A Translation Algorithm to Solve Semantic Conflicts in Information Exchange

by

Nancy C. Gotta

S.B., Computer Science and Engineering (1997)
Massachusetts Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 1998

Copyright 1998 Nancy C. Gotta
All rights reserved

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part.

Signature of Author . . . .
Department of Electrical Engineering and Computer Science
May 22, 1998

Certified by . . . .
Stuart E. Madnick
John Norris Maguire Professor of Information Technology
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Theses
Department of Electrical Engineering and Computer Science

Eng.

To Jason

A Translation Algorithm to Solve Semantic Conflicts in Information Exchange

by

Nancy C. Gotta

Submitted to the Department of Electrical Engineering and Computer Science on May 22, 1998, in Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Semantic conflicts that arise in the exchange of data among different users and data sources can lead to the misinterpretation of facts and can potentially have devastating effects. The Context Interchange system was created to resolve such conflicts.

In this thesis, a translation system has been constructed specifically for use with Context Interchange. The translation system allows users to read existing user contexts and create new contexts using a set of editors. The system translates between English and Penny, the programming language currently used for the expression of contexts. By using this system, it is possible for users to read and construct contexts without having any knowledge of the Penny language.

The system is comprised of four editors and a translator. Each of the editors creates a specific component of the Penny context: the domain model, the context axioms, the conversion functions, and the elevation axioms. The translator transforms a context written in Penny into a plain English description of the context.

Thesis Supervisor: Stuart E. Madnick
Title: John Norris Maguire Professor of Information Technology, Sloan School of Management

Acknowledgments

I would like to acknowledge and thank Professor Stuart Madnick for the invaluable guidance he has given me over the course of the past year. I would also like to thank the entire Context Interchange research group for their guidance and assistance. I have gained substantial experience while working on this project.

I would like to thank my parents, Alexander and Colleen Gotta, for their advice, understanding, support, and love. I also thank my friends for their patience and encouragement.

Finally, I would like to thank my fiancé, Jason Hintersteiner. Despite a busy schedule, he found the time to read several drafts of this thesis. He has shown great patience, understanding, and love, and it is to him that I dedicate this thesis.

Biography

Nancy C. Gotta received a Bachelor of Science degree in Computer Science and Engineering from the Massachusetts Institute of Technology in June 1997. As an undergraduate, she did extensive coursework in the areas of software engineering, artificial intelligence, and data networking. She also worked on several projects as a software engineer. As a graduate student, she is continuing to pursue her current research interests in data networking. After finishing her Master's degree, she plans to work in computer technical consulting.

Table of Contents

1 **Introduction** 11

1.1 Background 11
1.2 Organization of Thesis 13

2 **The Context Interchange System** 15

2.1 The Overall System 15
2.2 Applications of COIN 17
2.3 The Structure of a COIN Context 19
2.3.1 Domain Model 20
2.3.2 Context Axioms 20
2.3.3 Conversion Functions 21
2.3.4 Elevation Axioms 21

3 **The Penny Language** 23

3.1 History of Penny 23
3.2 Syntax of Penny 24
3.2.1 Domain Model 24
3.2.2 Context Axioms 25
3.2.3 Conversion Functions 27
3.2.4 Elevation Axioms 28

4 **The Translating and Editing System** 31

4.1 Overview of the System 32
4.2 Major Components of the Translating and Editing System 34
4.2.1 The Penny to English Translator 34
4.2.2 The Domain Model Editor 37
4.2.3 The Context Axiom Editor 43
4.2.4 The Conversion Function Editor 45
4.2.5 The Elevation Axiom Editor 49

5 **Example of Use of Translating and Editing System** 53

5.1 Domain Model Editor 53
5.2 Context Axiom Editor 56
5.3 Conversion Function Editor 60
5.4 Elevation Axiom Editor 64
5.5 Penny to English Translator 67

6 **Conclusions and Future Work** 71

6.1 Summary of Contributions 71
6.2 Future Work 71
6.2.1 Natural Language Processing 71
6.2.2 Graphical Interface 72
6.2.3 Translation Directly to Datalog 73
6.3 Conclusions 73

A **Penny to English Translator Rules** 75

B **Sample Application Penny Code** 79

B.1 Domain Model 82
B.2 Context Axioms. 82
B.2.1 Disclosure (DiscAF). 82
B.2.2 Worldscope (WorldAF). 83
B.2.3 Datastream (DStreamAF). 83
B.2.4 Olsen 84
B.3 Conversion Functions. 84
B.4 Elevation Axioms. 86
B.4.1 Disclosure. 86
B.4.2 Worldscope 87
B.4.3 Olsen 88
B.4.4 Datastream 88
B.4.5 Auxiliary Elevation Axioms 89

C **List of Abbreviations** 91

Table of Figures and Tables

Chapter 2

Figure 2.1 Architecture of the COIN system 16
Figure 2.2 Architecture of the COIN mediator 17

Chapter 3

Figure 3.1 Domain model axiom 25
Figure 3.2 Context axioms 26
Figure 3.3 Intensional context axiom. 27
Figure 3.4 Conversion functions 27
Figure 3.5 Elevation axioms 28

Chapter 4

Figure 4.1 Main menu of translation system 33
Figure 4.2 Sample domain model 39
Figure 4.3 Initial domain model editor dialog and creation of first domain model axiom 41
Figure 4.4 Creation of second and third domain model axioms 42
Figure 4.5 Sample context axioms 44
Figure 4.6 User dialog for creation of context axioms 45
Figure 4.7 Sample conversion functions 47
Figure 4.8 Dialog for creation of simple conversion function 48
Figure 4.9 Dialog for creation of complex conversion function 49
Figure 4.10 Sample elevation axioms 50
Figure 4.11 Dialog for creation of elevation axioms 51

Chapter 5

Figure 5.1 User dialog for obtaining file name for domain model 54
Figure 5.2 Dialog for creation of domain model axiom 55
Figure 5.3 Penny domain model axiom . . . .
56
Figure 5.4 Initial user dialog for context axiom editor 56
Figure 5.5 First two lines of context axiom Penny code 57
Figure 5.6 Menu of semantic types 57
Figure 5.7 Menu of semantic type modifiers 58
Figure 5.8 User dialog for entering modifier value 58
Figure 5.9 Penny context axiom 58
Figure 5.10 Dialog for the creation of an intensionally defined modifier . . . 59
Figure 5.11 Final line of context axiom Penny code 60

Figure 5.12 User dialog for obtaining file name for conversion axioms. . . . 60
Figure 5.13 User dialog for obtaining domain model and context names . . 61
Figure 5.14 First two lines of conversion functions 61
Figure 5.15 Conversion function menu 61
Figure 5.16 User dialog for creating a conversion function where the source and target values are the same 62
Figure 5.17 A conversion function where source and target values are the same 62
Figure 5.18 User dialog for entering complex conversion functions 62
Figure 5.19 User dialog for addition of string and number conversion functions 63
Figure 5.20 String and number conversion functions 63
Figure 5.21 Final line of conversion functions 63
Figure 5.22 User dialog for obtaining file name for elevation axioms 64
Figure 5.23 Initial user dialog for elevation axiom editor 64
Figure 5.24 User dialog for entering semantic objects into elevation axioms. 65
Figure 5.25 First three lines of elevation axiom Penny code 66
Figure 5.26 User dialog for overriding value declarations. 66
Figure 5.27 Sample Penny code for translation 68
Figure 5.28 User dialog for obtaining file names for translator 69
Figure 5.29 Translator output 69

Appendix A

Table A.1 Rules considered regardless of value of *Contexttype* 75
Table A.2 Rules considered when *Contexttype* = “domain model” 76
Table A.3 Rules considered when *Contexttype* = “context axioms”. 76
Table A.4 Rules considered when *Contexttype* = “conversion functions”. . 77
Table A.5 Rules considered when *Contexttype* = “elevation axioms”. . . . 77

Appendix B

Table B.1 Axiom reference numbers for domain model and context axioms 80
Table B.2 Axiom reference numbers for conversion functions and elevation axioms 81

This page intentionally left blank

Chapter 1

Introduction

1.1 Background

In the past decade, there has been an unprecedented growth in the number of information sources and receivers around the world. Additionally, it has become much easier to transmit information globally using large-scale communications media such as the Internet. These factors have led to a vast increase in the quantity of information being exchanged on a daily basis throughout the world.

The capacity to transmit information in this manner has led to a considerable problem, however. While it is now possible to exchange information easily across the globe, this information is meaningless unless considered in its underlying context. A statement considered in the wrong context can lead to major problems of miscommunication.

A simple example of this problem is the representation of dates. The date 4-2-1997 would, in the United States, be interpreted as April 2, 1997. In the United Kingdom, on the other hand, it would be read as February 4, 1997.

Another example of this problem is the comparison of financial data in different currencies. There are almost as many currencies in use in the world today as there are countries, and the relative values of these currencies shift on a daily basis. One U.S.

dollar is a very different amount of money from one Japanese yen or one British pound sterling.

Conflicts such as these, where information interpreted in one context means something entirely different from the same information interpreted in another context, are referred to as semantic conflicts. If these conflicts are not detected and resolved, the information exchanged becomes meaningless.

The Context Interchange (COIN) strategy attempts to solve the problems of semantic conflicts by mediating data access. Semantic conflicts among information sources and receivers are not identified a priori. Rather, they are detected and reconciled by a context mediator, which compares the contexts of the information sources and receivers and translates the data accordingly[1].

A context is expressed as a set of assertions in a programming language known as Penny.[2] Penny is a deductive object-oriented language which allows the logic-based axioms and statements that comprise a context to be expressed.

Penny, however, can be very difficult to understand and use, especially for a person who is unfamiliar with the language and not accustomed to computer programming. Deciphering a context in Penny is not a trivial task. Determining what a particular variable refers to in a context is even more difficult, and creating new contexts can be a considerable challenge.

---

[1] S. Bressan, et. al., "The COntext INterchange Mediator Prototype," ACM SIGMOD International Conference on Management of Data, 1997.

[2] Fortunato Pena, "PENNY: A Programming Language and Compiler for the Context Interchange Project," M.Eng. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1997.

The goal of this thesis, therefore, is to create a system to translate Penny contexts into a format that is easier to understand. This system enables users who are unfamiliar with the Penny language and the complexities of context creation to read and understand existing contexts, and either to modify these contexts or to create new contexts to suit their needs.

1.2 Organization of Thesis

This thesis is organized into six chapters. Chapter 2 contains a detailed description of the COIN system and the context mediator at its heart. Chapter 3 discusses the Penny language, including its capabilities and syntax. Chapter 4 describes the translating and editing system, including both the translator used to express Penny contexts in an easy to understand form and the editors used to create the different parts of Penny contexts. Chapter 5 consists of a sample run through each of the editors and the translator. Finally, in Chapter 6, conclusions are presented along with suggestions for future research.

This page intentionally left blank

Chapter 2

The Context Interchange System

The Context Interchange (COIN) strategy attempts to solve the problems of semantic conflicts by mediating data access. Semantic conflicts among information sources and receivers are not identified *a priori*. Instead, these semantic conflicts are detected and reconciled by a context mediator, which compares the contexts of the information sources and receivers and translates the data accordingly. With the system in place, a user need never know that a semantic conflict ever occurred, as the conflict is detected and resolved by the context mediator.[1]

2.1 The Overall System

The COIN system consists of three main components: the user, the remote data source from which the user wishes to retrieve data, and the context mediator between them. Figure 2.1 illustrates these components.

---

[1] For more information on the COIN system, see: "The COntext INterchange Project," World Wide Web: *http://context.mit.edu/~coin/*.

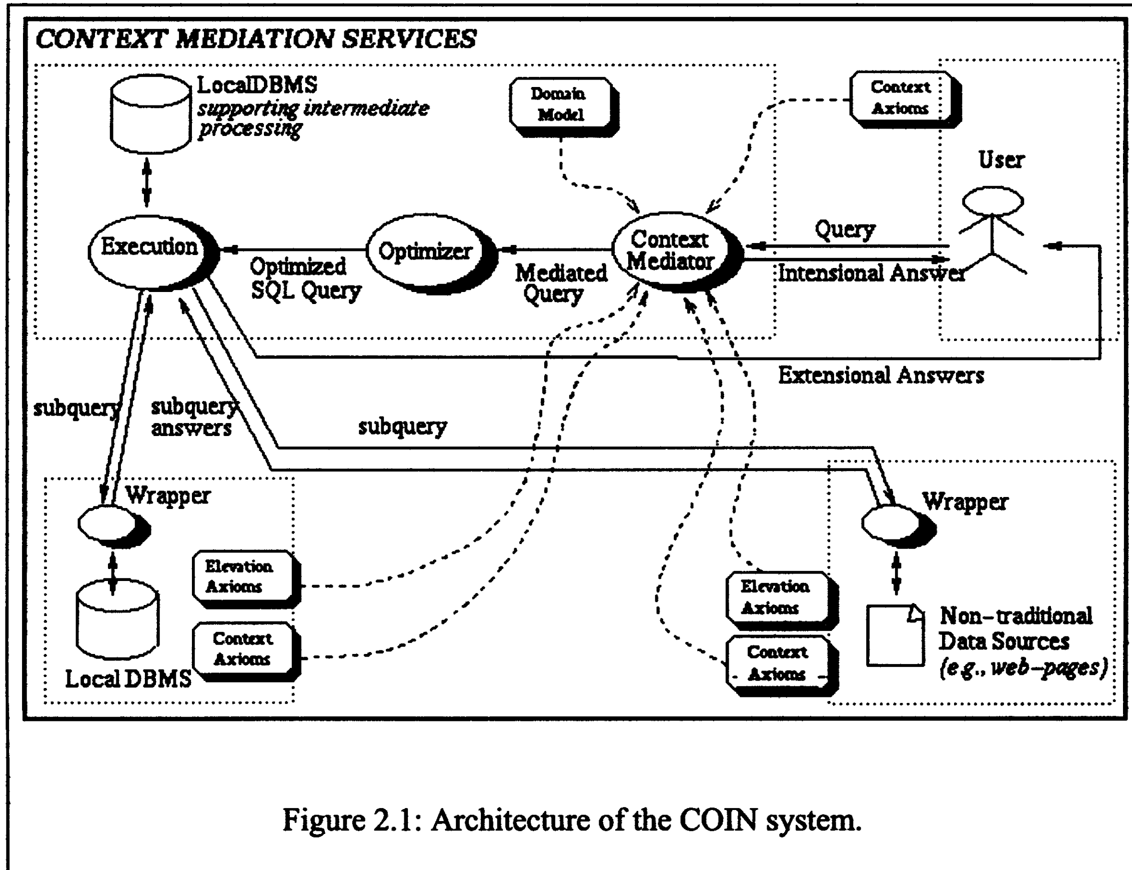

Figure 2.1: Architecture of the COIN system.

A user who wishes to query a remote database first expresses the query in Structured Query Language (SQL). This query is sent to the context mediator. The mediator uses the contexts of both the information source (the remote database) and the information receiver (the user) to translate the user's query into SQL subqueries appropriate to the source's context, as necessary. The mediated subqueries are sent to the information source, which returns data to the mediator. Finally, the mediator translates the data from the source into the receiver's context and passes the mediated data on to the receiver.

Figure 2.2 shows the architecture of the context mediator. The mediator receives an SQL query from the user, along with information about the user's context. The

mediator is also provided with a domain model, and with context, elevation, and supplementary axioms describing the context of the remote site being queried. All of this information is expressed in the logic-based language Datalog. An abduction engine is used to create a new SQL query in the context of the remote site. This query is then sent to the information source.

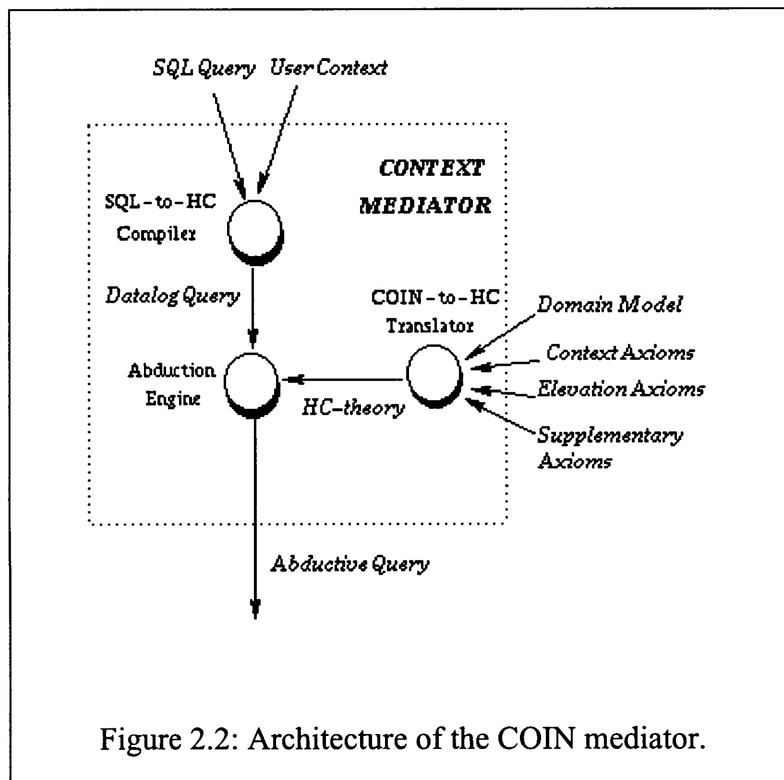

Figure 2.2: Architecture of the COIN mediator.

2.2 Applications of COIN

The COIN system can be used in any situation where data must be shared among diverse information sources and receivers which may not share the same context. A few examples of common data exchange situations where COIN may prove useful are as follows.[2]

[2] Some of these applications are discussed in the Context Interchange World Wide Web site at *http://context.mit.edu/~coin/*

- **Financial Decision Support:** There is a vast number of data sources for financial information available to investors. Many of these are foreign stock exchanges. Financial data tend to be presented in various currencies with various scale factors. Some databases express financial data in U.S. dollars; others use thousands of Japanese yen; still others present financial data for each company in the currency of the country where that company was incorporated. It is critical that queries to such databases be mediated to retrieve meaningful data and to make meaningful comparisons.
- **Manufacturing Inventory Control:** Large manufacturing processes are associated with large amounts of data, including design, engineering, manufacturing, and inventory information. Hundreds of contractors may be involved, and each contractor may present its data in a different manner. Context mediation is necessary to compare data in these disparate formats in a meaningful way.
- **Medical Information Systems:** The ability to share meaningful information about patients is necessary to ensure a high quality of medical care. Within a hospital, different departments such as internal medicine, admitting, and billing maintain separate sets of patient information, but must share these data to ensure good patient care. On a larger scale, different medical centers and clinics must share information with each other as well as with insurance companies and state and federal regulators. To share this information in a meaningful way, differences in procedure codes, classifications, and payment must be reconciled.

- **Academic Performance:** Academic performance is measured on different scales at different universities. For example, some universities use a five-point scale for grade point averages, while others use a four-point scale, and still others grade their students on 100-point scales. It is necessary to translate students' academic records among these diverse systems when comparing students from different universities, such as when evaluating graduate school applications.

Without a system like COIN in place to resolve semantic conflicts, these conflicts must be compensated for by the user. The user must be aware of all possible conflicts that may occur and how to correct each of them. For example, an employer comparing the academic records of potential employees must be aware of the fact that different universities use different grading systems. Two students may list their grade point averages as 4.0, but one student's university may grade on a scale of 4.0 while the other's grades on a scale of 5.0. Although the students' GPA's are the same, the student whose university bases its grades on a 4.0 scale has a better academic record than the student graded on a 5.0 scale. It would be an error for the employer to equate the two students' records. The problem becomes even more complex when international students are considered, since international students' academic performances are rated differently.

2.3 The Structure of a COIN Context

A context within COIN consists of four distinct segments: a domain model, context axioms, conversion functions, and elevation axioms.

2.3.1 Domain Model

The domain model is the basis of the context. It lists all of the semantic types that may potentially be involved in semantic conflicts, along with attributes and modifiers for each semantic type that link that type with other semantic types in the context.

Each semantic type consists of a name and a data type. The data type is normally one of the types native to the underlying system, including strings, integers, and so forth. Each semantic type may also include attributes and modifiers. Each attribute and modifier also has an associated data type, which may be one of the underlying system types or another semantic type defined within the domain model.

Several contexts may share the same domain model. In fact, it is necessary for good semantic conflict resolution that the semantic type involved in the conflict appear in the domain models of both the user and the remote data source.

2.3.2 Context Axioms

Context axioms specify values for each modifier of each semantic type defined in the domain model. For instance, if the domain model indicates that financial data are subject to a scale factor, the specific scale factor is specified in a context axiom.

The mediator compares the context axioms of the data source to those of the user to identify any semantic conflicts. When the value of a modifier in the context of the data source differs from the value of the same modifier in the context of the user, a semantic conflict that requires resolution by the mediator may occur.

Each context axiom contains the name of the modifier, the name of the associated semantic type, and the value of the modifier. The value of the modifier must be in the data type specified for that modifier in the domain model.

2.3.3 Conversion Functions

At times, the value of a given semantic object may be known with respect to one context, and this knowledge may restrict the possible values of that semantic object in another context. Conversion functions specify how the value of that semantic object may be determined with respect to the second context if its value in the first context is known.

2.3.4 Elevation Axioms

Elevation axioms define the mapping of semantic types from the domain model to the equivalent semantic objects in the data source. A set of elevation axioms includes the name of the external relation to be elevated, the context in which the source values are defined, the elevated relation name, and the names of the elevated semantic objects. It may also include explicit value declarations and semantic objects, if necessary.

This page intentionally left blank

Chapter 3

The Penny Language

Penny is a programming language designed specifically for use with the Context Interchange Project. It allows end users of COIN to create the clear and well-defined contexts necessary for semantic conflict resolution.

3.1 History of Penny

Penny was developed from COINL, a deductive object-oriented language also designed specifically for use with COIN.[1] COINL was a very powerful language for specifying complex contexts, but even very simple contexts required a large amount of cryptic COINL code. Learning and using COINL syntax was thus difficult and time-consuming.[2]

After several applications were designed with COINL, it became obvious that COINL was not meeting the needs of users. Users wanted to be able to read existing contexts and construct new ones very quickly, without spending weeks or months learning the complicated syntax of COINL.

---

[1] Fortunato Pena, "PENNY: A Programming Language and Compiler for the Context Interchange Project," M.Eng. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1997, 11-17.

[2] For more information on COINL, see: C. Goh, "Representing and Reasonning about Semantic Conflicts in Heterogeneous Information Systems," Ph.D. thesis, Sloan School of Management, Massachusetts Institute of Technology, 1996.

To meet this goal, Penny was created. Like COINL, it is a deductive object-oriented language. Like COINL, Penny is a compiled language; a compiler was created to transform Penny code into Datalog, which is understandable to the context mediator at the heart of the COIN system. Unlike COINL, however, Penny is a much easier language to learn. Its syntax is much less complicated, and is in some ways similar to the syntax of the C programming language. Learning the language, especially for an experienced computer programmer, thus takes much less time and effort than learning COINL.

Nevertheless, despite the fact that Penny is much easier to work with than COINL, Penny is still difficult to learn and use, especially for those unfamiliar with the techniques of computer programming. A solution to this problem is the translation and editing system presented in this thesis.

3.2 Syntax of Penny

As dictated by the design of COIN, there are four components to a context: the domain model, the context axioms, the conversion functions, and the elevation axioms. The syntax of each component is unique, and is described here in detail.

3.2.1 Domain Model

The domain model provides the underlying structure for a context. It contains a number of semantic types, each with associated modifiers and attributes.[3]

---

[3] Pena, 17-18.

Figure 3.1 shows an example of a domain model axiom.

```
semanticType companyName::number {
      modifier string format(ctx);
      attribute string countryIncorp;
};
```

Figure 3.1: Domain model axiom.

In this example, as shown in the first line of the axiom, the name of the semantic type being considered is *companyName*, with a data type of *number*. All semantic types, modifiers, and attributes are assigned data types; these data types may be either basic types such as *string* and *number*, or other semantic types defined elsewhere in the domain model.

Each semantic type may also have modifiers and attributes. This semantic type has one modifier, *format*, of type *string*. It also has one attribute, *countryIncorp*, also of type *string*.

3.2.2 Context Axioms

Context axioms specify values for the modifiers of the semantic types included in the domain model. If a semantic conflict is present, the values specified in the context axioms for the data source and the end user will disagree.[4]

Context axioms use the format shown in Figure 3.2.

[4] Pena, 18-19.

```
use ('/home/ncgotta/Work/dm0.pny');
context c_ds;
  format<companyName> = ~("ds_name");
  ...
end c_ds;
```

Figure 3.2: Context axioms.

First, a file specifying the domain model must be specified in the *use* statement. The domain model specifies the semantic types and modifiers which are assigned values in the context axioms following the statement.

The set of context axioms are delimited by the *context* and *end* statements. These statements also serve the purpose of defining the name of the context, in this case *c_ds*.

Each context axiom includes the names of the modifier and its semantic type, and the value being assigned to that modifier. The axiom takes the form

modifier-name < *semantic-type-name* > = ~(*value*)

where the value is of the same data type as the one specified for the modifier in the domain model.

It is also possible to define modifiers intensionally, that is, by assigning a value to the modifier that depends on the data being examined. For example, currencies are revalued from time to time, and the scale factor of financial data in a database may change depending on whether the information was recorded before or after the revaluation. When retrieving such data, it is necessary to identify the date corresponding to each entry and to be sure to use the appropriate scale factor.

An example of an intensional context axiom is presented in Figure 3.3. This particular axiom indicates that, in this context, the currency in which financial data are

given for each company is the currency of the country in which that company was incorporated.

```
use ('/home/ncgotta/Work/dm0.pny');
context c_ds;
  ...
  currency<companyFinancials> = ~($) <-
    Comp = self.company,
    Country = Comp.countryIncorp,
    CurrencyType = Country.officialCurrency,
    $ = CurrencyType.value;
  ...
end c_ds;
```

Figure 3.3: Intensional context axiom.

3.2.3 Conversion Functions

Conversion functions are used to obtain the value of a semantic object in one context if that value is known in another context. Conversion functions are of the form shown in Figure 3.4.

```
use ('/home/ncgotta/Work/dm0.pny');
context c_ds;

cvt()<companyName> <-
      SrcMod = self.format(source),
      TgtMod = self.format(target),
      SrcMod.value = TgtMod.value,
      $ = self.value;

...

end c_ds;
```

Figure 3.4: Conversion functions.

As with context axioms, conversion functions are delimited by use of the *context* and *end* statements, and a *use* statement is necessary to specify the underlying domain model.

Each conversion function starts with the key word *cvt*. This particular conversion function indicates that if the value of the modifier *format* of the semantic type *companyName* is the same in the source as it is in the target (i.e. it is the same in the contexts of both the end user and the data source), then if the value of *companyName* is known in either context, the value in the other context must be the same.[5]

3.2.4 Elevation Axioms

Elevation axioms relate the semantic objects used in the domain model to their equivalents in a database. These elevation axioms take the form shown in Figure 3.5.

```
elevate 'DiscAF'(cname, fyEnding, shares, income, sales, assets,
            incorp)
in c_ds
as 'DiscAF_p'(^cname:companyName, ^fyEnding:date, ^shares:void,
            ^income:companyFinancials, ^sales:companyFinancials,
            ^assets:companyFinancials, ^incorp:countryName)
{
      ...
};
```

Figure 3.5: Elevation axioms.

The first line of the set of elevation axioms names the external relation to be elevated, in this case *DiscAF*. It also lists each of the semantic objects in *DiscAF* that will be elevated. The second line names the context into which this relation will be

[5] Pena, 19-20.

elevated. In this case that context is *c_ds*. The third line provides a list of each semantic object in the external relation, paired with its corresponding semantic type in the context.

Following these statements, value declarations are listed within the brackets. Specific value declarations do not need to be provided, as the Penny compiler will create default declarations. These default values can, however, be overridden here with specific statements of the form

^semantic-object1 . modifier = ^semantic-object2

to allow specific mappings between given semantic objects and modifiers.[6]

[6] Pena, 20-21.

This page intentionally left blank

Chapter 4

The Translating and Editing System

The contexts of both users and data sources can be very complex and intricate. These contexts must be carefully defined, so as to avoid errors in definition that can lead to unresolved semantic conflicts or to the "resolution" of conflicts where none exist. Therefore, Penny, the language used to express these contexts in the COIN system, must of necessity be very complex to express these definitions adequately. As a result, it can be a difficult language to read, understand, and use.

Before the COIN system can be used, it is necessary that users wishing to use COIN, either to retrieve or to provide data, be able to define their contexts clearly. The first step in this process is to ensure that users can read the contexts they intend to use. The Penny translator accomplishes this goal by translating the domain models, context axioms, conversion functions, and elevation axioms expressed in Penny into a more readable and easier-to-understand format.

Once users are able to read existing contexts, it is necessary that they also be able to construct their own Penny domain models, context axioms, conversion functions, and elevation axioms to specify their own contexts. A set of editors has been created to allow the user to create different parts of a Penny context. Each editor allows the user to construct one of the four subsections of a context. This modular design allows users to utilize parts of a preexisting context, modifying them for their own needs. For instance, a

user may use a preexisting domain model as a basis for defining new context axioms with different values.

A system has been created that integrates all of these functions into one easy-to-use package. It was implemented in ANSI C in the UNIX operating environment. It was also compiled for use in the Windows 95 and Windows NT operating environments. Using this system, a user is able to read existing contexts and create new ones while knowing nothing about the syntax of the underlying Penny programming language. This system is described below.

Throughout this chapter, an example of an application of Context Interchange will be used to illustrate the capabilities of the system. This example is concerned with student academic records at universities. Each student's record contains a listing of the student's name, the number of subject units or credit hours the student has completed, and the student's grade point average (GPA). Since different universities assign different numbers of units to different courses and compute GPA's in different ways, there are several opportunities for semantic conflicts to arise.

4.1 Overview of the System

When the Penny translating and editing system is started, it presents the user with a main menu, as shown in Figure 4.1.

```
1   Translate existing context from Penny to English.
    Requires an existing, legal, Penny context.

2   Create Penny context axioms.
    Requires an existing Penny domain model.

3   Create a Penny domain model.

4   Create a set of Penny conversion functions.

5   Create Penny elevation axioms.

6   View a file (Penny or English).

0   Quit.

Please choose from the list above or enter 0 (zero) if finished:
```

Figure 4.1: Main menu of translation system.

The following options are available.

- **Option 1:** Penny to English translation system. This translator, described in section 4.2.1, allows the user to translate any part of a context defined in Penny into English.
- **Option 2:** Context axiom editor. This editor facilitates the construction of a set of context axioms based on a given domain model. It is described in section 4.2.3. Note that before the context axiom editor is run, a domain model must already exist as the basis for the context axioms to be created.
- **Option 3:** Domain model editor. This component, described in section 4.2.2, allows the user to construct a domain model by entering semantic types with their attributes and modifiers.
- **Option 4:** Conversion function editor. This component, described in section 4.2.4, allows the user to create conversion functions.

- **Option 5:** Elevation axiom editor. This editor, described in section 4.2.5, allows for the easy creation of elevation axioms.
- **Option 6:** File viewer. This component allows the user to view any file, Penny or English. Thus, users need not exit the system to see, for instance, the output of the Penny to English translator or the Penny code of the context axioms they have created. When this component is run, the system asks the user for a file name. If the file does not exist, the user is so informed. Otherwise, the file is displayed, twenty lines at a time, and the user is asked to press the enter key to see the next twenty lines. When the end of the file is reached, the user is asked to press the enter key, and is then returned to the main menu.
- **Option 0:** Exits the translating and editing system.

4.2 Major Components of the Translating and Editing System

There are five main components to the Penny translating and editing system. These are the Penny to English translator, the domain model editor, the context axiom editor, the conversion function editor, and the elevation axiom editor. These components are explained further in the following subsections.

4.2.1 The Penny to English Translator

The Penny translator is a knowledge-based system that uses a series of rules to determine how to translate each of the statements in a context. Each line of Penny code is parsed to determine whether it is part of a domain model, a context axiom, a conversion

function, or an elevation axiom. After this determination is made, the information contained in that line is then inserted into an appropriate template and written to an output file in a format that is easier for a person unfamiliar with Penny to read and understand. Thus, the output file contains all relevant information from the Penny context in a format understandable to anyone with a basic knowledge of the concepts used in COIN. Thus, specific knowledge of Penny syntax on the part of the end user becomes unnecessary.

The translator uses a set of rules to parse a Penny file and output a new file with the same information expressed in a more intuitive and easier to understand fashion. The list of rules used by the translator appears in Appendix A. This is an ad-hoc approach, since it depends on specific aspects of Penny syntax in order to arrive at a correct translation.

The Penny to English translator begins by asking the user to enter the name of the file containing the Penny code (the input file), followed by the name of the file to which the translation is to be written (the output file).

The input file should be an existing file consisting entirely of valid Penny code. If the input file contains no valid Penny code, the resulting output of the translator will be an empty output file. If the input file contains a mix of Penny code and other text, the contents of the output file may be unpredictable.

The output file should not be a preexisting file. If, however, there is a file by the same name already in existence, the user is provided with the following three options.

1. Append new information to the existing file. This option is useful if the user wishes to add more semantic types to a preexisting domain model.
2. Overwrite the existing file. This option erases all data in the existing file and replaces it with the new information.
3. Specify a different file name for the output.

The translator then proceeds to read the Penny file, one line at a time. It parses each line using its rule base to determine whether it is a domain model axiom, context axiom, conversion function, or elevation axiom, and which words in the line constitute the names of semantic types, modifiers, and values. If appropriate, the translator then writes a line to the output file stating this information in a format that is clearer than raw Penny code.

At times, it is impossible to tell from the single line under consideration whether that line is part of a domain model, context axiom, or elevation axiom, although the code surrounding that line makes it clear what sort of statement is being parsed. In these situations, the translator keeps track of what type of axiom to expect, and parses each line according to its expectations. For example, context axioms are set off from the surrounding Penny code by the use of *context* and *end* statements. When the translator encounters a *context* statement, it expects that each line following this statement will be a context axiom until an *end* statement appears.

4.2.2 The Domain Model Editor

The domain model editor allows a user to construct the domain model that underlies a context. The domain model lists the semantic types to be considered by the mediator, along with any attributes and modifiers those semantic types may have. It is the basis of the context, and it is therefore necessary to have a domain model in place before creating context and elevation axioms.

There is great freedom in the information that can be encapsulated in a domain model. A semantic type can be given any name, and it may or may not have attributes and modifiers. Because of this freedom, the editor communicates with the user by prompting the user to enter each required item, rather than by providing a menu of options.

When the user enters the domain model editor, the editor first requests a name for the file in which the Penny domain model code is to be stored. The file name may be any valid UNIX file name, and should not include spaces, asterisks, or other characters forbidden by UNIX. As with the Penny to English translator, if a file with the name provided by the user is found to exist, the user is provided with three options: append new data to the existing file, overwrite the existing file, or specify a different file name.

The first lines written to the output file are comments indicating that this file contains a domain model and stating the date and time of its creation.

The user is then prompted for the name of the first semantic type to be included in the domain model. Names of semantic types may not include spaces, tabs, or other

whitespace characters. When the semantic type name has been entered, the user is asked for its data type. In general, the data type will be either a string or a number.

Next, the system will prompt the user to enter any attributes desired for this semantic type. For each attribute, the user is asked for the attribute name and the type. An attribute's type may be one of the predefined data types (i.e. string or number) or a semantic type defined elsewhere in the domain model. At any time, the user may type the word *done*, instead of an attribute name, to indicate that no more attributes are to be added.

The system then prompts the user to enter any modifiers desired for this semantic type. As with attributes, the user is asked for each modifier's name and type. At any time, the user may type the word *done*, instead of a modifier name, to indicate that there are no more modifiers to be added to the semantic type.

At this point, the semantic type definition is complete. The user is asked if additional semantic types are to be entered. If the answer is yes, the process of entering a semantic type is repeated. Otherwise, the output file is closed and the process of creating a domain model is complete.

A domain model for the student academic record example might resemble the domain model presented in Figure 4.2.

```
semanticType studentRecord::string {
  attribute string studentName;
  attribute GPA studentGPA;
  attribute subjectUnits unitsCompleted;
}

semanticType GPA::number {
  modifier number maxGPA;
}

semanticType subjectUnits::number {
  modifier number unitsPerSubject;
}
```

Figure 4.2: Sample domain model.

The first domain model axiom expresses the information to be found in each student's record: name, GPA, and units completed. Since each record represents a different student with, presumably, a different name, students' names would never be compared. Therefore, a student's name would not cause a semantic conflict, and it is listed simply as an attribute of type string. GPA's and units completed may cause conflicts, however, and so they are expanded upon in additional domain model axioms.

Different universities calculate GPA's in different ways. Some universities use a scale where the maximum GPA attainable is 4.0, while others, such as the Massachusetts Institute of Technology (MIT), use a scale with a maximum of 5.0. Therefore, the second domain model axiom indicates that the GPA has a modifier called *maxGPA* which indicates whether the maximum attainable GPA at this student's university is 4.0, 5.0, or something else.

In the same way, different universities assign different numbers of units or credit hours to their courses. For instance, a typical course at MIT has 12 subject units, but other universities may assign 3 or 4 credit hours to a typical course, and still others

merely count the number of courses the student has taken, making each course worth 1 unit. The third domain model axiom indicates this by assigning a modifier, *unitsPerSubject*, that indicates the number of units assigned to a typical subject taught at the student's university.

The following dialog takes place when this domain model is constructed. Figure 4.3 shows the creation of the initial domain model statements and the first domain model axiom. Figure 4.4 shows the creation of the second and third domain model axioms.

```
Please enter the name of the output file
dm0.pny

Please enter the name of the semantic type
studentRecord
Please enter a type for this semantic type, ie. string, number.
string

Please enter the name of the first attribute.
If you do not wish to assign any attributes, type DONE then press
enter.
studentName
Please enter a type for attribute studentName, ie.
ie. string, number, or a previously defined semantic type.
string

Please enter the name of the next attribute.
If you do not wish to assign any more attributes, type DONE then
press enter.
studentGPA
Please enter a type for attribute studentGPA, ie.
ie. string, number, or a previously defined semantic type.
GPA

Please enter the name of the next attribute.
If you do not wish to assign any more attributes, type DONE then
press enter.
unitsCompleted
Please enter a type for attribute unitsCompleted, ie.
ie. string, number, or a previously defined semantic type.
subjectUnits

Please enter the name of the next attribute.
If you do not wish to assign any more attributes, type DONE then
press enter.

done

Please enter the name of the first modifier.
If you do not wish to assign any modifiers, type DONE then press
enter.
done

Do you wish to create any more axioms (y/n)
y
```

Figure 4.3: Initial domain model editor dialog and creation of first domain model axiom.

```
Please enter the name of the semantic type
GPA
Please enter a type for this semantic type, ie. string, number.
number

Please enter the name of the first attribute.
If you do not wish to assign any attributes, type DONE then press
enter.
done

Please enter the name of the first modifier.
If you do not wish to assign any modifiers, type DONE then press
enter.
maxGPA
Please enter a type for modifier maxGPA,
ie. string, number, or a previously defined semantic type.
number

Please enter the name of the next modifier.
If you do not wish to assign any modifiers, type DONE then press
enter.
done

Do you wish to create any more axioms (y/n)
y

Please enter the name of the semantic type
subjectUnits
Please enter a type for this semantic type, ie. string, number.
number

Please enter the name of the first attribute.
If you do not wish to assign any attributes, type DONE then press
enter.
done

Please enter the name of the first modifier.
If you do not wish to assign any modifiers, type DONE then press
enter.
unitsPerSubject
Please enter a type for modifier unitsPerSubject,
ie. string, number, or a previously defined semantic type.
number

Please enter the name of the next modifier.
If you do not wish to assign any modifiers, type DONE then press
enter.
done

Do you wish to create any more axioms (y/n)
n
```

Figure 4.4: Creation of second and third domain model axioms.

4.2.3 The Context Axiom Editor

The context axiom editor allows the user to specify context axioms, which define values for each modifier of each semantic type in a domain model. Because the semantic types and modifiers defined in the domain model are integral parts of the context axioms, it is necessary that the domain model be specified before context axioms can be created.

When the user enters the context axiom editor, the editor first asks for the name of the file containing the domain model, and for the name of the output file to which context axioms are to be written. As with the domain model editor, if this output file already exists, the user is asked whether the system should append the new context axioms to it, replace the existing file, or request a new file name.

The user is then asked for a context name. The name serves to identify the context, particularly when elevation axioms are constructed. The system writes to the output file comments indicating that the file contains a set of context axioms, the name of the context, and the date and time.

The user is presented with a menu listing each semantic type defined in the domain model. Any semantic type from the menu may be selected, or 0 may be entered if the user has finished defining context axioms.

If the user selects a semantic type, a menu is provided that lists all the modifiers associated with that semantic type. Again, the user may choose any modifier from the menu, or enter 0 if no context axioms are to be defined for any of the listed modifiers. If the user enters 0, the system returns to the previous menu listing semantic types.

If the user selects a modifier, a reminder of the semantic type and modifier chosen is provided, and the user is asked for a value for the modifier. Any string or number may be entered. As with names of semantic types, modifier values may not include spaces, tabs, or other whitespace characters.

After entering a value for the modifier, the user is returned to the menu listing the semantic types defined in the domain model. When all context axioms have been entered, the user may enter 0 when presented with this menu. When this occurs, the output file is closed and the user is returned to the main menu.

Context axioms for the student academic record example might resemble the context axioms presented in Figure 4.5. These context axioms express the context of MIT, where the maximum attainable GPA is 5.0 and a typical course is worth 12 subject units.

```
use ('dm0.pny');

context c_mit;

maxGPA<GPA> = ~(5.0);
unitsPerSubject<subjectUnits> = ~(12);

end c_mit;
```

Figure 4.5: Sample context axioms.

The user dialog required to create these context axioms is shown in Figure 4.6.

```
Please enter name of Penny file specifying domain model
dm0.pny
Please enter the name of the output file
ca.pny
Please enter a name for this context: c_mit

1   studentRecord
2   GPA
3   subjectUnits
Please choose from the list above or enter 0 (zero) if finished:  2

Please choose one of the modifiers listed below for semantic type GPA
Or enter 0 (zero) to return to previous menu.

1  maxGPA (type: number)
Please choose from the list above or enter 0 (zero) if finished:  1

You have selected semantic type GPA and modifier maxGPA.
Please enter a value for this modifier.
5.0

1   studentRecord
2   GPA
3   subjectUnits
Please choose from the list above or enter 0 (zero) if finished:  3

Please choose one of the modifiers listed below for semantic type
subjectUnits
Or enter 0 (zero) to return to previous menu.

1  unitsPerSubject (type: number)
Please choose from the list above or enter 0 (zero) if finished:  1

You have selected semantic type subjectUnits and modifier
unitsPerSubject.
Please enter a value for this modifier.
12
1   studentRecord
2   GPA
3   subjectUnits
Please choose from the list above or enter 0 (zero) if finished:  0
```

Figure 4.6: User dialog for creation of context axioms.

4.2.4 The Conversion Function Editor

Since conversion functions are often very complex, it is difficult to capture them adequately, either in a form-driven system, such as the one used in the domain model editor, or in a menu-driven system, such as the one used in the context axiom editor.

Simpler conversion functions can be created by a form-driven editor, while more intricate functions require a knowledge of Penny. The conversion editor therefore employs two approaches: a form-driven interface for simpler axioms, and the option to enter more complex axioms directly in Penny.

The system first asks the user for a name for the output file that will contain the conversion functions. As with the previously described components of the system, if the file specified already exists, the user has the option to append new information to the file, overwrite the file, or choose a new file name.

The system starts by writing comments into the output file, indicating that this file contains conversion functions, and stating the date and time of its creation. Next, the user is asked for the name of the file containing the domain model, followed by the name of the context. The first two lines of the conversion functions, specifying this information, are written to the output file.

The user is then presented with a menu. The first option of this menu allows the user to create a conversion function for a situation where the value of a particular modifier is the same in both the source and the target systems. If this option is chosen, the user is then asked for the names of the semantic type and modifier in question, and an appropriate conversion function is constructed automatically.

The second menu option allows the user to enter conversion functions directly in Penny code. This is necessary for more complex functions. These conversion functions are written directly to the output file specified by the user. When the conversion function

has been entered, the user may type the word *done* on a line by itself. This will return the user to the menu.

When all the conversion functions have been created, the user may enter a 0 at the menu prompt. At this point, the user is asked whether the simple conversion functions for strings and numbers should be added. If the answer is yes, these functions are added to the output file. The ending line of the conversion functions is then written to the output file, which is closed, and the user is returned to the main menu.

For the academic record example discussed above, a number of conversion functions must be created to account for differences in GPA scales and subject units among universities. The two conversion functions shown in Figure 4.7 can be used to deal with GPA's.

```
cvt()<GPA> <-
  SrcMod = self.maxGPA(source),
  TgtMod = self.maxGPA(target),
  SrcMod.value = TgtMod.value,
  $ = self.value;

cvt()<GPA> <-
  SrcMod = self.maxGPA(source),
  TgtMod = self.maxGPA(target),
  SrcVal = SrcMod.value,
  TgtVal = TgtMod.value,
  SrcVal <> TgtVal,
  Val = self.value,
  Ratio = TgtVal / SrcVal,
  $ = Val * Ratio;
```

Figure 4.7: Sample conversion functions.

In each of these conversion functions, a student's GPA is known in one context (the source) and the maximum GPA in both the source and target contexts is known. The first conversion function states that if the maximum GPA is the same for both source and

target, then the student's GPA in the target context is the same as in the source. The second function indicates that if the maximum GPA is not the same, the student's GPA in the target context is equal to his GPA in the source context, multiplied by the ratio of the maximum GPA in the target context to the maximum GPA in the source context.

The first conversion function is of the simpler variety where the value of the *maxGPA* modifier is the same in both the source and the target contexts. This type of conversion function can be created without direct knowledge of Penny. Only the names of the semantic type and modifier in question must be entered. The dialog for the creation of this conversion function is shown in Figure 4.8.

```
1   Make a conversion function for a situation where the source and
    target values of a particular modifier are the same.
2   Make a more complex conversion function (requires knowledge of
    Penny).

Please choose from the list above or enter 0 (zero) if finished:  1
Enter the name of the semantic type with which this conversion
function is concerned.
GPA
Enter the name of the modifier in question.
maxGPA
```

Figure 4.8: Dialog for creation of simple conversion function.

The second conversion function, however, is more complex and requires knowledge of Penny. It would be created as shown in Figure 4.9.

```
1   Make a conversion function for a situation where the source and
    target values of a particular modifier are the same.
2   Make a more complex conversion function (requires knowledge of
    Penny).

Please choose from the list above or enter 0 (zero) if finished:  2
Enter conversion function in Penny below.
When you are finished, type DONE on a line by itself.

cvt()<GPA> <-
  SrcMod = self.maxGPA(source),
  TgtMod = self.maxGPA(target),
  SrcVal = SrcMod.value,
  TgtVal = TgtMod.value,
  SrcVal <> TgtVal,
  Val = self.value,
  Ratio = TgtVal / SrcVal,
  $ = Val * Ratio;
done
```

Figure 4.9: Dialog for creation of complex conversion function.

4.2.5 The Elevation Axiom Editor

The elevation axiom editor allows the user to construct that part of a context which relates the semantic types used in the domain model to their equivalents in a particular database. It requires that the domain model be provided by the user, so that valid semantic types can be extracted from it.

When the elevation axiom editor is started, the user is first asked for the name of a file specifying a valid domain model. Semantic types for the elevation axioms are drawn from the specified domain model. Next, the user must provide a file name for the output file. As with the previous editors, if the output file already exists, the user may choose to append new data to it, replace it, or choose a new file name.

The user is then asked to provide the names of the external relation to be elevated, the context, and the elevated relation. At this point, the system writes comments into the

output file indicating that this file contains elevation axioms, stating the names of the external relation and the context, and stating the date and time of the file's creation.

The user is then asked to enter each elevated semantic object and to choose a semantic type from the domain model for each such object. This information is restructured into Penny code and written to the output file.

The user is then asked to enter any overriding value declarations. Since these declarations can be quite complex, they must be entered by the user in Penny code and written directly to the output file. When all overriding value declarations have been entered, the user may type the word *done* on a line by itself, at which point the output file is closed and the user is returned to the main menu.

Elevation axioms for the academic record example may resemble the elevation axioms in Figure 4.10.

```
elevate 'Students'(name, gradePointAvg, unitsTaken)
in c_mit
as 'Students_p'(^name          : studentName,
                ^gradePointAvg : studentGPA,
                ^unitsTaken    : unitsCompleted)
{};
```

Figure 4.10: Sample elevation axioms.

The external relation to be elevated is *Students*, and its elevated relation is *Students_p*. The relation is being elevated in the context *c_mit* discussed in Section 4.2.3. The semantic objects to be elevated from the database are *name*, *gradePointAvg*, and *unitsTaken*; their corresponding semantic types in the context are *studentName*,

studentGPA, and *unitsCompleted*, respectively. There are no overriding value declarations.

The user dialog necessary to create these elevation axioms is shown in Figure 4.11.

```
Please enter the name of the output file
elev.pny

Please enter the name of the external relation to be elevated
Students

Please enter the name of the context
c_mit

Please enter the name of the elevated relation
Students_p

Please enter the name of the first elevated semantic object, as
defined in the external relation.
name

Enter its corresponding semantic type from the context.
studentName

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
gradePointAvg

Enter its corresponding semantic type from the context.
studentGPA

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
unitsTaken

Enter its corresponding semantic type from the context.
unitsCompleted

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
done
Enter below, in Penny, any overriding value declarations you wish to
include.
When you are finished, type DONE on a line by itself.

done
```

Figure 4.11: Dialog for creation of elevation axioms.

This page intentionally left blank

Chapter 5

Example of Use of Translating and Editing System

A sample run through the translating and editing system will indicate more clearly how the system works in practice. The sample Penny code used here is drawn from the code used in the implementation of a sample application created to illustrate the capabilities of the COIN system. The application integrates several data sources containing financial information for various foreign and domestic companies. This code is reproduced in full in Appendix B, along with reference codes to indicate the structural similarity among different axioms.[1]

Throughout this chapter, user input is indicated in *italics*.

5.1 Domain Model Editor

Because the domain model is the underlying core of any context, the first step in defining the context is creation of the domain model. To accomplish this, the translating and editing system is run and option 2 is chosen from the main menu, as described in the previous chapter.

The system then asks for a file name for the domain model, as shown in Figure 5.1.

---

[1] For more information on this application, see the Context Interchange World Wide Web site at *http://context.mit.edu/~coin/*

```
Please enter the name of the output file
dm0.pny
```

Figure 5.1: User dialog for obtaining file name for domain model.

The user is then asked for the name of the first semantic type, from which the first domain model axiom will be constructed. Figure 5.2 shows the dialog that takes place in the construction of the first domain model axiom in the sample application.

```
Please enter the name of the semantic type
companyFinancials
Please enter a type for this semantic type, ie. string, number.
number

Please enter the name of the first attribute.
If you do not wish to assign any attributes, type DONE then press
enter.
company
Please enter a type for attribute company, ie.
ie. string, number, or a previously defined semantic type.
companyName

Please enter the name of the next attribute.
If you do not wish to assign any more attributes, type DONE then
press enter.
fyEnding
Please enter a type for attribute fyEnding, ie.
ie. string, number, or a previously defined semantic type.
date

Please enter the name of the next attribute.
If you do not wish to assign any more attributes, type DONE then
press enter.
done

Please enter the name of the first modifier.
If you do not wish to assign any modifiers, type DONE then press
enter.
scaleFactor
Please enter a type for modifier scaleFactor,
ie. string, number, or a previously defined semantic type.
number

Please enter the name of the next modifier.
If you do not wish to assign any modifiers, type DONE then press
enter.
currency
Please enter a type for modifier currency,
ie. string, number, or a previously defined semantic type.
currencyType

Please enter the name of the next modifier.
If you do not wish to assign any modifiers, type DONE then press
enter.
done

Do you wish to create any more axioms (y/n)
y

Please enter the name of the semantic type
```

Figure 5.2: Dialog for creation of domain model axiom.

The Penny code resulting from this dialog is shown in Figure 5.3.

```
semanticType companyFinancials::number {
  attribute companyName company;
  attribute date fyEnding;

  modifier number scaleFactor(ctx);
  modifier currencyType currency(ctx);
};
```

Figure 5.3: Penny domain model axiom.

Each time an axiom is completed, the user is asked if he wishes to create any more axioms. If the answer is yes, the process is repeated. If the answer is no, the output file is closed and the user is returned to the main menu.

5.2 Context Axiom Editor

Once the domain model is defined, the user must move on to define context axioms by selecting option 3 from the main menu.

The editor first asks for the name of the file specifying the domain model, followed by a file name for the context axioms, and a name for the context to be created, as shown in Figure 5.4.

```
Please enter name of Penny file specifying domain model
dm0.pny
Please enter the name of the output file
c_ds.pny
Please enter a name for this context: c_ds
```

Figure 5.4: Initial user dialog for context axiom editor.

At this point, the first two lines of the set of context axioms are written, as shown in Figure 5.5.

```
use ('dm0.pny');

context c_ds;
```

Figure 5.5: First two lines of context axiom Penny code.

These lines indicate the name of the file containing the domain model and the name of the context being created, as well as indicating that the following lines are specific context axioms.

Below are the steps involved in the creation of one of the context axioms from the Disclosure context. The user is first presented with a menu of semantic types, as shown in Figure 5.6.

```
1    companyFinancials
2    companyName
3    exchangeRate
4    date
5    currencyType
6    countryName
Please choose from the list above or enter 0 (zero) if finished:  1
```

Figure 5.6: Menu of semantic types.

Next, the user is presented with a menu of the modifiers of the semantic type chosen. This menu is shown in Figure 5.7.

```
Please choose one of the modifiers listed below for semantic type
companyFinancials
Or enter 0 (zero) to return to previous menu.

1  scaleFactor (type: number)
2  currency (type: currencyType)
Please choose from the list above or enter 0 (zero) if finished: 1
```

Figure 5.7: Menu of semantic type modifiers.

The user is now asked to enter a value for the chosen modifier, as shown in Figure 5.8.

```
You have selected semantic type companyFinancials and modifier
scaleFactor.
Please enter a value for this modifier or enter $ to define it
intensionally.
1
```

Figure 5.8: User dialog for entering modifier value.

At this point, the context axiom has been created. The resulting Penny code is shown in Figure 5.9. The user is then returned to the menu of semantic types.

```
scaleFactor<companyFinancials> = ~(1);
```

Figure 5.9: Penny context axiom.

It is also possible to define modifiers intensionally, which requires some knowledge of Penny code. The user selects the semantic type and modifier as usual, but instead of entering a value for the modifier, a dollar sign ($) is entered. At this point the user is prompted to enter the intensional definition of the modifier, in Penny, and to end by typing the word *done* on a line by itself.

Figure 5.10 shows the full dialog that would occur when creating a context axiom of this variety.

```
1   companyFinancials
2   companyName
3   exchangeRate
4   date
5   currencyType
6   countryName
Please choose from the list above or enter 0 (zero) if finished:  1

Please choose one of the modifiers listed below for semantic type
companyFinancials
Or enter 0 (zero) to return to previous menu.

1  scaleFactor (type: number)
2  currency (type: currencyType)
Please choose from the list above or enter 0 (zero) if finished: 2

You have selected semantic type companyFinancials and modifier
currency.
Please enter a value for this modifier or enter $ to define it
intensionally.
$

Enter context axiom in Penny below.
When you are finished, type DONE on a line by itself.

Comp=self.company,
Country=Comp.countryIncorp,
CurrencyType=Country.officialCurrency,
$=CurrencyType.value;
done

1   companyFinancials
2   companyName
3   exchangeRate
4   date
5   currencyType
6   countryName
Please choose from the list above or enter 0 (zero) if finished:
```

Figure 5.10: Dialog for the creation of an intensionally defined modifier.

Whenever the menu of semantic types appears, the user may select another semantic type and create another context axiom, in which case the process described

above is repeated. The user may also enter 0, in which case the following line, shown in Figure 5.11, is written to the output file, indicating the end of the context axioms.

```
end c_ds;
```

Figure 5.11: Final line of context axiom Penny code.

After this final line is written, the output file is closed and the user is returned to the main menu.

5.3 Conversion Function Editor

The user now chooses option 4 from the main menu to enter the conversion function editor. This editor begins by asking the user for the name of the output file, as shown in Figure 5.12.

```
Please enter the name of the output file
conv_ds.pny
```

Figure 5.12: User dialog for obtaining file name for conversion axioms.

The user is then asked to enter the name of the file containing the domain model, and the name of the context under consideration, as shown in Figure 5.13.

```
Please enter name of Penny file specifying domain model
dm0.pny

Please enter the name of the context
c0
```

Figure 5.13: User dialog for obtaining domain model and context names.

This dialog results in the creation of the first two lines of the conversion function file, shown in Figure 5.14.

```
use('dm0.pny');

context c0;
```

Figure 5.14: First two lines of conversion functions.

At this point, the user is presented with the following menu.

```
1   Make a conversion function for a situation where the source and
    target values of a particular modifier are the same.
2   Make a more complex conversion function (requires knowledge of
    Penny).

Please choose from the list above or enter 0 (zero) if finished:
```

Figure 5.15: Conversion function menu.

For each conversion function where the source and target values of a modifier are the same, the user selects 1 from this menu, and the dialog shown in Figure 5.16 occurs, after which the user is returned to the menu.

```
Enter the name of the semantic type with which this conversion
function is concerned.
date
Enter the name of the modifier in question.
dateFmt
```

Figure 5.16: User dialog for creating a conversion function where the source and target values are the same.

This dialog creates the conversion function shown in Figure 5.17.

```
cvt()<date> <-
  SrcMod = self.dateFmt(source),
  TgtMod = self.datefmt(target),
  SrcMod.value = TgtMod.value,
  $ = self.value;
```

Figure 5.17: A conversion function where source and target values are the same.

For more complex conversion functions, the user chooses option 2 from the menu, and the dialog shown in Figure 5.18 occurs.

```
Enter conversion function in Penny below.
When you are finished, type DONE on a line by itself.

cvt(scaleFactor,Val)<companyFinancials><-
  FsvMod=self.scaleFactor(source),
  FtvMod=self.scaleFactor(target),
  Fsv=FsvMod.value,
  Ftv=FtvMod.value,
  Fsv<>Ftv,
  Ratio=Fsv/Ftv,
  $=Val*Ratio;
done
```

Figure 5.18: User dialog for entering complex conversion functions.

When the complex conversion functions have been entered, the user types the word *done* on a line by itself. When this occurs, the user is returned to the menu.

When all conversion functions have been entered, the user enters a 0 at the menu prompt. At this point, the editor asks the user whether it should add the simple conversion functions for strings and numbers. This dialog is shown in Figure 5.19.

```
Do you want to add the simple conversion functions for strings and
numbers?
yes
```

Figure 5.19: User dialog for addition of string and number conversion functions.

If the answer is yes, the additional conversion functions shown in Figure 5.20 are added to the output file.

```
cvt()<number> <-
  $ = self.value;

cvt()<string> <-
  $ = self.value;
```

Figure 5.20: String and number conversion functions.

After this is done, the final line of the conversion functions, shown in Figure 5.21, is written to the output file.

```
end c0;
```

Figure 5.21: Final line of conversion functions.

Then, the output file is closed and the user is returned to the main menu.

5.4 Elevation Axiom Editor

Finally, the user creates the last component of the context, the elevation axioms. Here the user will create a subset of the elevation axioms for Disclosure.

The user starts by selecting option 5 from the main menu, and entering the output file name when prompted, as shown in Figure 5.22.

```
Please enter the name of the output file
ds_elev.pny
```

Figure 5.22: User dialog for obtaining file name for elevation axioms.

The system then asks for the names of the external relation, the context, and the elevated relation, as shown in Figure 5.23.

```
Please enter the name of the external relation to be
elevated
DiscAF

Please enter the name of the context
c_ds

Please enter the name of the elevated relation
DiscAF_p
```

Figure 5.23: Initial user dialog for elevation axiom editor.

At this point, the system will ask for the names of all elevated semantic objects, as defined in the external relation, and their corresponding semantic types within the context. This dialog is shown in Figure 5.24.

```
Please enter the name of the first elevated semantic object, as
defined in the external relation.
cname

Enter its corresponding semantic type from the context.
companyName

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
fyEnding

Enter its corresponding semantic type from the context.
date

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
shares

Enter its corresponding semantic type from the context.
void

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
income

Enter its corresponding semantic type from the context.
companyFinancials

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
sales

Enter its corresponding semantic type from the context.
companyFinancials

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
assets

Enter its corresponding semantic type from the context.
companyFinancials

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
incorp

Enter its corresponding semantic type from the context.
countryName

Please enter the name of the next elevated semantic object,
or type DONE if you are finished entering objects.
done
```

Figure 5.24: User dialog for entering semantic objects into elevation axioms.

The first three lines of the elevation axioms, including the names of the external relation, context, elevated relation, and all elevated semantic objects with their corresponding semantic types, are then written to the output file, as shown in Figure 5.25.

```
elevate 'DiscAF'(cname, fyEnding, shares, income, sales,
assets, incorp)
in c_ds
as 'DiscAF_p'(^cname:companyName,
  ^fyEnding:date,
  ^shares:void,
  ^income:companyFinancials,
  ^sales:companyFinancials,
  ^assets:companyFinancials,
```

Figure 5.25: First three lines of elevation axiom Penny code.

The user is then asked for any overriding value declarations, which must be entered directly in Penny. This is shown in Figure 5.26.

```
Enter below, in Penny, any overriding value declarations you wish to
include.
When you are finished, type DONE on a line by itself.

^cname.countryIncorp=^incorp;
^income.company=^cname;
^income.fyEnding=^fyEnding;
...
done
```

Figure 5.26: User dialog for overriding value declarations.

When the user types *done* to finish entering value declarations, the remainder of the elevation axioms are written to the output file, that file is closed, and the user is returned to the main menu.

5.5 Penny to English Translator

As an example of the capabilities of the Penny to English translator, this section demonstrates the translation of a small subset of the Penny code from the sample application into English using the translator.

The code translated in this example includes one domain model axiom, one context axiom, and a shortened set of elevation axioms. This code is shown in Figure 5.27.

```
% Domain Model
% Created: Sat Mar  7 18:16:25 1998


semanticType companyFinancials::number {
  attribute companyName company;
  attribute date fyEnding;
  modifier number scaleFactor(ctx);
  modifier currencyType currency(ctx);
};

% Context Axioms for context c_ds
% Created: Sat Mar  7 18:17:59 1998


use ('test.pny');

context c_ds;

scaleFactor<companyFinancials> = ~(1);

end c_ds;

% Elevation Axioms for external relation DiscAF into context
c_ds
% Created: Sat Mar  7 18:18:58 1998


elevate 'DiscAF'(income, sales, assets)
in c_ds
as 'DiscAF_p'(^income : companyFinancials,
  ^sales : companyFinancials,
  ^assets : companyFinancials)
{
  ^income.company=^cname;
  ^income.fyEnding=^fyEnding;
};
```

Figure 5.27: Sample Penny code for translation.

When the translator is started, it asks the user for the names of the Penny input file and the output file. The output file should not be a preexisting file; if it is, the user may choose to overwrite the existing file, append new data to it, or specify a new filename. This dialog is shown in Figure 5.28.

```
Please enter the name of the Penny input file
sample.pny

Please enter the name of the output file
sample1.out

This file already exists.  Please choose one of the following options:
1   Overwrite the file (existing data in the file will be permanently
erased)
2   Append new data to the existing file
3   Specify a different filename

3
Please enter the name of the output file
sample.out
```

Figure 5.28: User dialog for obtaining file names for translator. Includes dialog that occurs if the output file name provided already exists.

At this point, the translator proceeds to read in one line at a time from the input file, translate it, and write output to the output file.

For the sample Penny code given above, the translator produces the output shown in Figure 5.29.

```
Domain model axioms for semantic type: companyFinancials
  Attribute company is of type companyName
  Attribute fyEnding is of type date
  Modifier scaleFactor is of type number
  Modifier currency is of type currencyType

Context Axioms for c_ds
Modifier scaleFactor of semantic type companyFinancials
has the value 1

Elevation axioms for 'DiscAF'
using context c_ds

  Semantic object ^income is of type companyFinancials,
  Semantic object ^sales is of type companyFinancials,
  Semantic object ^assets is of type companyFinancials
  ^income.company=^cname;
  ^income.fyEnding=^fyEnding;
```

Figure 5.29: Translator output.

This page intentionally left blank

Chapter 6

Conclusions and Future Work

6.1 Summary of Contributions

The translating and editing system created in this thesis provides a useful extension to the COIN system, in that it enables users to read and create COIN contexts in Penny without specific knowledge of Penny formats and syntax. This will, in turn, make it more likely that a given user will choose to use COIN for semantic conflict mediation. COIN is useless unless contexts are provided by both the user and the information source; hence, the more widespread COIN becomes, the more useful it becomes.

6.2 Future Work

There are a number of possibilities for future extensions to this translating and editing system. Some of these are described below.

6.2.1 Natural Language Processing

The system can be extended to take advantage of natural language processing techniques to allow users to express a desired context in written or spoken English without the need for the structure of menus and prompts. Natural language processing would allow the system to take a sentence such as "I want all financial information in U.S. dollars" and generate any necessary code. The system may either generate Penny

code, which would then be compiled into Datalog, or it may generate Datalog code directly, for immediate use with the context mediator.

6.2.2 Graphical and World Wide Web Interfaces

The system currently uses a simple text-based interface. Users respond to menus and prompts that are not always intuitive. A graphical user interface for context creation and display can make the system more user-friendly.

One possible component of a graphical interface is the capacity to create a graphical representation of a domain model, where each semantic type is represented by a block and the interconnecting modifiers and attributes are represented by lines. This representation would allow a user to see the interactions among semantic types more easily, and to create new semantic types, attributes, and modifiers by merely drawing a few lines and boxes.

An option for the addition of a graphical user interface is the construction of a set of HTML documents, which would allow for interaction with the translation software through the World Wide Web. Menu options would be presented as hyperlinks, and prompts where the user must enter data would be presented as form inputs, which would allow the user to enter data to be passed to the translation software. Such an interface would also allow the translation to be used on many different platforms; at present, the system can only be used on a few platforms.

6.2.3 Translation Directly to Datalog

At present, the system translates only between Penny and the user. It is incapable of either reading or writing Datalog code. Instead, any code created by the user must be compiled into Datalog before it can be used by the context mediator. A possible future extension of this project is to construct a translating and editing system that translates between Datalog and the user, eliminating Penny altogether.

6.3 Conclusions

The translation and editing system constructed in this thesis is an invaluable tool for aiding end users in the use of the Context Interchange system. The system allows users to read existing contexts and to construct new ones quickly and easily, without requiring that the users learn how to structure these contexts in Penny, thus enhancing the Context Interchange system.

This page intentionally left blank

Appendix A

Penny to English Translator Rules

The following rules are used to ensure that the Penny to English translator correctly parses and translates any Penny code with which it is provided.

The translator works by examining each line, parsing it, and, if necessary, writing the appropriate text into the output file. For more details, see Chapter 4.

Each rule used by the translator consists of two parts: an "if" clause detailing the conditions that must be met for the rule to be fired, and a "then" clause detailing the actions that are performed when the rule is fired.

In the rules, the following variables are used. *Firstword* refers to the first word on the line under consideration, while *Secondword* refers to the second word on the line and *Thirdword* refers to the third word on the line.

Contexttype is a state variable used to indicate whether the translator expects the line under consideration to be part of a domain model, part of a set of context axioms, part of a set of conversion functions, or part of a set of elevation axioms.

Table A.1: Rules considered regardless of value of *Contexttype*

If	Then
First character of *Firstword* is %	do nothing (the line is a comment)

Table A.2: Rules considered when *Contexttype* = "domain model"

If	Then
Contexttype = "domain model" *Firstword* = "semanticType"	write to output: "Domain model axiom: *Secondword*"
Contexttype = "domain model" *Firstword* = "attribute"	write to output: " Attribute *Thirdword* is of type *Secondword*"
Contexttype = "domain model" *Firstword* = "modifier"	write to output: " Modifier *Thirdword* is of type *Secondword*"
Contexttype = "domain model" *Firstword* = "context"	*Contexttype* = "context axioms"
Contexttype = "domain model" *Firstword* = "elevate"	*Contexttype* = "elevation axioms"

Table A.3: Rules considered when *Contexttype* = "context axioms"

If	Then
Contexttype = "context axioms" *Firstword* = "end"	*Contexttype* = "domain model"
Contexttype = "context axioms" *Firstword* = "cvt"	*Contexttype* = "conversion functions" write to output: "Conversion functions appear below"
Contexttype = "context axioms" *Firstword* != "end" and *Firstword* != "cvt"	write to output: "*Firstword* modifier of *Secondword* type has value *Thirdword*"

Table A.4: Rules considered when *Contexttype* = "conversion functions"

If	Then
Contexttype = "conversion functions" *Firstword* = "end"	*Contexttype* = "domain model"
Contexttype = "conversion functions" *Firstword* != "end"	write line from input file to output

Table A.5: Rules considered when *Contexttype* = "elevation axioms"

If	Then
Contexttype = "elevation axioms" *Firstword* = "elevate"	write to output: "Elevation axioms for *Secondword*"
Contexttype = "elevation axioms" *Firstword* = "in"	write to output: "using context *Secondword*"
Contexttype = "elevation axioms" *Firstword* = "as"	1. until a { is encountered, read in two words from input file, and write to output: "Semantic object *Firstword* is of type *Secondword*" 2. after a { is encountered and until a } is encountered, read in each line from input file, and write it to output file 3. *Contexttype* = "domain model"

This page intentionally left blank

Appendix B

Sample Application Penny Code

This appendix contains the full Penny code for the sample application presented in Chapter 5. This application is aimed at integrating various data sources containing financial information for a number of both foreign and domestic companies.[1]

Each axiom in the Penny code is accompanied by a reference number. Those reference numbers, defined in Table B.1 and Table B.2 below, indicate which of the axioms illustrated in Chapter 5 the axiom in question most resembles; the procedure used to create the axiom shown in Chapter 5 can be used to create all axioms with the same reference number.

[1] This code is drawn from: Fortunato Pena, "PENNY: A Programming Language and Compiler for the Context Interchange Project," M.Eng. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1997, 37-45.

Table B.1: Axiom reference numbers for domain model and context axioms

Reference Number	Axiom Type	Axiom Illustrated in Chapter 5

D1	Domain Model	semanticType companyFinancials::number {
		attribute companyName company;
		attribute date fyEnding;
		
		modifier number scaleFactor(ctx);
		modifier currencyType currency(ctx);
		};

CH	Context Axiom Header	use('dm0.pny'); context c_ds;
C1	Ordinary Context Axiom	scaleFactor<companyFinancials> = ~(1);
C2	Intensional Context Axiom	currency<companyFinancials> = ~($) <- Comp = self.company, Country = Comp.countryIncorp, CurrencyType = Country.officialCurrency, $ = CurrencyType.value;
CE	Context Axiom End	end c_ds;

Table B.2: Axiom reference numbers for conversion functions and elevation axioms

<table>
<tr><th>Reference Number</th><th>Axiom Type</th><th>Axiom Illustrated in Chapter 5</th></tr>
<tr><td>VH</td><td>Conversion Function Header</td><td><pre>use('dm0.pny');
context c0;</pre></td></tr>
<tr><td>V1</td><td>Simple Conversion Function</td><td><pre>cvt()<date> <-
  SrcMod = self.dateFmt(source),
  TgtMod = self.dateFmt(target),
  SrcMod.value = TgtMod.value,
  $ = self.value;</pre></td></tr>
<tr><td>V2</td><td>Complex Conversion Function</td><td><pre>cvt(scaleFactor,Val)<companyFinancials> <-
  FsvMod = self.scaleFactor(source),
  FtvMod = self.scaleFactor(target),
  Fsv = FsvMod.value,
  Ftv = FtvMod.value,
  Fsv <> Ftv,
  Ratio = Fsv / Ftv,
  $ = Val * Ratio;</pre></td></tr>
<tr><td>VSN</td><td>String and Number Conversion Functions</td><td><pre>cvt()<number> <-
  $ = self.value;
cvt()<string> <-
  $ = self.value;</pre></td></tr>
<tr><td>VE</td><td>Conversion Function End</td><td><pre>end c0;</pre></td></tr>
<tr><td>EH</td><td>Elevation Axiom Header</td><td><pre>elevate 'DiscAF'(cname, fyEnding, shares,
income, sales, assets, incorp)
in c_ds
as 'DiscAF_p'(^cname:companyName,
  ^fyEnding:date,
  ^shares:void,
  ^income:companyFinancials,
  ^sales:companyFinancials,
  ^assets:companyFinancials,
  ^incorp:countryName)</pre></td></tr>
<tr><td>EB</td><td>Elevation Axiom Body</td><td><pre>{
     ^income.company = ^cname;
     ^income.fyEnding = ^fyEnding;

     ^sales.company = ^cname;
     ^sales.fyEnding = ^fyEnding;

     ^assets.company = ^cname;
     ^assets.fyEnding = ^fyEnding;

     ^incorp.officialCurrency = ~curType;

     ~curType.value = $ <-
          ~curType = Incorp.officialCurrency,
          Y = Incorp.value,
          'Currencytypes'(Y, $);
};</pre></td></tr>
</table>

B.1 Domain Model

D1
```
semanticType companyFinancials::number {
    attribute companyName company;
    attribute date fyEnding;

    modifier number scaleFactor(ctx);
    modifier currencyType currency(ctx);
};
```

D1
```
semanticType companyName::number {
    modifier string format(ctx);
    attribute string countryIncorp;
};
```

D1
```
semanticType exchangeRate::number {
    attribute currencyType fromCur;
    attribute currencyType toCur;
    attribute date txnDate;
};
```

D1
```
semanticType date::string {
    modifier string dateFmt(ctx);
};
```

D1
```
semanticType currencyType::string {
    modifier string curTypeSym(ctx);
};
```

D1
```
semanticType countryName::string {
    attribute currencyType officialCurrency;
};
```

B.2 Context Axioms

B.2.1 Disclosure (DiscAF)

CH
```
use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

context c_ds;
```

C1
```
scaleFactor<companyFinancials> = ~(1);
```

C2
```
currency<companyFinancials> = ~($) <-
     Comp = self.company,
     Country = Comp.countryIncorp,
     CurrencyType = Country.officialCurrency,
     $ = CurrencyType.value;
```

C1
```
format<companyName> = ~("ds_name");
```

```
C1    dateFmt<date> = ~("American Style /");

C1    curTypeSym<currencyType> = ~("3char");

CE    end c_ds;
```

B.2.2 Worldscope (WorldAF)

```
CH    use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

      context c_ws;

C1    scaleFactor<companyFinancials> = ~(1000);

C1    currency<companyFinancials> = ~('USD');

C1    format<companyName> = ~("ws_name");

C1    dateFmt<date> = ~("American Style /");

C1    curTypeSym<currencyType> = ~("3char");

CE    end c_ws;
```

B.2.3 Datastream (DStreamAF)

```
CH    use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

      context c_dt;

C1    scaleFactor<companyFinancials> = ~(1000);

C2    currency<companyFinancials> = ~($) <-
           Comp = self.company,
           Country = Comp.countryIncorp,
           CurrencyType = Country.officialCurrency,
           $ = CurrencyType.value;

C1    format<companyName> = ~("dt_name");

C1    dateFmt<date> = ~("European Style -");

C1    curTypeSym<currencyType> = ~("2char");

CE    end c_dt;
```

B.2.4 Olsen

```
CH    use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

      context c_ol;

C1    dateFmt<date> = ~("European Style /");

CE    end c_ol;
```

B.3 Conversion Functions

```
VH    use('/home/tpena/work/coinlc/Penny/examples/tasc/penny/dm0.pny');

      context c0;

      %%-----------------------------------------------------------------
      %% conversion functions for companyFinancials
      %%-----------------------------------------------------------------
V2    cvt()<companyFinancials> <-
           U = self.value,
           W = self.cvt(scaleFactor, U),
           $ = self.cvt(currency, W);

      %%-----------------------------------------------------------------
      %% conversion functions for  companyFinancials w.r.t. scaleFactors
      %%-----------------------------------------------------------------
V1    cvt(scaleFactor, Val)<companyFinancials> <-
           SrcMod = self.scaleFactor(source),
           TgtMod = self.scaleFactor(target),
           SrcMod.value = TgtMod.value,
           $ = Val;

V2    cvt(scaleFactor, Val)<companyFinancials>  <-
           FsvMod = self.scaleFactor(source),
           FtvMod = self.scaleFactor(target),
           Fsv = FsvMod.value,
           Ftv = FtvMod.value,
           Fsv <> Ftv,
           Ratio = Fsv / Ftv,
           $ = Val * Ratio;

V1    cvt(currency, Val)<companyFinancials> <-
           SrcMod = self.currency(source),
           TgtMod = self.currency(target),
           SrcMod.value = TgtMod.value,
           $= Val;
```

```
V2    cvt(currency, Val)<companyFinancials> <-
           SrcMod = self.currency(source),
           TgtMod = self.currency(target),
           SrcMod.value <> TgtMod.value,
           FyDate = self.fyEnding,
           olsen_p(FromCur, ToCur, Rate, TxnDate),
           SrcMod.value = FromCur.value,
           TgtMod.value = ToCur.value,
           FyDate.value = TxnDate.value,
           $ = Val * Rate.value;

      %%-------------------------------------------------------------------
      %% conversion functions for companyName
      %%-------------------------------------------------------------------
V1    cvt()<companyName> <-
           SrcMod = self.format(source),
           TgtMod = self.format(target),
           SrcMod.value = TgtMod.value,
           $ = self.value;

V2    cvt()<companyName> <-
           SrcMod = self.format(source),
           TgtMod = self.format(target),
           SrcMod.value = "ds_name",
           TgtMod.value = "ws_name",
           'Name_map_Ds_Ws_p'(DsName, WsName),
           self.value = DsName.value,
           $ = WsName.value;

V2    cvt()<companyName> <-
           SrcMod = self.format(source),
           TgtMod = self.format(target),
           SrcMod.value = "ws_name",
           TgtMod.value = "ds_name",
           'Name_map_Ds_Ws_p'(DsName, WsName),
           self.value = WsName.value,
           $ = DsName.value;

      %%-------------------------------------------------------------------
      %% conversion functions for dateFmts
      %%-------------------------------------------------------------------
V1    cvt()<date> <-
           SrcMod = self.dateFmt(source),
           TgtMod = self.dateFmt(target),
           SrcMod.value = TgtMod.value,
           $ = self.value;

V2    cvt()<date> <-
           SrcMod = self.dateFmt(source),
           TgtMod = self.dateFmt(target),
           SrcFormat = SrcMod.value,
           TgtFormat = TgtMod.value,
           SrcFormat <> TgtFormat,
           SrcVal = self.value,
           datexform($, SrcVal, SrcFormat, TgtFormat);
```

```
      %%-----------------------------------------------------------------
      %% conversion functions for currency symbols.
      %%-----------------------------------------------------------------
V1    cvt()<currencyType> <-
           SrcMod = self.curTypeSym(source),
           TgtMod = self.curTypeSym(target),
           SrcMod.value = TgtMod.value,
           $ = self.value;

V2    cvt()<currencyType> <-
           SrcMod = self.curTypeSym(source),
           TgtMod = self.curTypeSym(target),
           SrcMod.value = "3char",
           TgtMod.value = "2char",
           'Currency_map_p'(Char3, Char2),
           self.value = Char3.value,
           $ = Char2.value;

V2    cvt()<currencyType> <-
           SrcMod = self.curTypeSym(source),
           TgtMod = self.curTypeSym(target),
           SrcMod.value = "2char",
           TgtMod.value = "3char",
           'Currency_map_p'(Char3, Char2),
           self.value = Char2.value,
           $ = Char3.value;

      %%-----------------------------------------------------------------
      %% conversion functions for number, a simple value
      %% cvt function for number is overridden by the following:
      %%    * companyFinancials
      %%-----------------------------------------------------------------
VSN   cvt()<number> <-
           $ = self.value;

      %%-----------------------------------------------------------------
      %% conversion functions for strings, a simple value
      %% cvt function for string is overridden by the following:
      %%    * companyName
      %%    * date
      %%-----------------------------------------------------------------
VSN   cvt()<string> <-
           $ = self.value;

VE    end c0;
```

B.4 Elevation Axioms

B.4.1 Disclosure

```
      %%-----------------------------------------------------------------
      %% Disclosure:DiscAF
      %%-----------------------------------------------------------------
```

```
EH    elevate 'DiscAF'(cname, fyEnding, shares, income, sales, assets,
            incorp)
      in c_ds
      as 'DiscAF_p'(^cname : companyName, ^fyEnding : date,
            ^shares : void, ^income: companyFinancials,
            ^sales : companyFinancials, ^assets: companyFinancials,
            ^incorp: countryName)
      {
EB         ^cname.countryIncorp = ^incorp;

           ^income.company = ^cname;
           ^income.fyEnding = ^fyEnding;

           ^sales.company = ^cname;
           ^sales.fyEnding = ^fyEnding;

           ^assets.company = ^cname;
           ^assets.fyEnding = ^fyEnding;

           ^incorp.officialCurrency = ~curType;

           ~curType.value = $ <-
                ~curType = Incorp.officialCurrency,
                Y = Incorp.value,
                'Currencytypes'(Y, $);
      };
```

B.4.2 Worldscope

```
      %%------------------------------------------------------------------
      %% Worldscope:WorldAF (Keys: CompanyName and Date)
      %%------------------------------------------------------------------
EH    elevate 'WorldAF'(cname, fyEnding, shares, income, sales, assets,
            incorp)
      in c_ws
      as 'WorldAF_p'(^cname : companyName,    ^fyEnding : date, ^shares
      : number,
                     ^income: companyFinancials, ^sales :
      companyFinancials,
                     ^assets: companyFinancials, ^incorp: countryName)
      {
EB         ^cname.countryIncorp = ^incorp;

           ^income.company = ^cname;
           ^income.fyEnding = ^fyEnding;

           ^sales.company = ^cname;
           ^sales.fyEnding = ^fyEnding;

           ^assets.company = ^cname;
           ^assets.fyEnding = ^fyEnding;

           ^incorp.officialCurrency = ~curType;

           ~curType.value = $ <-
                ~curType = Incorp.officialCurrency,
```

```
          Y = Incorp.value,
          'Currencytypes'(Y, $);
};
```

B.4.3 Olsen

```
    %%------------------------------------------------------------------
    %% Olsen (Keys: Source Currency, Target Currency, and Date)
    %%------------------------------------------------------------------
EH  elevate olsen(exchanged, expressed, rate, date)
    in c_ol
    as olsen_p(^exchanged : currencyType, ^expressed : currencyType,
               ^rate      : exchangeRate, ^date      : date)
    {
EB       ^rate.fromCur = ^expressed;
         ^rate.toCur   = ^exchanged;
         ^rate.txnDate = ^date;
    };
```

B.4.4 Datastream

```
    %%------------------------------------------------------------------
    ------
    %%  Datastream: DStreamAF (Keys:  As_of_date and  Name)
    %%------------------------------------------------------------------
    ------
EH  elevate 'DStreamAF'(date, name, total_sales, total_items_pre_tax,
                  earned_for_ordinary, currency)
    in c_dt
    as 'DStreamAF_p'(^date                : date,
                  ^name                : companyName,
                  ^total_sales         : companyFinancials,
                  ^total_items_pre_tax : companyFinancials,
                  ^earned_for_ordinary : companyFinancials,
                  ^currency            : currencyType)
    {
EB       ^name.countryIncorp = ~incorp;

         ^total_sales.fyEnding = ^date;
         ^total_sales.company = ^name;

         ^total_items_pre_tax.fyEnding = ^date;
         ^total_items_pre_tax.company = ^name;

         ^earned_for_ordinary.fyEnding = ^date;
         ^earned_for_ordinary.company = ^name;

         ~incorp.officialCurrency = ^currency;
         ~incorp.value = $ <-
              Currency = ~incorp.officialCurrency,
              Two = Currency.value,
              'Currencytypes_p'(Country, Three),
              Two = Three.value,
```

```
            $ = Country.value;
};
```

B.4.5 Auxiliary Elevation Axioms

```
%%---------------------------------------------------------------
------
%%  Auxillary Tables
%%---------------------------------------------------------------
------
```

EH
```
elevate 'Currencytypes'(country, currency)
in c_ds
as 'Currencytypes_p'(^country : string, ^currency : string) {};
```

EH
```
elevate 'Currency_map'(cur3, cur2)
in c_ds
as 'Currency_map_p'(^cur3 : string, ^cur2 : string) {};
```

EH
```
elevate 'Name_map_Ds_Ws'(dsNames, wsNames)
in c_ds
as 'Name_map_Ds_Ws_p'(^dsNames : string, ^wsNames : string) {};
```

EH
```
elevate 'Name_map_Dt_Ds'(dtName, dsName)
in c_dt
as 'Name_map_Dt_Ds_p'(^dtName : string, ^dsName : string) {};
```

EH
```
elevate 'Name_map_Dt_Ws'(dtName, wsName)
in c_dt
as 'Name_map_Dt_Ws_p'(^dtName : string, ^wsName : string) {};
```

This page intentionally left blank

Appendix C

List of Abbreviations

ANSI	American National Standards Institute
COIN	COntext INterchange
COINL	COntext INterchange Language
HTML	HyperText Markup Language
SQL	Structured Query Language

This page intentionally left blank

References

Bressan, S., et. al. The COntext INterchange Mediator Prototype. ACM SIGMOD International Conference on Management of Data, 1997.

Goh, Cheng Hian. "Representing and Reasoning about Semantic Conflicts in Heterogeneous Information Systems." Ph.D. thesis, Sloan School of Management, Massachusetts Institute of Technology, 1996.

Hoffman, James. "Introduction to Structured Query Language." World Wide Web: *http://w3.one.net/~jhoffman/sqltut.htm.* 1998.

Pena, Fortunato. PENNY: A Programming Language and Compiler for the Context Interchange Project. M.Eng. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1997.

"The COntext INterchange Project." World Wide Web: *http://context.mit.edu/~coin/.*