

conext(c_ds).
conext(c_dt).
conext(c_ws).
conext(c_ya).
conext(c_ol).
Appendix B: Data Source Descriptions

%%----------------------------------------------------------
%% Domain model (sources.coin)
%%----------------------------------------------------------

%%
%% Source Address
%%

source(tasc, web,
    'http://context/-coin/demos/wrapper/cgi-bin/prolog-wrapper.cgi').

source(oracle, db,
    'http://context.mit.edu/-coin/demos/oracle-wrapper/cgi-bin/oracle-wraper.cgi').

source('DateXform', web,
    'http://context.mit.edu/cgi-bin/qbe-dev/client/datexform.cgi').

%%
%% Relations
%%

relation(view,
    countryIncorp,
    e,
    [['COMPANY_NAME',string],['COUNTRY',string]],
    cap([[0,0]], [[]]).

relation(oracle,
    'Currencytypes',
    i,
    [['COUNTRY',string],['CURRENCY',string]],
    cap([[0,0]], [[]]).

relation(oracle,
    'Currency_map',

i,
[
['CHAR3_CURRENCY', string],
['CHAR2_CURRENCY', string]
],
cap([[0, 0]], [])
).

relation('DateXform',
datexform,
i,
[
['Date1', string],
['Format1', string],
['Date2', string],
['Format2', string]
],
cap([[1, 1, 0, 1], [0, 1, 1, 1]])
['<', '>', '<', '>=']
).

relation(tasc,
olsen,
ie,
[['Exchanged', string],
['Expressed', string],
['Rate', real],
['Date', string]]
, cap([[1, 1, 0, 1]], ['<', '>', '<', '>=']
).

relation(tasc,
quotes,
ie,
[['Cname', string],
['Last', string]]
, cap([[1, 0]], ['<', '>', '<', '>=']
).

relation(oracle,
'DiscAF',
ie,
[['COMPANY_NAME', string],
['LATEST_ANNUAL_DATA', string],
['CURRENT_SHARES_OUTSTANDING', integer],
['NET_INCOME', integer],
['NET_SALES', integer],
['TOTAL_ASSETS', integer],
['LOCATION_OF_INCORP', string]]
, cap([[0, 0, 0, 0, 0, 0, 0]], [])
).

relation(oracle,
'WorldcAF',
ie,
[['COMPANY_NAME', string],
['LATEST_ANNUAL_FINANCIAL_DATE', string],
['LATEST_ANNUAL_DATA', string],
['CURRENT_SHARES_OUTSTANDING', integer],
['NET_INCOME', integer],
['NET_SALES', integer],
['TOTAL_ASSETS', integer],
['LOCATION_OF_INCORP', string]]
, cap([[0, 0, 0, 0, 0, 0, 0, 0, 0]], [])
).
["CURRENT_OUTSTANDING_SHARES",integer],
["NET_INCOME",integer],
["SALES",integer],
["TOTAL_ASSETS",integer],
["COUNTRY_OF_INCORP", string],
cap([[0,0,0,0,0,0]], [])
)
)

relation(oracle, 'WorldcAFT', ie,
[["COMPANY_NAME",string],
["LATEST_ANNUAL_FINANCIAL_DATE",string],
["CURRENT_OUTSTANDING_SHARES",integer],
["NET_INCOME",integer],["SALES",integer],
["TOTAL_ASSETS",integer],
["COUNTRY_OF_INCORP", string]],
cap([[0,0,0,0,0,0]], [])
)
)

relation(view, 'CNames', e, 
[["COMPANY_NAME",string]],
cap([[0]], []))
)

relation(oracle, 'Name_map_Ds_Ws', i,
[["DS_NAMES",string],"WS_NAMES",string]],
cap([[0,0]], []))
)

relation(oracle, 'Ticker_Lookup2', i,
[["COMP_NAME",string],"TICKER",string],
["EXC", string]],
cap([[0,0,0]], []))
)

relation(oracle, 'Name_map Dt_Ds', i,
[{"DT_NAMES",string},{"DS_NAMES",string}],
cap([[0,0]], [1]).

relation(oracle,
   'Name_map_Dt_Ws',
   i,
   [{"DT_NAMES",string},{"WS_NAMES",string}],
cap([[0,0]], [1]).

relation(oracle,
   'DStreamAF',
   ie,
   [{"AS_OF_DATE",string},{"NAME",string},
    "TOTAL_SALES",integer],
   [{"TOTAL_EXTRAORD_ITEMS_PRE_TAX",integer},
    "earned_for_ordinary",integer],
   {"CURRENCY",string}],
cap([[0,0,0,0,0]], [1])).
Appendix C: Elevation Axioms

%%%----------------------------------------------------------
%%% Elevation Axioms (elevation.coin)
%%%----------------------------------------------------------

%%%----------------------------------------------------
%%% Disclosure:DiscAF (Keys: CompanyName and Date)
%%%----------------------------------------------------

'DiscAF_p'(  
  skolem(companyName, Name, c_ds, 1, 'DiscAF'(Name, FYEnd, Shares, 
  Income, Sales, Assets, Incorp)),  
  skolem(date, FYEnd, c_ds, 2, 'DiscAF'(Name, FYEnd, Shares, Income, 
  Sales, Assets, Incorp)),  
  skolem(basic, Shares, c_ds, 3, 'DiscAF'(Name, FYEnd, Shares, 
  Income, Sales, Assets, Incorp)),  
  skolem(companyFinancials, Income, c_ds, 4, 'DiscAF'(Name, FYEnd, 
  Shares, Income, Sales, Assets, Incorp)),  
  skolem(basic, Sales, c_ds, 5, 'DiscAF'(Name, FYEnd, Shares, 
  Income, Sales, Assets, Incorp)),  
  skolem(companyFinancials, Assets, c_ds, 6, 'DiscAF'(Name, FYEnd, 
  Shares, Income, Sales, Assets, Incorp)),  
  skolem(countryName, Incorp, c_ds, 7, 'DiscAF'(Name, FYEnd, Shares, 
  Income, Sales, Assets, Incorp))  
) :- 'DiscAF'(Name, FYEnd, Shares, Income, Sales, Assets, Incorp).

attr(X,countryIncorp,Y) :- 'DiscAF_p'(X,_,_,_,_,_,Y).

attr(X,officialCurrency,Y) :- 'Currencytypes_p'(Z,Y),  
  'DiscAF_p'(_,_,_,_,_,_,X),  
  value(X, c_ds, U),  
  value(Z, c_ds, U).

attr(X,company,Y) :- 'DiscAF_p'(Y,_,_,X,_,_,_).

attr(X,fyEnding,Y) :- 'DiscAF_p'(_,Y,_,X,_,_,_).

attr(X,company,Y) :- 'DiscAF_p'(Y,_,_,_,X,_,).
attr(X,fyEnding,Y) :- 'DiscAF_p'(_,Y,_,_,_,X,_,).

%%-----------------------------------------------------------------
%% Worldscope:WorldcAF (Keys: CompanyName and Date)
%%-----------------------------------------------------------------

'WorldcAF_p'(
    skolem(companyName, Name, c_ws, 1, 'WorldcAF'( Name, FYEnd,
    Shares, Income, Sales, Assets, Incorp)),
    skolem(date, FYEnd, c_ws, 2, 'WorldcAF'( Name, FYEnd, Shares,
    Income, Sales, Assets, Incorp)),
    skolem(basic, Shares, c_ws, 3, 'WorldcAF'( Name, FYEnd, Shares,
    Income, Sales, Assets, Incorp)),
    skolem(companyFinancials, Income, c_ws, 4, 'WorldcAF'( Name,
    FYEnd, Shares, Income, Sales, Assets, Incorp)),
    skolem(companyFinancials, Sales, c_ws, 5, 'WorldcAF'( Name, FYEnd,
    Shares, Income, Sales, Assets, Incorp)),
    skolem(companyFinancials, Assets, c_ws, 6, 'WorldcAF'( Name, FYEnd,
    Shares, Income, Sales, Assets, Incorp)),
    skolem(countryName, Incorp, c_ws, 7, 'WorldcAF'( Name, FYEnd,
    Shares, Income, Sales, Assets, Incorp))
) :- 'WorldcAF'(Name, FYEnd, Shares, Income, Sales, Assets, Incorp).

attr(X,countryIncorp,Y) :- 'WorldcAF_p'(X,_,_,_,_,_,Y).

attr(X,company,Y) :- 'WorldcAF_p'(Y,_,_,_,_,_).

attr(X,fyEnding,Y) :- 'WorldcAF_p'(_,Y,_,_,_,_,_).

attr(X,company,Y) :- 'WorldcAF_p'(Y,_,_,_,_,_).

attr(X,fyEnding,Y) :- 'WorldcAF_p'(_,Y,_,_,_,_,_).

attr(X,company,Y) :- 'WorldcAF_p'(Y,_,_,_,_,_).

attr(X,fyEnding,Y) :- 'WorldcAF_p'(_,Y,_,_,_,_,_).

attr(X,officialCurrency,Y) :- 'Currencytypes_p'(Z, Y),

86
'WorldcAF_p'(_,_,_,_,_,_,X),
value(X, c_ws, U),
value(Z, c_ws, U).

%------------------------------------------------------------------
% Worldscope:WorldcAFT (Keys: CompanyName and Date) ! Historical
%------------------------------------------------------------------

'WorldcAFT_p'(
  skolem(companyName, Name, c_ws, 1, 'WorldcAFT' (Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),
  skolem(date, FYEnd, c_ws, 2, 'WorldcAFT' (Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),
  skolem(basic, Shares, c_ws, 3, 'WorldcAFT' (Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),
  skolem(companyFinancials, Income, c_ws, 4, 'WorldcAFT' (Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),
  skolem(companyFinancials, Sales, c_ws, 5, 'WorldcAFT' (Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),
  skolem(companyFinancials, Assets, c_ws, 6, 'WorldcAFT' (Name, FYEnd, Shares, Income, Sales, Assets, Incorp)),
  skolem(countryName, Incorp, c_ws, 7, 'WorldcAFT' (Name, FYEnd, Shares, Income, Sales, Assets, Incorp))
) :- 'WorldcAFT' (Name, FYEnd, Shares, Income, Sales, Assets, Incorp).

attr(X,countryIncorp,Y) :- 'WorldcAFT_p'(X,_,_,_,_,_,Y).

attr(X,company,Y) :- 'WorldcAFT_p'(Y,_,_,_,_,_,X,_,_).

attr(X,fyEnding,Y) :- 'WorldcAFT_p'(Y,_,_,_,_,_,X,_,_,_), currentDate_p(Y).

attr(X,company,Y) :- 'WorldcAFT_p'(Y,_,_,_,_,_,X,_,_).

attr(X,fyEnding,Y) :- 'WorldcAFT_p'(Y,_,_,_,_,_,X,_,_,_), currentDate_p(Y).

attr(X,company,Y) :- 'WorldcAFT_p'(Y,_,_,_,_,_,X,_,_).

attr(X,fyEnding,Y) :- 'WorldcAFT_p'(Y,_,_,_,_,_,X,_,_,_), currentDate_p(Y).

attr(X,officialCurrency,Y) :- 'Currencytypes_p'(Z,Y),
'WorldcAFT_p'(_,_,_,_,_,X),
value(X, c_ws, U),
value(Z, c_ws, U).

%%------------------------------------------------------------------
%% Datastream (Keys: Date and CompanyName)
%%------------------------------------------------------------------

'DStreamAF_p'( skolem(date, AsofDate, c_dt, 1, 'DStreamAF'( AsofDate, Name, 
TotSales, ItemsPretax, EarnedOrd, Currency)),
skolem(companyName, Name, c_dt, 2, 'DStreamAF'( AsofDate, Name, 
TotSales, ItemsPretax, EarnedOrd, Currency)),
skolem(companyFinancials, TotSales, c_dt, 3, 'DStreamAF'( 
AsofDate, Name, TotSales, ItemsPretax, EarnedOrd, Currency)),
skolem(companyFinancials, ItemsPretax, c_dt, 4, 'DStreamAF'( 
AsofDate, Name, TotSales, ItemsPretax, EarnedOrd, Currency)),
skolem(companyFinancials, EarnedOrd, c_dt, 5, 'DStreamAF'( 
AsofDate, Name, TotSales, ItemsPretax, EarnedOrd, Currency)),
skolem(currencyType, Currency, c_dt, 6, 'DStreamAF'( AsofDate, 
Name, TotSales, ItemsPretax, EarnedOrd, Currency)) ) :- 'DStreamAF'(AsofDate, Name, TotSales, ItemsPretax, EarnedOrd, Currency).

attr(X,company,Y) :- 'DStreamAF_p'(_,Y,X,_,_,_,).
attr(X,fyEnding,Y) :- 'DStreamAF_p'(Y,_,X,_,_,_,).
attr(X,company,Y) :- 'DStreamAF_p'(Y,_,_,X,_,_).
attr(X,fyEnding,Y) :- 'DStreamAF_p'(Y,,,,X,_,_).
attr(X,countryIncorp,Y) :- 'DStreamAF_p'(_,X,_,_,_,_,Z),
atr(X, officialCurrency, Y2) :- 'CurrencyTypes_p'(X,Y1),

attr(X,officialCurrency,Y2) :- 'CurrencyTypes_p'(X,Y1),
'DStreamAF_p'(.,_,_,_,_,Y2),
value(Y1, c_dt, Y),
value(Y2, c_dt, Y).

%%%------------------------------------------------------------------
%%% Olsen (Keys: Source Currency, Target Currency, and Date)
%%%------------------------------------------------------------------

olsen_p(
    skolem(currencyType, Exch, c_ol, 1, olsen(Exch, Express, Rate, Date)),
    skolem(currencyType, Express, c_ol, 2, olsen(Exch, Express, Rate, Date)),
    skolem(exchangeRate, Rate, c_ol, 3, olsen(Exch, Express, Rate, Date)),
    skolem(date, Date, c_ol, 4, olsen(Exch, Express, Rate, Date))
) :- olsen(Exch, Express, Rate, Date).

attr(X, txnDate, Y) :- olsen_p(_,_,X,Y).

attr(X, fromCur, Y) :- olsen_p(_,Y,X,_).

attr(X, toCur, Y) :- olsen_p(Y,_,X,_).

%%%------------------------------------------------------------------
%%% Generic Quotes (Keys: Name)
%%%------------------------------------------------------------------

quotes_p(
    skolem(companyName, Cname, c_ya, 1, quotes(Cname, Last)),
    skolem(basic, Last, c_ya, 2, quotes(Cname, Last))
) :- quotes(Cname, Last).

attr(X, countryIncorp, Y) :- quotes_p(X,_),
    'WorldcAF_p'(C,_,_,_,_,_,Y),
    value(X, c_ya, Y1),
    value(C, c_ya, Y1).

%%%------------------------------------------------------------------
%% TickerLookup2 (Keys: Name)

'Ticker_Lookup2_p'

  skolem(basic,Name,c_ws , 1, 'Ticker_Lookup2'(Name,Ticker, _)),
  skolem(basic,Ticker,c_ws , 2, 'Ticker_Lookup2'(Name,Ticker, _))

) :- 'Ticker_Lookup2'(Name,Ticker, _).

%%--------------------------------------------------------------

%% Currencytypes (Keys: Country Name and Official Currency Symbol)

'Currencytypes_p'

  skolem(countryName, Co, c_ds, 1, 'Currencytypes'(Co, Cu)),
  skolem(currencyType, Cu, c_ds, 2, 'Currencytypes'(Co, Cu))

) :- 'Currencytypes'(Co, Cu).

% attr(X, officialCurrency, Y) :- 'Currencytypes_p'(X,Y).
% the above causes a pb (e.g. see query 12 - big query -)

%%----------------------------------------------------------------

%% Currency_map (Keys: 3 Char. Currency Symbol, 2 Char. Currency Symbol)
%%---------------------------------------------------------------

'Currency_map_p'

  skolem(basic, Cu3, c_ds, 1, 'Currency_map'(Cu3, Cu2)),
  skolem(basic, Cu2, c_ds, 2, 'Currency_map'(Cu3, Cu2))

) :- 'Currency_map'(Cu3, Cu2).

%%----------------------------------------------------------------

%% Name_map_Ds_Ws (Keys: Disclosure Name and WorldScope Name)
%% This relationship is elevated in the source context c_ds

'Name_map_Ds_Ws_p'

  skolem(basic,Disc, c_ds, 1, 'Name_map_Ds_Ws'(Disc, WScope)),
  skolem(basic,WScope,c_ds,2, 'Name_map_Ds_Ws'(Disc, WScope))

) :- 'Name_map_Ds_Ws'(Disc, WScope).
%---------------------------------------------------------------
% Name_map_Dt_Ds (Keys: Datastream Name and Disclosure Name)
%---------------------------------------------------------------

'Name_map_St_Ds_p'(
    skolem(basic, DSt, c_dt, 1, 'Name_map_St_Ds'(DSt, Disc)),
    skolem(basic, Disc, c_dt, 2, 'Name_map_St_Ds'(DSt, Disc))
) :- 'Name_map_St_Ds'(DSt, Disc).

%---------------------------------------------------------------
% Name_map_Dt_Ws (Keys: Datastream Name and WorldScope Name)
%---------------------------------------------------------------

'Name_map_Dt_Ws_p'(
    skolem(basic, DSt, c_dt, 1, 'Name_map_Dt_Ws'(DSt, WScope)),
    skolem(basic, WScope, c_dt, 2, 'Name_map_Dt_Ws'(DSt, WScope))
) :- 'Name_map_Dt_Ws'(DSt, WScope).

%---------------------------------------------------------------
% Prolog view for the date (evaluated at abduction time)
%---------------------------------------------------------------

currentDate_p(skolem(date, V, c_ws, 1, currentDate(V))) :-
currentDate(V).

currentDate(Date) :-
    (date(D),
     substring(D, 5, 3, Month),
     substring(D, 9, 2, Day),
     substring(D, 23, 2, Year)),
     month(Month, NumMonth),
     {concat_string([NumMonth, "/", Day, "/", Year], Date)}.

month("Jan", '01').
month("Feb", '02').
month("Mar", '03').
month("Apr", '04').
month("May", '05').
month("Jun", '06').
month("Jul", '07').
month("Aug", '08').
month("Sep", '09').
month("Oct", '10').
month("Nov", '11').
month("Dec", '12').

%% Abduction time view for the name mapping

name_map(Val, "ds_name", "ws_name", V) :-
   'Name_map_Ds_Ws'(Val, V).

name_map(Val, "ws_name", "ds_name", V) :-
   'Name_map_Ds_Ws'(V, Val).

name_map(Val, "dt_name", "ds_name", V) :-
   'Name_map_Dt_Ds'(Val, V).

name_map(Val, "ds_name", "dt_name", V) :-
   'Name_map_Dt_Ds'(V, Val).

name_map(Val, "dt_name", "ws_name", V) :-
   'Name_map_Dt_Ws'(Val, V).

name_map(Val, "ws_name", "dt_name", V) :-
   'Name_map_Dt_Ws'(V, Val).

name_map(Val, "ws_name", "ya_name", V) :-
   'Ticker_Lookup2'(Val, V, _).

name_map(Val, "ya_name", "ws_name", V) :-
   'Ticker_Lookup2'(V, Val, _).

name_map(Val, "dt_name", "ya_name", V) :-
   'Name_map_Dt_Ws'(Val, Z),
   'Ticker_Lookup2'(Z, V, _).

name_map(Val, "ya_name", "dt_name", V) :-
'Ticker_Lookup2'(Z, Val, _),
    'Name_map Dt_Ws'(V, Z).

name_map(Val, "ds_name", "ya_name", V) :-
    'Name_map_Ds_Ws'(Val, Z),
    'Ticker_Lookup2'(Z, V, _).

name_map(Val, "ya_name", "ds_name", V) :-
    'Ticker_Lookup2'(Z, Val, _),
    'Name_map_Ds_Ws'(V, Z).

%-----------------------------------------------------------
% View for Names
%----------------------------------------
'CNames_p'(skolem(companyName, Name, c_ds, 1, 'CNames'(Name))):-
    'CNames'(Name).

'CNames'(Name):-'DiscAF'(Name, __, __, __, __).
'CNames'(Name):-'WorldcAF'(Name, __, __, __, __, __, __).
'CNames'(Name):-'DStreamAF'(__, Name, __, __, __, __).
Appendix D: Integrity Constraints

%%% Integrity Constraints (ic.coin)

%%

%% WorldcAF(COMPANY_NAME, LATEST_ANNUAL_FINANCIAL_DATE,
%% CURRENT_OUTSTANDING_SHARES, NET_INCOME, SALES, TOTAL_ASSETS,
%% COUNTRY_OF_INCORP).
%%
worldcaf_icl @ '??'WorldcAF'(X,A,B1,C1,D1,E1,F1),
'??'WorldcAF'(X,A,B2,C2,D2,E2,F2)
===> B1=B2,C1=C2,D1=D2,E1=E2,F1=F2.

%%

%% DiscAF(COMPANY_NAME, LATEST_ANNUAL_DATA, CURRENT_SHARES_OUTSTANDING,
%% NET_INCOME, NET_SALES, TOTAL_ASSETS, LOCATION_OF_INCORP).
%%
discaf_icl @ '??'DiscAF'(X,A,B,C,D,E,F), '??'DiscAF'(X,A,B1,C1,D1,E1,F1)
===> B=B1,C=C1,D=D1,E=E1,F=F1.

%%

%% DStreamAF(AS_OF_DATE, NAME, TOTAL_SALES,
%% EXTRA_ORDINARY_ITEMS_PRE_TAX,
%% EARNED_FOR_ORDINARY, CURRENCY).
%%
dstreamaf_icl @ '??'DStreamAF'(X,A,B,C,D,E),
'??'DStreamAF'(X,A,B1,C1,D1,E1)
===> B=B1,C=C1,D=D1,E=E1.

%%

%% Name_map_Ds_Ws(DS_NAMES, WS_NAMES).
%%
name_map_ds_ws_ic1 @ '??'Name_map_Ds_Ws'(X,Z1),
'??'Name_map_Ds_Ws'(X,Z2)
===> Z1=Z2.
name_map_ds_ws_ic2 @ '??'Name_map_Ds_Ws'(Z1,X),
'??'Name_map_Ds_Ws'(Z2,X)
%%
%% Name_map_Dt_Ds(DT_NAMES, DS_NAMES).
%%
name_map_dt_ds_ic1 @ '?''Name_map_Dt_Ds'(X,Z1),
'?''Name_map_Dt_Ds'(X,Z2)
  ==> Z1=Z2.
name_map_dt_ds_ic2 @ '?''Name_map_Dt_Ds'(Z1,X),
'?''Name_map_Dt_Ds'(Z2,X)
  ==> Z1=Z2.

%%
%% Name_map_Dt_Ws(DT_NAMES, WS_NAMES).
%%
name_map_dt_ws_ic1 @ '?''Name_map_Dt_Ws'(X,Z1),
'?''Name_map_Dt_Ws'(X,Z2)
  ==> Z1=Z2.
name_map_dt_ws_ic2 @ '?''Name_map_Dt_Ws'(Z1,X),
'?''Name_map_Dt_Ws'(Z2,X)
  ==> Z1=Z2.

%%
%% Currencytypes(COUNTRY, CURRENCY).
%%
currencytypes_ic1 @ '?''Currencytypes'(Co, Cu1),
  '?''Currencytypes'(Co, Cu2) ==> Cu1 = Cu2.
currencytypes_ic2 @ '?''Currencytypes'(Co1, Cu),
  '?''Currencytypes'(Co2, Cu) ==> Co1 = Co2.

%%
%% Currency_map(3CHAR_CURRENCY, 2CHAR_CURRENCY).
%%
currency_map_ic1 @ '?''Currency_map'(Cu1, Cu21),
  '?''Currency_map'(Cu1, Cu22) ==> Cu21 = Cu22.
currency_map_ic2 @ '?''Currency_map'(Cu11, Cu2),
  '?''Currency_map'(Cu12, Cu2) ==> Cu11 = Cu12.
% Olsen (Exchanged, Expressed, Rate, Date).
%
olsen.icl @ '?olsen(E1,E2,R1,D1), '?olsen(E1,E2,R2,D1) ==> R1=R2.
%
%------------------------------------------------------------------
% Generic Quotes (Keys: Ticker)
%-----------------------------------------------------------------
quotes.icl @ '?quotes(X,Q1), '?quotes(X,Q2) ==> Q1=Q2.
%
%------------------------------------------------------------------
% Ticker Lookup (Keys: Company Name)
%-----------------------------------------------------------------
ticker.lookup2.icl @ '?ticker_lookup2(X,T1,T3),
 '?ticker_lookup2(X,T2,T4) ==> T1=T2, T3=T4.
%
% ignore the exchange (assume unique)
ticker.lookup2.ic2 @ '?ticker_lookup2(T1,X,T3),
 '?ticker_lookup2(T2, X, T4) ==> T1=T2, T3=T4.
%
%
% Datexform (Date1, Fmt1, Date2, Fmt2).
%
datexform.icl @ '?datexform(Date, Fmt1, Date1, Fmt2),
 '?datexform(Date, Fmt1, Date2, Fmt2)
 ==> Date1 = Date2.
datexform.ic2 @ '?datexform(Date1, Fmt1, Date, Fmt2),
 '?datexform(Date2, Fmt1, Date, Fmt2)
 ==> Date1 = Date2.
datexform.ic3 @ '?datexform(Date, Fmt1, Date1, Fmt2),
 '?datexform(Date2, Fmt2, Date, Fmt1)
 ==> Date1 = Date2.
datexform.ic4 @ '?datexform(Date1, Fmt1, Date, Fmt2),
 '?datexform(Date, Fmt2, Date2, Fmt1)
 ==> Date1 = Date2.
Appendix E: Context Definitions

%---------------------------------------------------
%- TASC Contexts (context.coin)
%---------------------------------------------------

% disclosure context
modifier(companyFinancials, O, scaleFactor, c_ds, M):-
cste(basic, M, c_ds, 1).

modifier(companyFinancials, O, currency, c_ds, M):-
attr(O, company, Company),
attr(Company, countryIncorp, Country),
attr(Country, officialCurrency, M).

modifier(companyName, O, format, c_ds, M):-
cste(basic, M, c_ds, "ds_name").

modifier(date, O, dateFmt, c_ds, M):-
cste(basic, M, c_ds, "American Style /").

modifier(currencyType, O, curTypeSym, c_ds, M):-
cste(basic, M, c_ds, "3char").

% datastream context
modifier(companyFinancials, O, scaleFactor, c_dt, M):-
cste(basic, M, c_dt, 1000).

modifier(companyFinancials, O, currency, c_dt, M):-
attr(O, company, Company),
attr(Company, countryIncorp, Country),
attr(Country, officialCurrency, M).

modifier(companyName, O, format, c_dt, M):-
cste(basic, M, c_dt, "dt_name").

modifier(date, O, dateFmt, c_dt, M):-
cste(basic, M, c_dt, "European Style -").
modifier(currencyType, 0, curTypeSym, c_dt, M):-
cste(basic, M, c_dt, "2char").

% Olsen context
modifier(companyFinancials, 0, scaleFactor, c_ol, M):-
cste(basic, M, c_ol, 1).

modifier(companyFinancials, 0, currency, c_ol, M):-
attr(O, company, Company),
attr(Company, countryIncorp, Country),
attr(Country, officialCurrency, M).

modifier(companyName, 0, format, c_ol, M):-
cste(basic, M, c_ol, "ws_name").

modifier(date, 0, dateFmt, c_ol, M):-
cste(basic, M, c_ol, "European Style /").

modifier(currencyType, 0, curTypeSym, c_ol, M):-
cste(basic, M, c_ol, "3char").

% Worldscope context
modifier(companyFinancials, 0, scaleFactor, c_ws, M):-
cste(basic, M, c_ws, 1000).

modifier(companyFinancials, 0, currency, c_ws, M):-
cste(currencyType, M, c_ws, "USD").

modifier(companyName, 0, format, c_ws, M):-
cste(basic, M, c_ws, "ws_name").

modifier(date, 0, dateFmt, c_ws, M):-
cste(basic, M, c_ws, "American Style /").

modifier(currencyType, 0, curTypeSym, c_ws, M):-
cste(basic, M, c_ws, "3char").

% Yahoo context
modifier(companyFinancials, 0, scaleFactor, c_ya, M):-
cste(basic, M, c_ya, 1).
modifier(companyName, 0, format, c_ya, M):-
    cste(basic, M, c_ya, "ya_name").

modifier(date, 0, dateFmt, c_ya, M):-
    cste(basic, M, c_ya, "American Style /").

modifier(companyFinancials, 0, currency, c_ya, M):-
    cste(currencyType, M, c_ya, "USD").

modifier(currencyType, 0, curTypeSym, c_ya, M):-
    cste(basic, M, c_ds, "3char").
Appendix F: Conversion Functions

%---------------------------------------------------------------
% TASC conversion functions (cvt.coin)
%---------------------------------------------------------------
%
%---------------------------------------------------------------
% conversion functions for companyFinancials, wrt scaleFactors
%---------------------------------------------------------------
%
/*
cvt(Sem-Type, 
   Sem-Object, 
   Modifier name, 
   Context, 
   Modifier value in source, 
   Object intermediary value in source, 
   Modifier value in target, 
   Object intermediary value in target)
*/
%
%---------------------------------------------------------------
% conversion functions for companyFinancials, wrt scaleFactors
%---------------------------------------------------------------
% if scale factors are same nothing to do

cvt(companyFinancials, _O, scaleFactor, Ctxt, Mvs, Vs, Mvt, Vt) :-
   Ratio is Mvs / Mvt,
   Vt is Vs * Ratio.

%---------------------------------------------------------------
% conversion functions for companyFinancials, wrt currencies
%---------------------------------------------------------------

cvt(companyFinancials, O, currency, Ctxt, Mvs, Vs, Mvt, Vt) :
   attr(O, fyEnding, FyDate),
   value(FyDate, Ctxt, DateValue),
   olsen_p(Fc, Tc, Rate, TxnDate),
   value(Fc, Ctxt, Mvs),
value(Tc, Ctxt, Mvt),
value(TxnDate, Ctxt, DateValue),
value(Rate, Ctxt, Rv),
Vt is Vs * Rv.

%%-----------------------------------------------
%% conversion functions for companyName
%%-----------------------------------------------

cvt(companyName, _O, format, _Ctxt, Mvs, Vs, Mvt, Vt) :-
    name_map(Vs, Mvs, Mvt, Vt).

%%-----------------------------------------------
%% conversion functions for dateFmts
%%-----------------------------------------------

cvt(date, _O, dateFmt, _Ctxt, Mvs, Vs, Mvt, Vt) :-
    datexform(Vs, Mvs, Vt, Mvt).

%%-----------------------------------------------
%% conversion functions for currency symbols.
%%-----------------------------------------------

cvt(currencyType, _O, curTypeSym, _Ctxt, "3char", Vs, "2char", Vt) :-
    'Currency_map'('Vs', Vt).

cvt(currencyType, _O, curTypeSym, _Ctxt, "2char", Vs, "3char", Vt) :-
    'Currency_map'(Vt, Vs).