A Design Document for the CORAL Deductive Database System

FANG LIN

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

April 2004

Your file Votre référence
ISBN: 0-612-91068-7
Our file Notre référence
ISBN: 0-612-91068-7

Abstract

A Design Document for the CORAL Deductive Database System

Fang Lin

CORAL is a deductive database system that supports a rich declarative language, and an interface to C++ that allows for a combination of declarative and imperative programming. It is important to know the design and the implementation for CORAL in order to be able to extend this system.

This report documents the design for CORAL. The Unified Modeling Language (UML) is used to describe the Architecture, Structural Model and Behavioral Model. The design documentation is based on the study of the source code, the user manual and the published papers about CORAL.

Acknowledgements

I would like to thank my supervisor, Dr. Gregory Butler, for his patience and valuable guidance.

Thanks to the thesis examiners for their precious advice.

Thanks to my colleagues, including Guang Wang, Liqian Zou and Lugang Xu, for their kindly help during different steps of my work.

Finally, special thanks to my husband, Xin Zhao, for his selfless support!

Table of Contents

List of Figures

Chapter 1 Introduction

This report is the design document of CORAL. CORAL is a deductive database system that supports a powerful declarative query language. The language supports general Horn clause logic programs, extended with SQL-style grouping, set-generation, and negation. Programs can be organized into independently optimized modules, and users can provide optimization hints in the form of high-level annotations. The system supports a wide variety of optimization techniques. It also supports non-ground facts. That means CORAL permits variables within facts. The data can be persistent on the disk or can be reside in main-memory [2]. CORAL supports a client-server architecture where one or more CORAL clients can execute queries and access facts from a shared CORAL server.

1.1 Why make the design document of CORAL?

In the Know-It-All Project [12] developed at Concordia University, there is a subproject "Diagrammatic Query Interface". The major goal of this subproject is to incorporate deductive databases, in the form of CORAL, into the Know-It-All framework for databases [12]. CORAL will work as a VIEWDB subclass defining an intensional database view of a relational database. For this purpose, CORAL needs to connect with a relational database. The existing CORAL provides support for both main-memory and disk-resident data. There are three ways to store the disk-resident data: first, the data can be stored through EXODUS storage manager; second, the data can be stored in files on the disk; third, the data can be stored in some relational databases. The data stored in files can be consulted by the system, but the whole file must be loaded in the main memory. If

the file is large, it will use lots of main memory. The data in the extensional database and the data stored in the CORAL runtime data structure can interact with each other by CORAL commands. In CORAL 1.5.2, CORAL supports to talk with two kinds of relational databases: Sybase and Commercial Ingres. However, these two kinds of database are not suitable to be used in Know-It-All project. So we need to do some extension on CORAL. CORAL is implemented in C/C++ [2]. The source code is open. It is possible to make some extension for the convenience of using CORAL. In order to extend CORAL to communicate with other relational databases, we must know how CORAL was designed and implemented. As the complexity of the software system grows, the challenge in understanding, maintaining and extending the system increases. The architecture and the design document for a software system ease the efforts required for software engineers to reuse and make changes to the source code. In some published papers [1] [2] [4], there are only some general descriptions of the system architecture. However, among all the publications related to CORAL, there is no document talking about the detailed design. It is necessary to analyze CORAL and its source code and then to extract the design document of this system. To describe the design of the system, the UML [6] notation is chosen because it is a well-established tool that has been used to model software systems. UML provides a rich set of diagrams (e.g., class diagrams, sequence diagrams) to model both the static structure and the dynamic structure of each component of a system.

1.2 Contribution of the Report

In this report, we describe the design of CORAL that are extracted by us from the source code, documents such as user manual [4], and existing literature on the CORAL

deductive database system [1] [2]. The main efforts and works of this report include: studying the main features of CORAL deductive database system; learning the query evaluation and optimization strategies of CORAL and comparing the effects of different optimization strategies; analyzing the structure and components of the architecture of CORAL, extracting the interactions between the individual components; classifying the CORAL system into subsystems and describing the relationships between them; modeling the static structure of the system, representing different dependent relations among classes, such as inheritance, aggregation and association; summarizing the main functionality of major classes, especially the classes that implement the query evaluation and optimization strategies and the classes that implement the communication between CORAL and a relational database; examining the dynamic aspects of CORAL and illustrating object behaviors using sequence diagrams; analyzing the major data structures of CORAL and explaining how CORAL represents different types of data.

I believe that the work on the design documents will facilitate a better understanding of CORAL, so that some new functions can be added in the future.

1.3 Organization of the Report

The rest of the report is organized as follows: Chapter 2 introduces the background of the work. Chapter 3 gives a brief description of CORAL, including some key concepts, declarative language features and the query evaluation and optimization strategies; Chapter 4 illustrates the architecture of the CORAL system; Chapter 5 describes some major data structures used in CORAL; Chapter 6 presents the structural model of

CORAL; Chapter 7 discusses the behavioral model of CORAL; and Chapter 8 concludes the report. Appendix A gives the data dictionary.

Chapter 2 Background

In this chapter we will discuss some background regarding to some concepts and tools using in this report.

2.1 Relational Databases and Deductive databases

As mentioned in previous chapter, CORAL is a deductive database. In this section we will discuss what is a deductive database, and first we will describe what is a relational database.

A relational database is a collection of relations with distinct relation names. A relation consists of a relation schema, which specifies the relation name, the name of each field, and the name of the domain associated with each field, and a relation instance, which is a set of tuples. The tuples in a relation have the same number of fields as the relation schema, and values in a field are required to be in the set of values associated with the domain for that field [5].

A deductive database is a combination of a conventional database containing facts, a knowledge base containing rules, and an inference engine which allows the derivation of information implied by the facts and rules. Commonly, the knowledge base is expressed in a subset of first-order logic and either a SLDNF or Datalog inference engine is used.

In other words, a deductive database is a database system that supports Datalog rules[5]. Datalog is a relational query language inspired by Prolog.

In addition to the data stored in a database (extensional facts), a deductive database stores rule based knowledge concerning parts of the real world which are modeled by the system. A deductive database management system (DDBMS) consists of two parts:

1) A database management system.
2) A deduction mechanism that uses extensional data (facts) as well as intensional data (rules) for deducing new facts.

The extensional database (EDB) refers to a set of facts. And the intensional database (IDB) refers to the collection of rules. At compiler time, only the IDB are examined. The actual contents of EDB are assumed to be unavailable at compiler time. So in the deductive database system, the IDB is viewed as a program and the EDB is viewed as the input to the program.

2.2 What CORAL differ from Other Deductive Databases

Now there are a couple of deductive database systemss, such as: Aditi [9], GLUE-NAIL [10] and LOLA [11]. Following is a brief description of them.

Aditi is developed at University of Melbourne. It is a multi-user deductive database system. It supports base relations defined by facts (relations in the sense of relational databases) and derived relations defined by rules that specify how to compute new information from old information.

GLUE-Nail is developed at Stanford University. It is a combination of the logic query language NAIL! with the imperative programming language GLUE in the context of an essentially relational data model.

LOLA is developed at University of Passau. It has been designed as the query answering component of a deductive database system and integrates ideas from logic programming

and relational query processing. LOLA is based on a clausal logic programming language with function symbols, negation, grouping and aggregation, special predicates for accessing multiple external relational databases, and user-definable computed predicates.

Among all the deductive database systems, CORAL is the deductive database that supports non-ground facts. This feature makes CORAL differ from most other deductive databases.

2.3 What is UML?

UML [6] (Unified Modelling Language) is the international standard notation for object oriented analysis and design. It is defined by the Object Management Group (www.omg.org). UML is a standard notation for the modeling of real-world objects as a first step in developing an object-oriented program. UML is a graphical language for visualizing, specifying, constructing and documenting the artifacts of software systems [6]. In this report, we use Class Diagram, Package Diagram, Sequence Diagram and Collaboration Diagram to describe the system design of CORAL Deductive Database System.

Chapter 3 CORAL Overview

CORAL is a database programming language developed at the University of Wisconsin--Madison. It is a deductive database system that supports a rich declarative language, provides a wide range of evaluation methods, and allows a combination of declarative and imperative programming[1]. The data can be persistent on disk or can reside in main memory.

The CORAL project was initiated in 1988-1989. Preliminary versions of the system have been used by a few groups, and the latest release version is 1.5.2 (November 26, 1997). Source code for CORAL (written in C++) is available by anonymous ftp from ftp://ftp.cs.wisc.edu/coral/. Information about CORAL can be accessed at http://www.wisc.edu/CORAL.

3.1 Key concepts in CORAL

Relations

In CORAL, data is stored as tuples in relations [1]. Relations of different kinds are supported: first, there are stored (or base) relations. Then there are view (or derived) relations. These relations are expressed as rules that specify how they are to be computed. When a query is asked on such a relation, the actual relation is computed based on the rules. Certain commands like print, display timer, etc. are builtin relations.

Tuple

Every stored relation is composed of tuples.

Arguments

A tuple is composed of arguments, which could have a type. Basic types like integers and strings, as well as complex types like sets and relations are supported.

Terms

A term is:

-- Any constant symbol.

-- Any variable.

-- Any n-place function symbol $f(t_1, \ldots, t_n)$ where f is a function symbol of arity n (functor), and each t_i is a term.

Functor

A functor is a function symbol.

Variables binding

To answer the query, the system retrieves all tuples from the relation that match the pattern specified in the query. Formally, a unification is performed between the query and the tuples in the relation, and when there is a successful unification, the bindings corresponding to the query variables are returned as the answer.

Iterator

When all the matching tuples need to be returned for a query, an iterator is set up over the relation. The interface to a relation is "get next tuple". It takes the query pattern (i.e. the argument list), and returns the next tuple from the relation that matches the query pattern.

Workspace

A workspace is a collection of relations, which can be either EDB relations or relations corresponding to predicates exported by modules.

Some lexical convention:

A relation is always specified using lower case letters.

A variable always begins with an UPPER CASE letter.

All CORAL command/rules end with a period.

3.2 Concepts of declarative language features

Rules

Rules in declarative modules are essentially Horn clauses. Rules take the form *head:-body*. A rule is to be read as an assertion that for all assignments of terms to the variables that appear in the rule, the head is true if the body is true. In particular, a fact is a rule with an empty body.

Modules

A CORAL user can organize sets of rules and facts into modules. Modules serve as a means of encapsulating the definition of derived relations. In addition to rules, a module may posses a numbers of annotations, which are user specified hints to the systems. These hints can be used to specify rewriting techniques, evaluation strategies and much more. A module begins with the keyword "module" and ends at the keyword "end_module". When CORAL receives "end_module", it compiles the module. When a module is compiled, it provides a definition for one or more predicates that it exports. Queries on these predicates are answered using the compiled form of these definitions. A predicate exported form one module can be used in another module.

Non-Ground Facts

CORAL permits variables within facts [1]. Here is an example to illustrate non-grand facts.

```
module listroutines.
export  append(bbf, bfb).
append([ ],L,L).
append([H|T], L, [H|L1]):- append(T, L, L1).
end_module.
```

Figure 3.1: An example of a module

Fgure 3.1 shows a module. The module exports the predicate *append*. CORAL permits to query *append* like the example in Figure 3.2:

```
?append([1,2,3,4,X],[Y,Z],ANS).
```

Figure 3.2: An example of a query

An answer with variable could be obtained as shown in Figure 3.3:

```
ANS=[1,2,3,4,X,Y,Z].
```

Figure 3.3 The answer of the query

Negation

In CORAL, the keyword "not" is used as a prefix to indicate a negative body literal [1]. For example, given a predicate *parent*, it can be tested if a is not a parent of b by using "*not parent(a,b)*". CORAL supports a class of programs with negation that properly contains the class of non-floundering left-to-right modularly stratified programs [8]. A program is non-floundering if all variables in a negated literal are ground before the literal is evaluated (in the left-to-right rule order).

Sets and Multisets

In CORAL, Sets and Multisets are allowed. They are used as values and a couple of operations defined for them. There are two ways to create sets and multisets. One is set-enumeration({}) and the other is set-grouping(<>). For example, {1,2,3,f(a,b),a} is a set. {1,f(a),f(a)} is a multiset.

3.3 An example of a CORAL code and its execution

Figure 3.4 is an example to describe what is the definition of a module and how to execute the query on it.

In Figure 3.4, commands in line 1 to line 4 are inputting facts to relation *parent*.

Line 5 is the command "list_rels." that display all the relation in the memory.

Line 8 is CORAL response to the user. It tells the user that there is a relation, *parent,* which is a base relation (not defined by rules).

In line 10, "module start_eg1" marks the beginning of this module; the name "start_eg1" of module is required.

In line 11, "export grandparent(ff,bf)." declares that this module provides a definition of the predicate "grandparent", and that two query forms are optimized. The first optimized query form is grandparent_ff; another is grandparent_bf. After compiler, for each query form, CORAL generates a compiled version internal module structure called QFModuleDate, along with other useful data structures. Here, f denotes an argument that can be free and b stands an argument that must be bound in the query.

In line 12, the rule "grandparent(X,Z):-parent(X,Y),parent(Y,Z)." defines the predicate *grandparent*. It mains: "X is the grandparent of Z if X is Y's parent and Y is Z's parent".

In line 13, "end_module." marks the end of the module. When CORAL receives "end_module", it compiles the module.

Line 15 is the command "list_rels" to verify if the module has been correctly complied.

Line 17,18, 19, 20 show the relations in the memory. It displays one base relation and two derived relations with corresponding query form; one is grandparent_ff, another is grandparent_bf.

In line 22, the command "?grandparent(U,V)." is a query to see all the grandparent facts. The query form is grandparent_ff. The query is evaluated using the optimized version of the module for the query form grandparent_ff. CORAL shows the two answer of this query form line 23 to line 27 in Figure 3.4.

In line 28, "?grandparent(adam,X)" is a query to find out all the grandchildren of "adam". The query is evaluated using the optimized version of the module for the query form grandparent_bf. CORAL shows only one answer to this query from line 29 to line 31.

```
1.  ready>>parent(adam,cain).
2.  ready>>parent(adam,abel).
3.  ready>>parent(eve,cain).
4.  ready>>parent(eve,abel).
5.  ready>>parent(abel,sem).
6.  ready>>list_rels.
7.
8.  Parent/2 :      (base)
9.
10. ready>>module start_eg1.
11. ready>>export grandparent(ff,bf).
12. ready>>grandparent(X,Z):-  parent(X,Y),parent(Y,Z).
13. ready>>end_module.
14.
15.  ready>>list_rels.
16.
17.  Parent/2 :      (base)
18.
19. grandparent_bf/2    : (derived) : indexes
20. grandparent_ff/2    : (derived) : indexes
21.
22.  ready>>?grandparent(U,V).
23.  U=adam, V=sem.
24.        ... next answer ? (y/n/all)[y]y
25.  U=eve, V=sem.
26.        ... next answer ? (y/n/all)[y]y
27.  (Number of Answers = 2)
28. ready>>?grandparent(adam, X).
29. X=sem.
30.       ... next answer ? (y/n/all)[y]y
31.   (Number of Answers = 1)
```

Figure 3.4: An example of CORAL code and its execution

3.4 Query Evaluation and Optimization

3.4.1 Overview

CORAL system uses a number of query evaluation strategies, and each technique is particularly efficient for some classes of programs, but may perform relatively poorly on others. CORAL uses two basic evaluation approaches: pipelining[2] and materialization[2].

3.4.2 Materialization

CORAL supports several kinds of materialized evaluation. They include: Basic Semi-Naïve[2], Predicate Semi-Naïve[2], and Ordered Search[2]. These kinds of materialization are all bottom-up fixpoint evaluation methods. Bottom-up evaluation iterates over a set of rules, repeatedly evaluating them until a fixpoint is reached.

SCC –Strong Connected Components

Considering the rules within a module, notice that interdependencies sometimes exist between some of the relations. A dependency graph is a directed graph showing the dependencies between relations. A strongly connected component, or SCC, is a region of this graph that is as small as possible such that there are no dependency cycles between SCC's. In other words, an SCC encapsulates all cycles between rules within itself. Within an SCC, all the relations, because of their interdependence on one another, must be solved simultaneously. Each SCC is evaluated individually using SemiNaive Evaluation.

SemiNaive Evaluation

SemiNaive Evaluation is the method used to evaluate an individual SCC. Because of the ordering of the evaluation of SCC's, when this SCC is being evaluated, all relations

outside this SCC that this SCC depends on have already been computed. SemiNaive evaluation is a type of fixpoint evaluation that can solve a set of rules that may contain inter-dependencies, or recursion.

3.4.3 Pipelining

Pipelining is essentially top-down evaluation as in Prolog. When evaluating a module, the first rule in the list associated with the queried predicate is evaluated. This could involve recursive calls on other rules within the module (which are also evaluated in a similar pipelined fashion). If the rule evaluation of the queried predicate succeeds, the state of the computation is frozen, and the generated answer is returned.

3.4.5 Annotations

The user is able to provide hints or annotations to control optimization and evaluation. CORAL has a default execution environment setting for the module that does not specify an annotation. The user can use annotations to control optimization.

In CORAL, there are a number of annotations to be used by users. They are classified into several groups:

Rewriting techniques: If materialization (Module level control) is chosen, it needs source-to-source rewriting of the user's program. The default rewriting technique is Supplementary Magic Templates [2]. CORAL also supports other rewriting techniques, such as Magic Templates [2], Supplementary Magic With GoalId Indexing [2], and Context Factoring [2]. The user can use annotations to overwrite the default rewriting technique.

Following annotations could affect the method of the rewriting techniques.

@no_rewriting: Perform no transformations.

@sup_magic : Perform supplementary magic rewriting (Default).

@magic: Perform magic rewriting.

@factoring: Perform context rule rewriting.

@naive_backtracking: Use naive backtracking.

Module level control: at the level of module, a number of choices exist. Basically, materialization or pipelining are used, and there are other strategies.

@pipelining: Execute using pipelining.

@ordered_search: Execute using ordered search.

@monotonic: Modifies treatment of negation and grouping.

@lazy_eval: Computes answers in a lazy fashion (returns answers at end of each iteration).

@multiset: Execute with multiset semantics.

@check_subsumption: Do subsumption checks on all predicates.

@index_deltas: Create indexes on delta relations.

@return_unify: Perform return-unify optimizations for non-ground terms.

@interactive_mode: Return answers one at a time and prompt the user for more answers.

Predicate level control: there are some annotations at the level of predicates in the module. These annotations could affect the set of answers returned to a query. Following are the description:

@make_index: Index on specified adorned predicate

@allowed_adornments: Adornment algorithm tries to use only these adornments

@check_subsumption: Subsumption checking on specified adorned predicate

@multiset : Specified adorned predicate treated as a multiset predicate

@aggregate:_Selection modifies duplicate elimination by specifying tuples to retain

@aggsel_per_iteration: The selection is only done once per iteration

@aggsel_multiple_answers: The selection allows multiple answers in each grouping

@prioritize prioritize: Use of facts in a derived relation

3.4.6 An Example using optimization

The following example illustrates the effect of using optimization of predicate level control.

Sometimes, it is sufficient to compute one answer to a query. Aggregate selections are very useful in dealing with such queries. In CORAL, the use of aggregate selection is taken advantage of to avoid further computation after an answer has been found. The program in Figure 3.5 illustrates the use of aggregate selection for single-answer queries. In Figure 3.5, the module example1 defines a predicate that uses the annotation @aggregate-selection.

In line 3, there is an annotation "@aggregate—selection p(X,Y) (X) any(Y).". The annotation says that for each value of X, at most one fact p(X,Y) needs to be retained. If more than one fact p(X,Y) is generated by the program for any X, the system arbitrarily picks one of the facts to retain, and discards the rest.

In line 18, the user makes a query "? p(1,X)", the computation stops after the first iteration over the recursive rule. The number of answer is 1.

```
1   ready>>module example1.
2   ready>>export p(bf).
3   ready>>@aggregate—selection p(X,Y) (X) any(Y).
4   ready>>p(X,Y) :- b(X,Y).
5   ready>>p(X,Z) :- b(X,Y), p(Y,Z).
6   ready>>end—module.
7   ready>>b(1,2).
8   ready>>b(1,3).
9   ready>>b(1,4).
10  ready>>b(2,5).
11  ready>>b(5,6).
12  ready>>b(6,7).
13  ready>>list_rels.
14
15   b/2        (base)
16   p_bf/2     : (derived) : indexes
17
18  ready>>?p(1,X).
19  X=4.
20          ... next answer ? (y/n/all)[y]y
21  (Number of Answers = 1)
```

Figure 3.5: The definition of module example1 and its execution

In contrast, Figure 3.6 shows the module example2 that defines a predicate in the same way as the module example1, except there is no annotation "@aggregate-selection". To answer the query "?p(1,X)" in line 15 of Figure 3.6, CORAL generates all the answers (the number is 6), though only one answer is demanded.

```
1   ready>>module example2.
2   ready>>export p(bf).
3   ready>>p(X,Y) :- b(X,Y).
4   ready>>p(X,Z) :- b(X,Y), p(Y,Z).
5   ready>>end—module.
6   ready>>b(1,2).
7   ready>>b(1,3).
8   ready>>b(1,4).
9   ready>>b(2,5).
10  ready>>b(5,6).
11  ready>>b(6,7).
12  ready>>list_rels.
13  b/2         (base)
14  p_bf/2    : (derived) : indexes
15  ready>>?p(1,X).
16  X=4.
17          ... next answer ? (y/n/all)[y]y
18  X=3.
19          ... next answer ? (y/n/all)[y]y
20  X=2.
21          ... next answer ? (y/n/all)[y]y
22  X=5.
23          ... next answer ? (y/n/all)[y]y
24  X=6.
25          ... next answer ? (y/n/all)[y]y
26  X=7.
27          ... next answer ? (y/n/all)[y]y
28  (Number of Answers = 6)
```

Figure 3.6: The definition of module example2 and its execution

Chapter 4 Architecture of CORAL

CORAL is designed primarily as a single-user database system. The architecture of CORAL is shown in Figure 4.1. CORAL consists of four components: User Interface, Query process system, Data Management system and Persistent data storage management system.

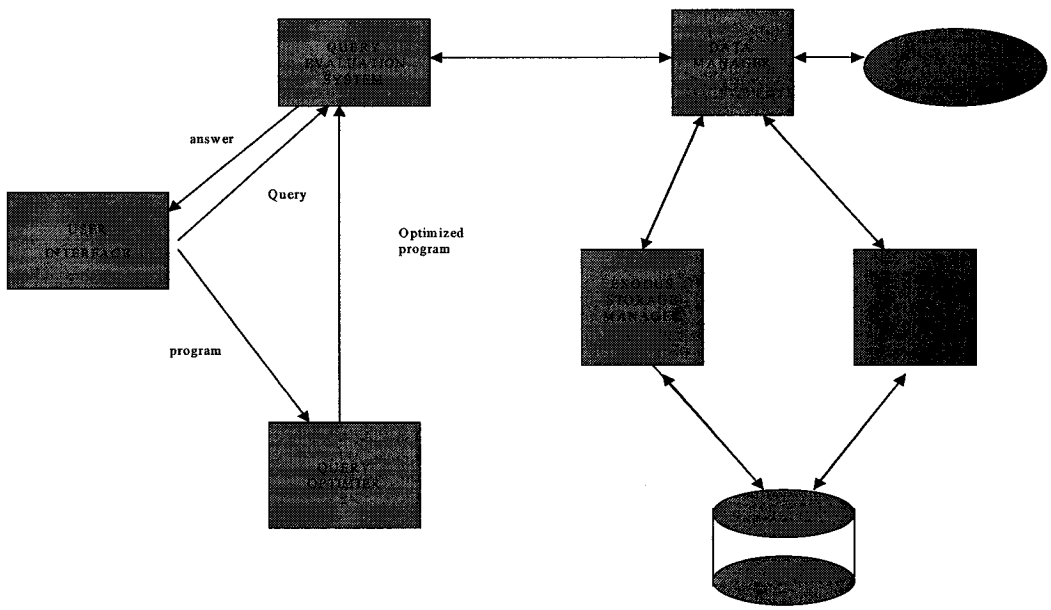

Figure 4.1: Architecture of CORAL [1]

4.1 Component Description

4.1.1 User Interface

The User Interface subsystem receives a user request, analyses it and decides which part of query process system to pass it to. It also shows the results of the user request. The inputs of the user can be classified into two kinds: imperative commands and declarative view definitions. Imperative commands fall into the following categories: queries, assign rules, append rules and delete rules. Declarative view definitions are made within a module.

These two kinds of input need different processing by the query processing system. The User Interface subsystem checks the user input and then lets the query process system performs the appropriate processing.

4.1.2 Query processing system

The query processing system consists of two main parts: a query optimizer and a query evaluation system. There are two kinds of query. One kind is a simple query, such as selecting facts from a base relation. These queries can be typed in at the user interface and can be processed by the Query Evaluation system directly. Anther kind of query is a complex query. These queries are typically defined in declarative `program modules' that export predicates with associated `query forms' (i.e. specifications of what kinds of queries, or selections, are allowed on the predicate). A complex query must be processed by the Query Optimizer first. Then the query is finished by the Query Evaluation system.

The Query Optimizer takes a program module and a query form as input, and generates a rewritten program that is optimized for the specified query forms. In addition to doing rewriting transformations, the Query Optimizer adds several control annotations (to those, if any, specified by the user). After the Query Optimizer's processing, the rewritten program is stored in a text file and is converted into an internal representation that is used by the Query Evaluation system.

The Query Evaluation system takes as input annotated declarative programs (in an internal representation), and database relations. The annotations in the declarative programs provide execution hints and directives. The Query Evaluation system interprets the internal form of the optimized program following the annotations. It then generates a result and sends back to user through the User Interface.

The Query Evaluation system has a well defined `get-next-tuple' interface with the data manager for access to relations[2]. The relations may be defined in different ways, such as base relations, declaratively through rules, or through system (Builtin relations). This interface is independent of how the relation is defined. The Query Evaluation system allows the different modules to be evaluated using different strategies.

4.1.3 Data management system

The data manager subsystem is responsible for maintaining and manipulating the data in relations. CORAL also has an extensional database interface. Through this interface, CORAL can communicate with relational databases and manage the extensional database. It can let CORAL connect with a relational database, map relational tables as CORAL relations, and process data interactively between CORAL and the relational database system.

4.1.4 Persistent data storage management system.

Persistent data is stored either in text files, or using the EXODUS storage manager which has a client-server architecture. The data stored in the text files can be "consulted", that means the data can be converted into main memory relations.

CORAL uses the EXODUS storage manager to support persistent relations. Currently, tuples in a persistent relation are restricted to have fields of primitive types only. EXODUS uses a client-server architecture; CORAL is the client process, and maintains buffers for persistent relations. If a requested tuple is not in the client buffer pool, a request is forwarded to the EXODUS server and the page with the requested tuple is retrieved.

In this report, we mainly discuss the management of data in main memory.

4.2 Component interactions

The components interact with each other to control the system and to pass data through the system. The following list of interactions describes how the components interoperate to satisfy the requirements of CORAL.

- The User Interface sends event and associated data (user program, to define module) to the Query Optimizer.
- The Query Optimizer receives data from the User Interface.
- After rewriting and compiling, the Query Optimizer sends the optimized program to the Query Evaluation system.

- The User interface sends events and data (imperative commands) to the Query Evaluation system.
- The Query Evaluation system receives event and data from the User Interface.
- The Query Evaluation system gets data from the Query Optimizer and the Data Manager. After processing, it sends back results to the User Interface.
- The User Interface receives the results from the Query Evaluation system and shows them to the user.

Chapter 5 Major Data Structures in CORAL

This section discusses how CORAL represents major data types, such as argument, term and relation. It also describes the structures of some important classes.

5.1 Representation of Argument

CORAL uses the Cor_CArg class to represent an argument. It is a basic class that can hold a number, string, list, variable, set or other implemented type.

Using Cor_CArg, CORAL implements many other things: functors are implemented as a name and an arglist(a list of Cor_CArg); lists are implemented via a CargStack, which holds the items of the list in reverse-order; and Sets are implemented as a CArg holding a pointer to a unnamed Relation. Duplicate elimination is turned on for sets and off for multisets; aggregates are implemented as using some attributes of CArg to define aggregating types and using an attribute to hold the result after evaluation. Following are the descriptions of these attributes:

SUM_GRPING_ARG: sum(<X>)

PROD_GRPING_ARG: prod(<X>)

CNT_GRPING_ARG: count(<X>)

AVG_GRPING_ARG: avg(<X>)

STDDEV_GRPING_ARG: stddev(<X>)

SUM_DGRPING_ARG: sum_distinct(<X>)

PROD_DGRPING_ARG: prod_distinct(<X>)

CNT_DGRPING_ARG: count_distinct(<X>)

AVG_DGRPING_ARG: avg_distinct(<X>)

MIN_GRPING_ARG : min(<X>)

MAX_GRPING_ARG: max(<X>)

SET_GRPING_ARG: distinct(<X>)

ANY_GRPING_ARG: any(<X>)

NO_AGG_GRPING_ARG: <X>

USR_DEFD_GRPING_ARG: User-defined grouping

AGG_RESULT_ARG: Aggregate result some types of groupings

5.2 Representation of terms

Simple terms are constant and variables. Complex terms are constructed using functors.

A functor is represented by a record containing: the function symbol; an array of arguments, or pointers to the arguments and extra information unification of such terms efficient.

A Term is an argument (Cor_CArg), along with a BindEnv that may bind some of the variables within the argument. A BindEnv maps each variable to a Term, so these structures are mutually recursive, allowing for multiple levels of binding.

In CORAL, a BindEnv is a binding environment. For a set of variables, it indicates what they are bound to. Therefore, whenever a variable is accessed during an inference, a corresponding BindEnv must be accessed to determine if the variable has been bound.

Figure 5.1 is an example of the term f(X,10,Y), where X is bound to 25, Y is bound to Z, and Z is bound to 50 in a separate BindEnv.

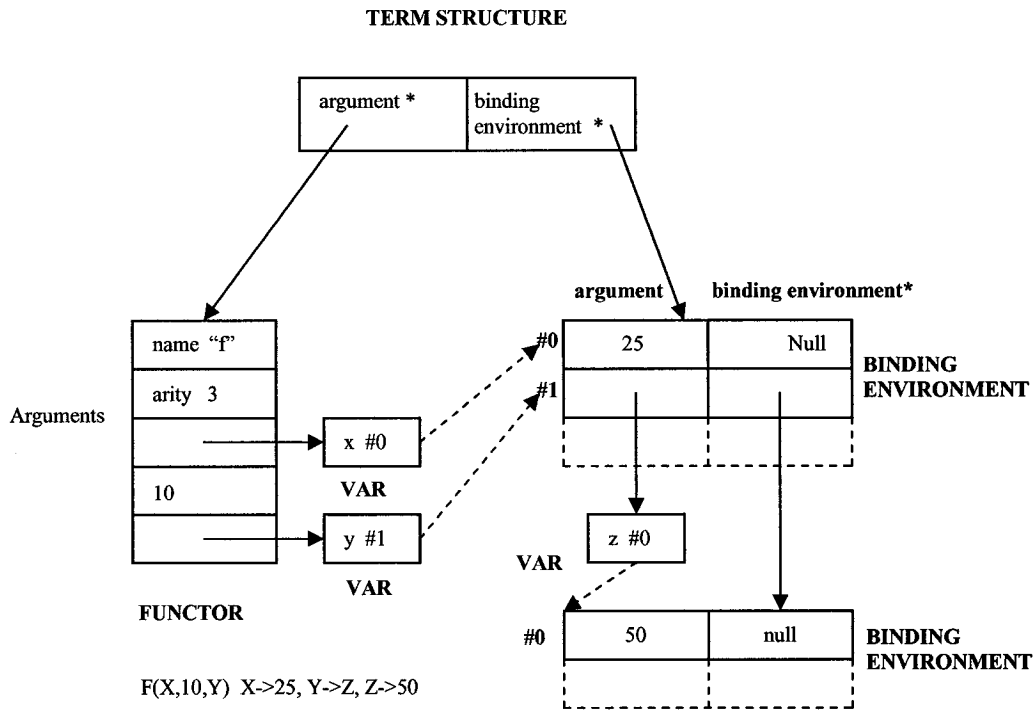

Figure 5.1 Representation of a term [2]

5.3 Representation of Tuples

In CORAL, an in-memory relation is composed of tuples. The class Cor_Tuple to describe to structure of tuple. The class Cor_Tuple has an attribute Cor_ArgList and some public flags. Cor_ArgList is used to hold an array of terms. Each element of Cor_ArgList is represented by a Cor_CArg type.

5.4 Representation of relations

An object of the class relation is a collection of tuples. CORAL uses an abstract class that provides common features of relation. There are also several subclasses of relation. Cor_Relation has some virtual functions, including insert_tuple(tuple : Cor_Tuple*) , delet_tuple(tuple : Cor_Tuple*), etc. It also uses an iterator interface that allows tuples to be returned from the relation. The class Cor_TupleIterator is used to implement this interface. It is used to store the state or position of a scan on the relation, and to allow multiple concurrent scans over the same relation.

CORAL relations support the ability to get marks into a relation, and distinguish between facts inserted after a mark was obtained and facts inserted before the mark was obtained. A mark is a mechanism that makes it possible to recognize which tuples in the relation were added after the mark was obtained, and which tuples were already in the relation before the mark was taken [1]. This feature is important for the implementation of all variants of semi-naive evaluation. The implementation of this extension involves creating subsidiary relations, one corresponding to each interval between marks, and transparently providing the union of the subsidiary relations corresponding to the desired range of marks. The attribute Rmark is used to select a sub-range of a relation. The range (RM1, RM2) of relation R is the set of tuples in R that were added to R after RM1 was created but before RM2 was created.

Either RM1 or RM2 can be NULL, which is equivalent to the beginning or end of a relation. An RMark is created using 'relation->getMark()'.

5.5 The TupleIterator Structure

As mentioned in 5.4, the class Cor_TupleIterator is used to get tuples from a relation. Figure 5.2 describes the main structure of this class. It contains the information about the relation, state or position of a scan on the relation, and other information, such as the type of the relation.

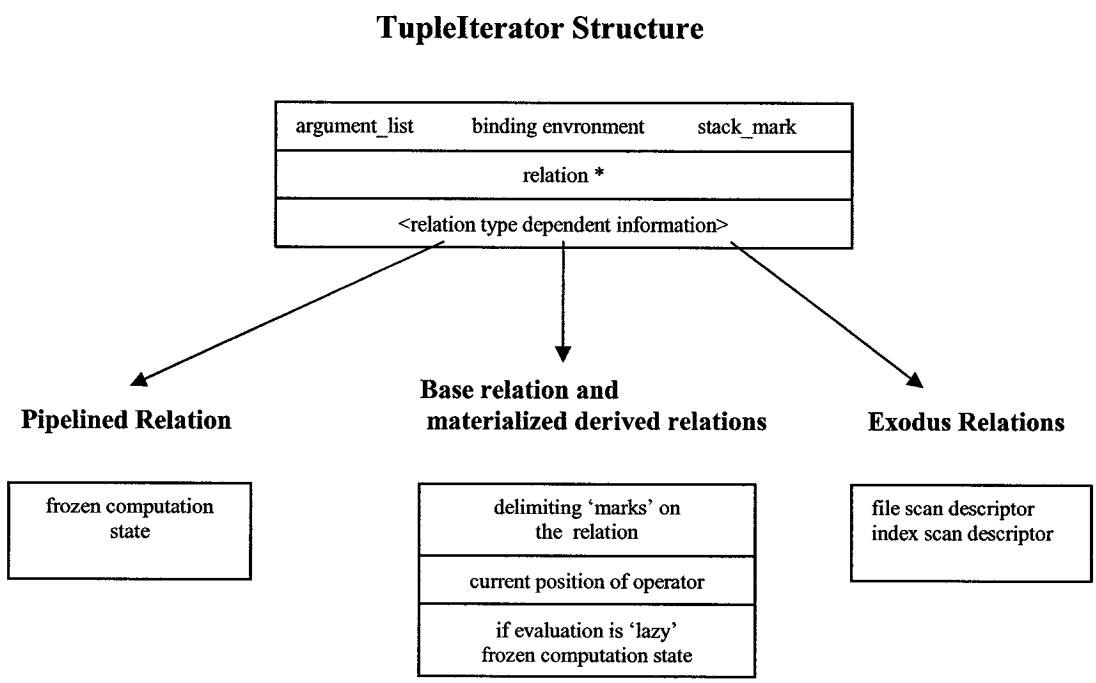

Figure 5.2: The TupleIterator Structure [1]

5.6 Modules and Rule data structures

The compilation of a materialized module generates an internal module structure that consists of a list of structures corresponding to the strongly connected components (SCCs) of the module, and each SCC structure contains structures corresponding to semi-naive rewritten versions of rules. These semi-naive rule structures have fields that specify the arguments of each body literal, and the predicates that they correspond to. Each semi-naive rule also contains evaluation order information, pre-computed backtrack points, and pre-computed offsets into a table of relations. Figure 5.3 shows these structures.

If a module is to be evaluated using pipelining, it is stored as a list of predicates defined in the module. Associated with each predicate is a list of rules defining it (in the order they occur in the module definition), each rule being represented using structures like those used for semi-naive rules.

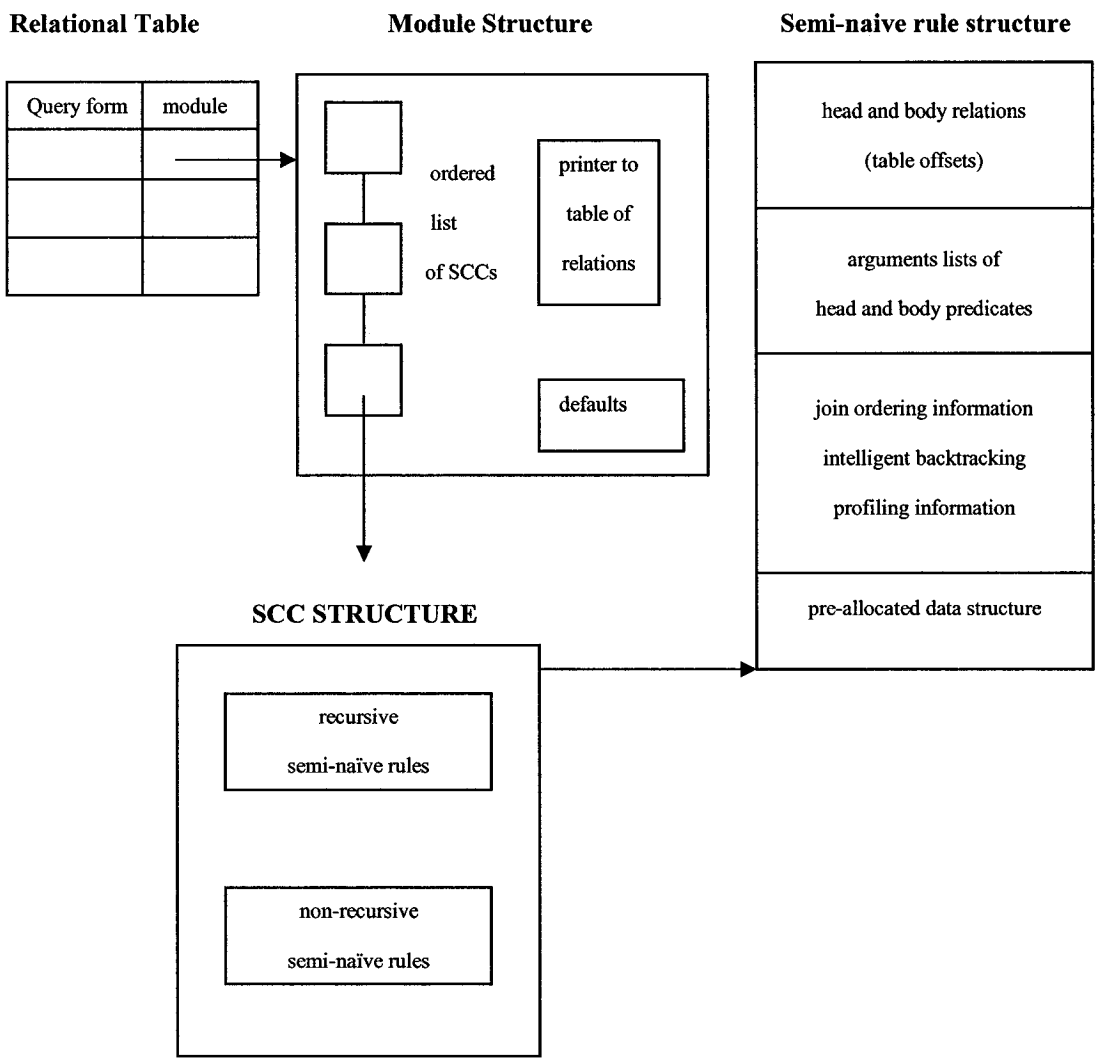

Figure 5.3: Runtime Data Structure[1]

Chapter 6 Structural Model

6.1 Subsystems

There are five subsystems in the CORAL as shown in Figure 6.1. They are: UserInterface, Optimizer, Evaluator, DataManager and Common. Each subsystem consists of several modules or classes.

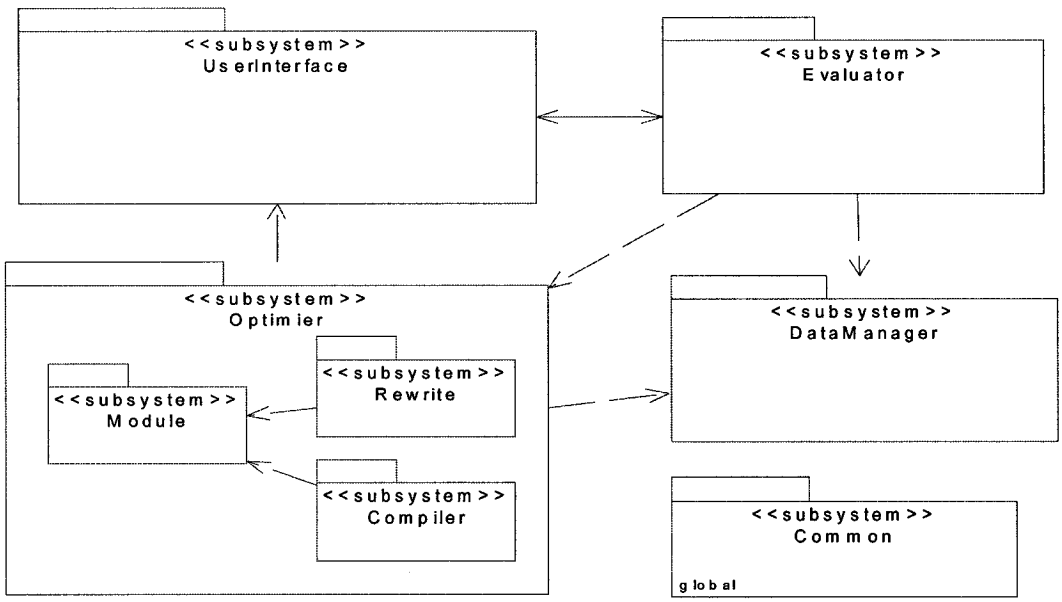

Figure 6.1: Class Diagram for Subsystem Collaboration

6.2 UserInterface Subsystem

The principal responsibility of the UserInterface subsystem is to provide the services for the user input. A User command are accepted and checked by the Cor_Scanner, then they are passed to the Cor_Parser. The Cor_Parser uses the class Cor_ParserStruct to store whatever it is reading in. All CORAL constructs (rules, queries, facts, etc.) end with a period. Each object of Cor_ParserStruct corresponds to one construct. However, an entire module is treated as one object. The parser operates as follows :

(a) Read in an entire Cor_ParserStruct

(b) Process it and take some actions that are defined in the class Cor_ParserActions.

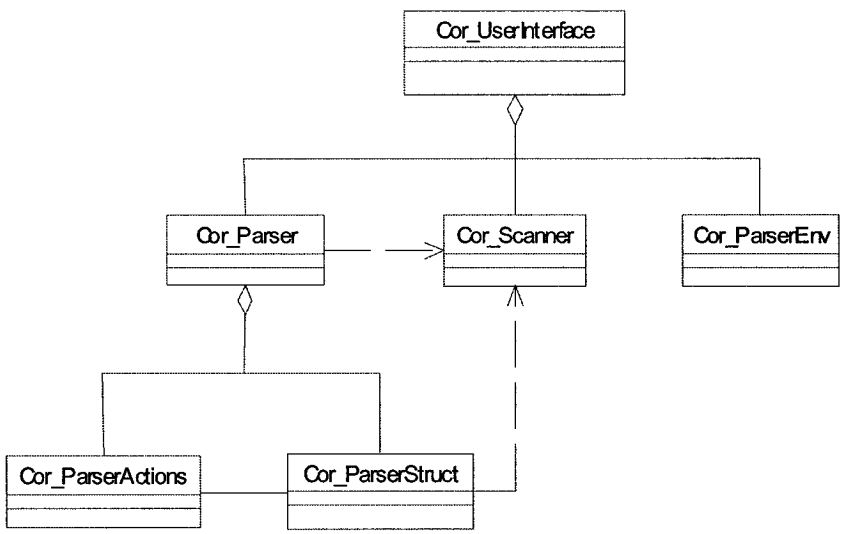

Figure 6.2: Class Disgram for UserInterface Subsystem

6.3 Optimizer Subsystem

The Optimizer subsystem consists of three subsystem: ReWrite subsystem, Compiler subsystem and Module subsystem.

6.3.1 Rewrite subsystem

The Rewrite subsystem is responsible to produce a transformed module in which the rules have been rewritten for optimization. Class Cor_Rewrite performs the rewriting process following the rewriting annotation that the user put in the module (if there is any), otherwise it will use the default rewriting technique. The class FilePrinters defines different methods to rewrite different parts of the module (such as rule, predicate, annotation, etc.).

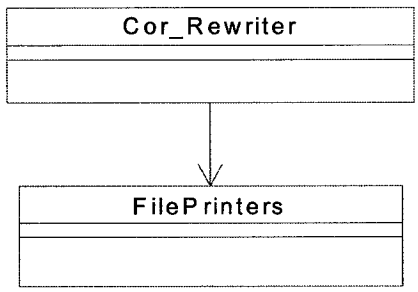

Figure 6.3: Class Diagram for Rewrite Subsystem

6.3.2 Module subsystem

The Module subsystem includes all the information for a CORAL module. Figure 6.4 shows the main classes in this subsystem.

They are:

Cor_Predicate: is to define a predicate that is identified by its name and arity.

Cor_Clause: is to define a rule, a rule structure has created by the parser.

Cor_ExportInfo: holds compile-time information about an exported predicate-adornment pair.

Cor_ModuleInfo: holds the information of a module includes name of module, list of

exported queries(predicates with adornments) and the information of evaluation strategy for this module. Figure 6.5 shows the details of class Cor_ModuleInfo.

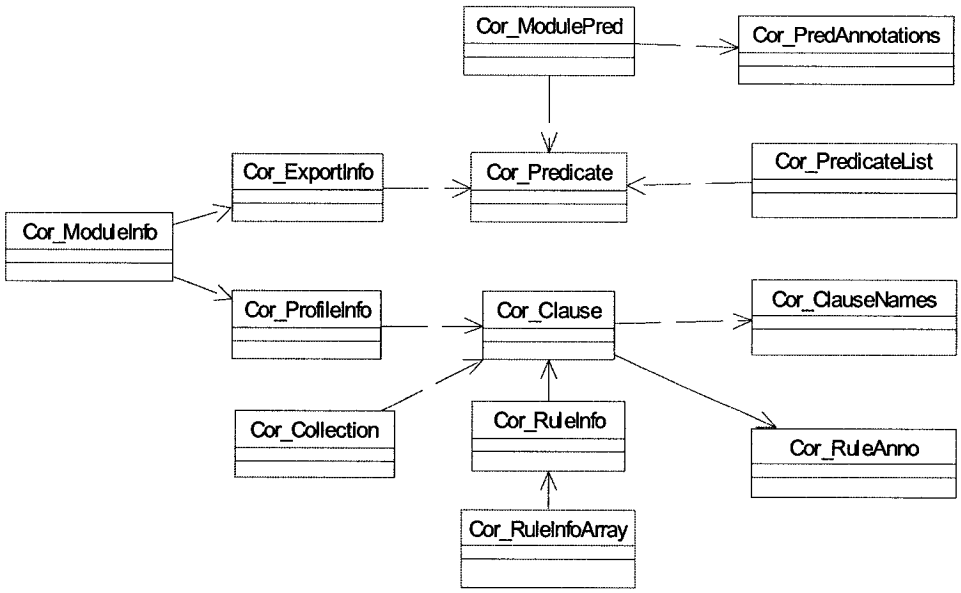

Figure 6.4: Class Diagram for Module Subsystem

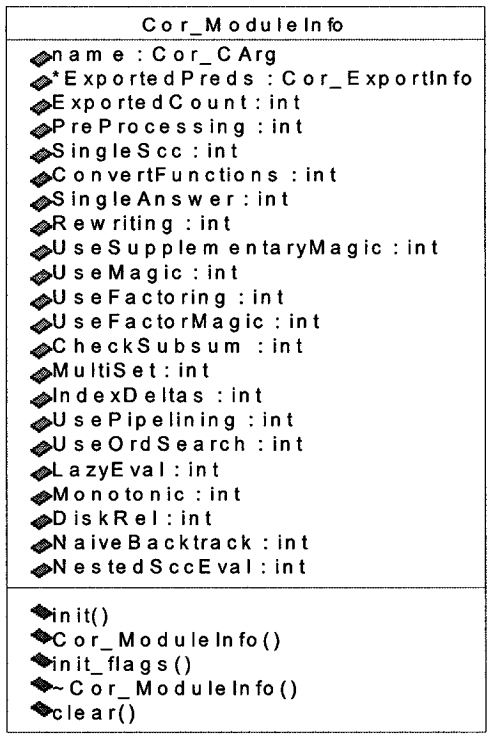

Figure 6.5: Class Diagram for class Cor_ModuleInfo

6.3.3 Compiler subsystem

In the Compiler subsystem, the class compiler does the following tasks: For each exported query form, a QFModuleData structure is created that encapsulates the information needed to evaluate that query. The exported query form and the associated Cor_QFModuleData structure are stored in a table of IterativeMethods. The Class Cor_QFModuleData is the intermediate structure that a module is compiled into. Each module is compiled into a Cor_QFModuleData structure. Cor_QFModuleData has two subclasses: one for materialized modules, and one for pipelined modules.

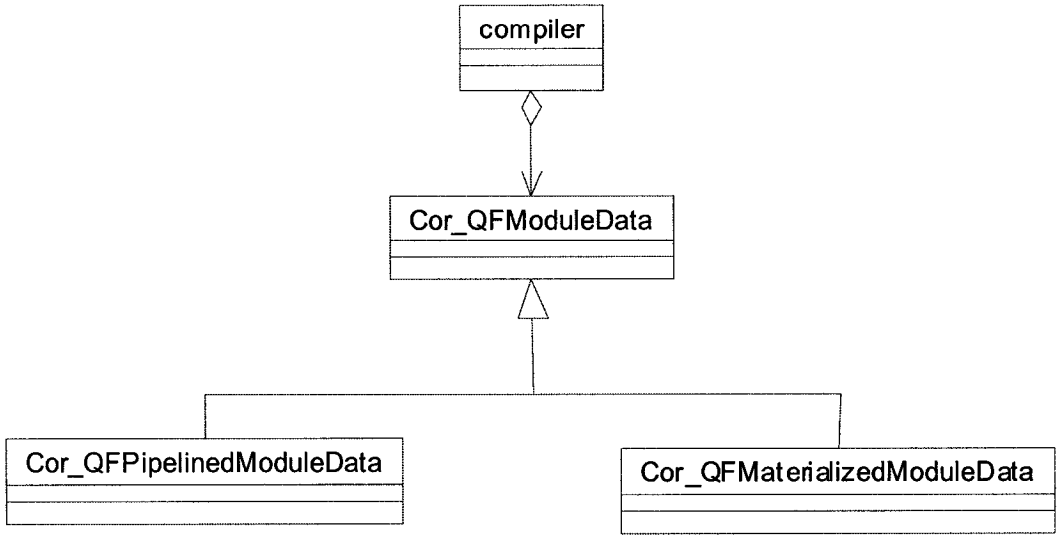

Figure 6.6: Class Diagram for Compiler Subsystem

Figure 6.7 shows detail of class Cor_QFMaterializedModuleData and its associated classes. For materialized modules, the predicates in a module need to broken into strongly connected components or SCCs. Within each SCC, the rules corresponding to the predicates need to be prepared to be evaluated in a seminaive fashion. That is, the Class Compiler uses the function Cor_interpret_module() to do following processing:

1. Performing a SCC analysis on the predicates in the module.
2. For each SCC, an SccInfo structure is created corresponding to the SCC. Each SccInfo struct contains an array of seminaive rules, as well as a number of additional fields and arrays that are used for optimizing execution. The function of class Cor_SccInfo: CreateSNStructs() creates of the seminaive structures.
3. Every SemiNaive rule has a pointer to a corresponding RuleInfo (or is it an offset into the RuleInfoArray) that specifies the syntactic rule.

4. After the Cor_QFMaterializedModuleData is finished, it is inserted into the DerivedRelation table as the ModuleData structure corresponding to the exported query form.

Figure 6.7 Class diagram for Cor_QFMaterializedModuleData

Figure 6.8 shows detail of Class Cor_QFPipelinedModuleData and its associated classes.

A pipelined module is one that has the annotation "@pipelining" inside the module. The compilation of a pipelined module results in the creation of a FPipelinedModuleData structure that represents the module. There are no SCCs. Instead of SemiNaive rules in the compilation of materialized module, structures called PipelinedExecInfos (PEIs) are created. These PEIs mainly contain backtracking information and the offset of the corresponding predicate in the Predicates array.

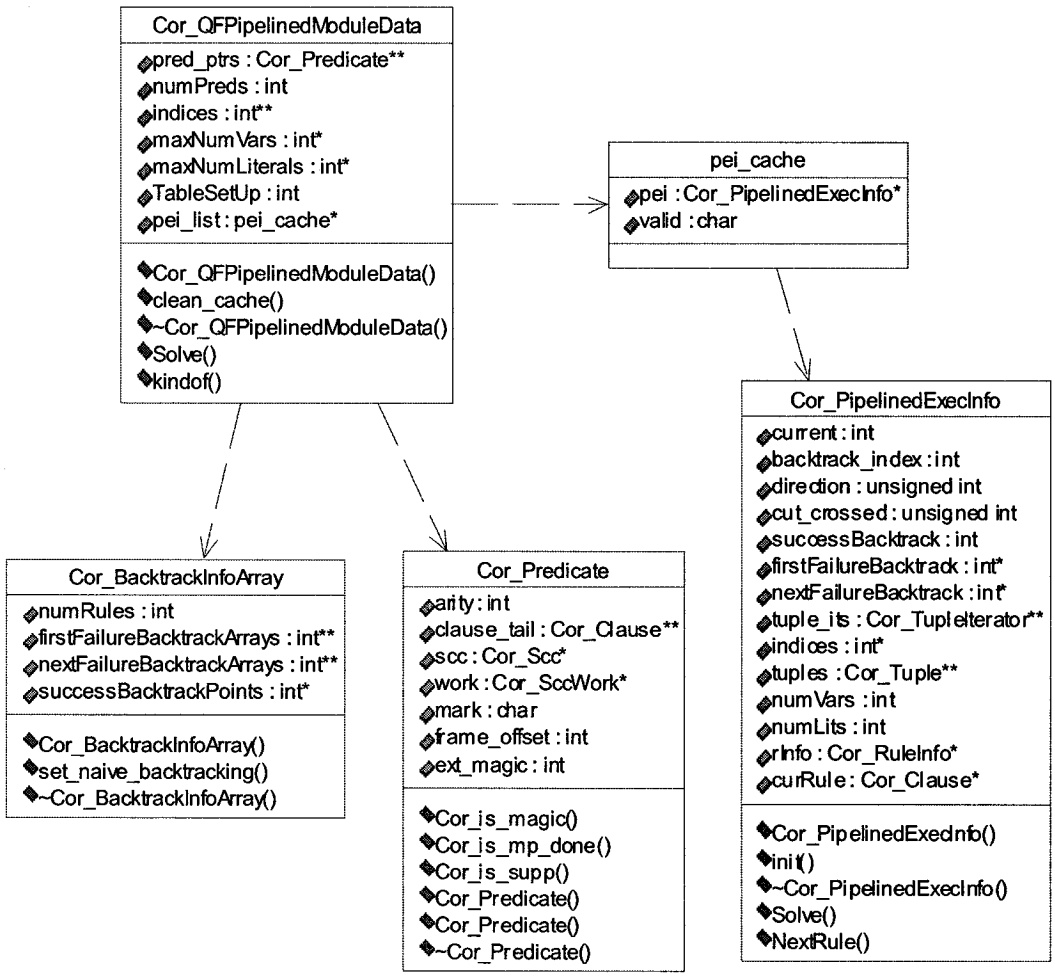

Figure 6.8: Class diagram for Cor_QFPipelinedModuleData

6.4 Evaluator subsystem

The Evaluator subsystem is responsible for performing the user's request. From the CORAL prompt, imperative commands may be specified. They fall into the following categories:

1. Queries
2. Assign rules
3. Append rules
4. Delete rules

These rules are evaluated using the same evaluation code as rules within a module. This is important, because the optimizations performed for rules within a module are applied to imperative rules as well. During evaluation, a seminaive rule structure is created from the rule typed in at the prompt. This rule is now evaluated using the SemiNaive::evaluate method. In order to do this however, some initial setup needs to be performed so that the evaluate code can work for both imperative rules, and rules within modules.

Append rules are treated similar to rules within modules. For assign rules, the contents of the head relation are first deleted, and the rule is then applied. For delete rules, the head relation is first changed to a temporary relation. All the tuples to be deleted are collected in this temporary. Finally, they are deleted from the original head relation.

In the class Cor_evaluator, the function Cor_execute_single_rule() can be used to finish user's imperative command. The function Cor_do_query() is called at run time to execute a query for the module. Figure 6.9 shows the main classes in this subsystem.

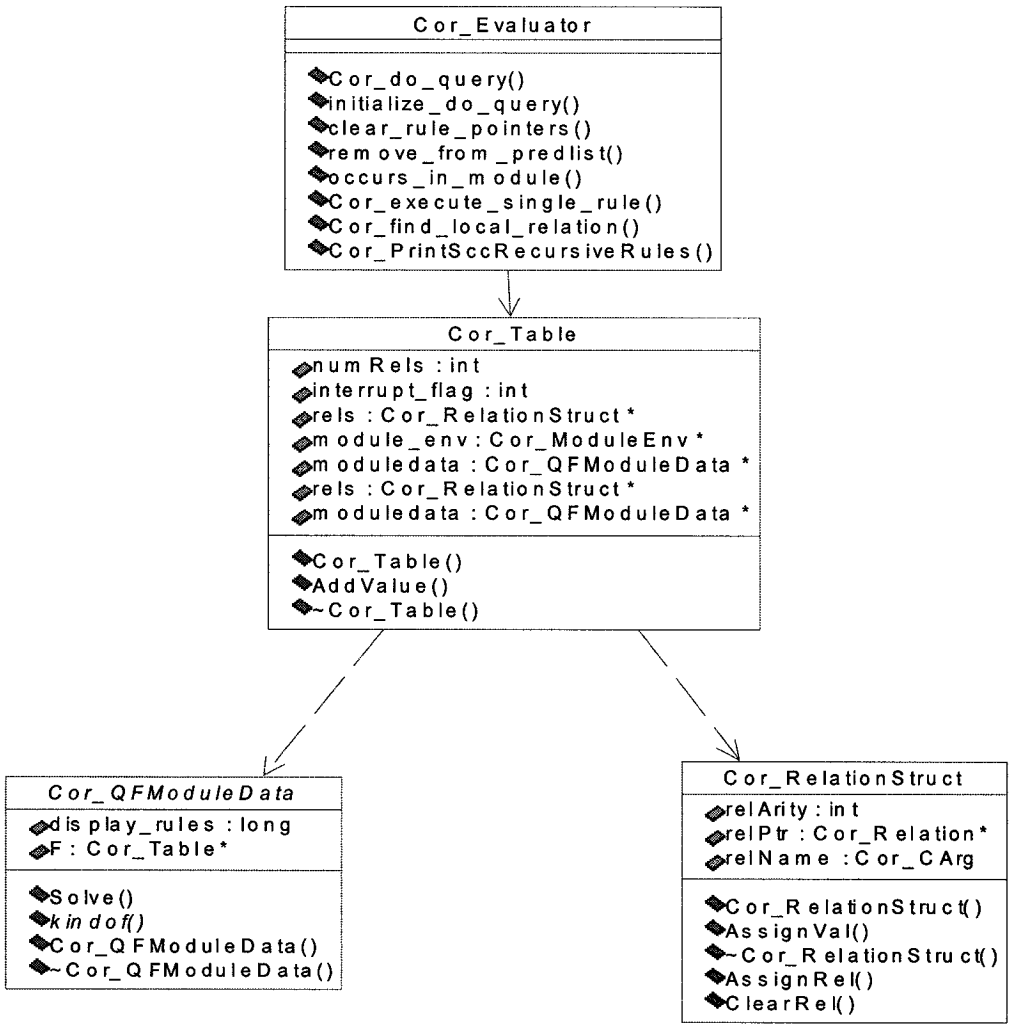

Figure 6.9: Class Diagram for Evaluator subsystem

6.5 Data Manager subsystem

The Data Manager subsystem is responsible to maintain and manipulate data in relations. Here we describe two main parts of Data Manager subsystem, one part is relation classes, another part is the classes for the interface of extensional relational database. Figure 6.10 shows the relation classes.

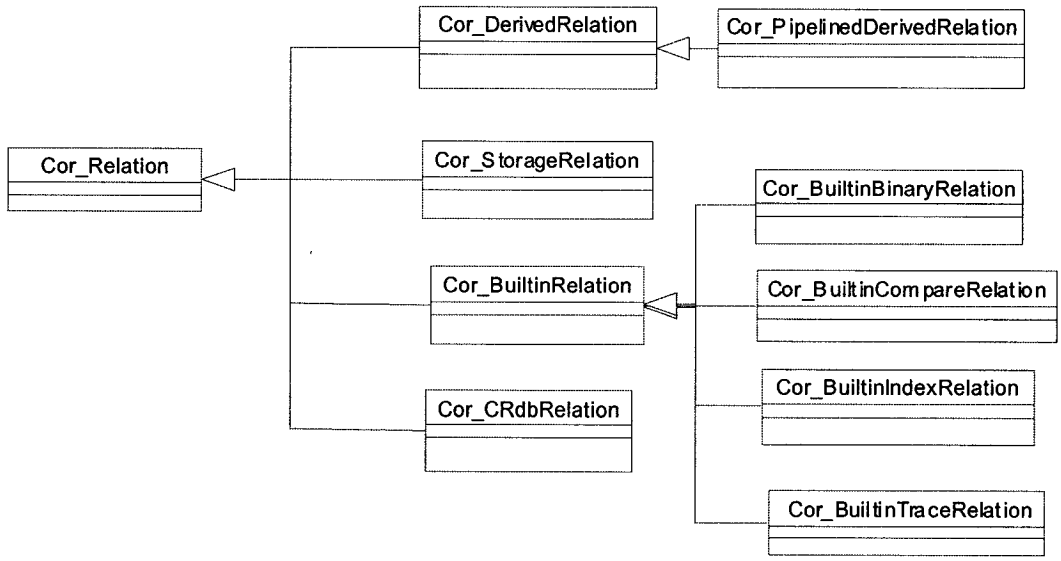

Figure 6.10: Class Diagram for Cor_Relation

The class Cor_relation is an abstract class that provides common features of relation. There are four kinds of subclass of Cor_relation:

1. The class Cor_StorageRelation is used to store in-memory regular relations.
2. The class Cor_DerivedRelation is used to store derived relation that is defined by module. After a user defined module has been optimized, an intermediate

structure (Cor_QFModuleData) is generated and is stored in a structure Cor_DerivedMethod. Class Cor_DerivedRelation has a pointer to it. The function call of get_next_tuple() may result in the module being evaluated. Figure 6.11 shows the detail of class Cor_ DerivedRelation.

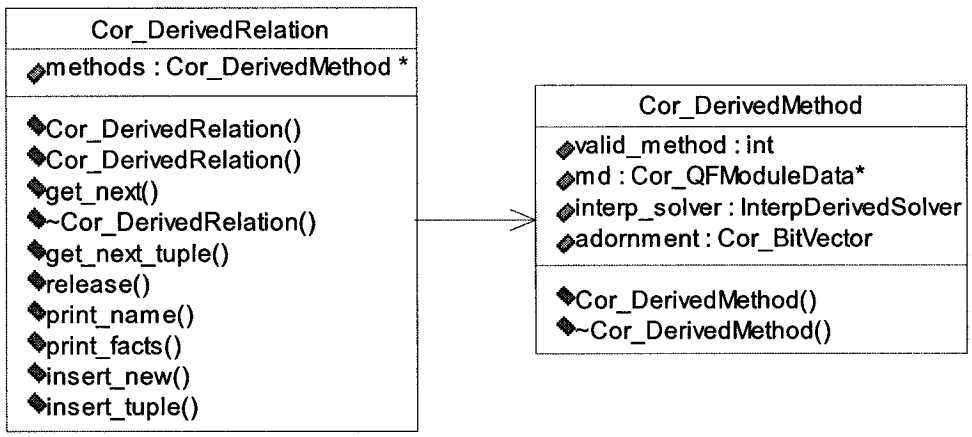

Figure 6.11 Class Diagram for Cor_DerivedRelation class

3. Cor_BuiltinRelation is used to store builtin Relations that are special kind of relations. In Coral, some of the user's input looks like a "command", but in fact, these commands are implemented as Builtin Relations. For example, the command "list_rels" is one Builtin Relation. This is a special subclass of Relation that presents the same get-next-tuple interface. However, for a Builtin Relation, get-next-tuple, or insert-tuple into the relation results in a command being executed.
4. Cor_CRdbRelation is for extensional database connections. In CORAL-1.5.2, the system supports the connection to an extension database, such as Sybase and Ingress.

The classes that implement the interface of an extensional database are called RDB classes, all the names beginning with "Rdb". Figure 6.12 shows the main RDB Classes inheritance hierarchy.

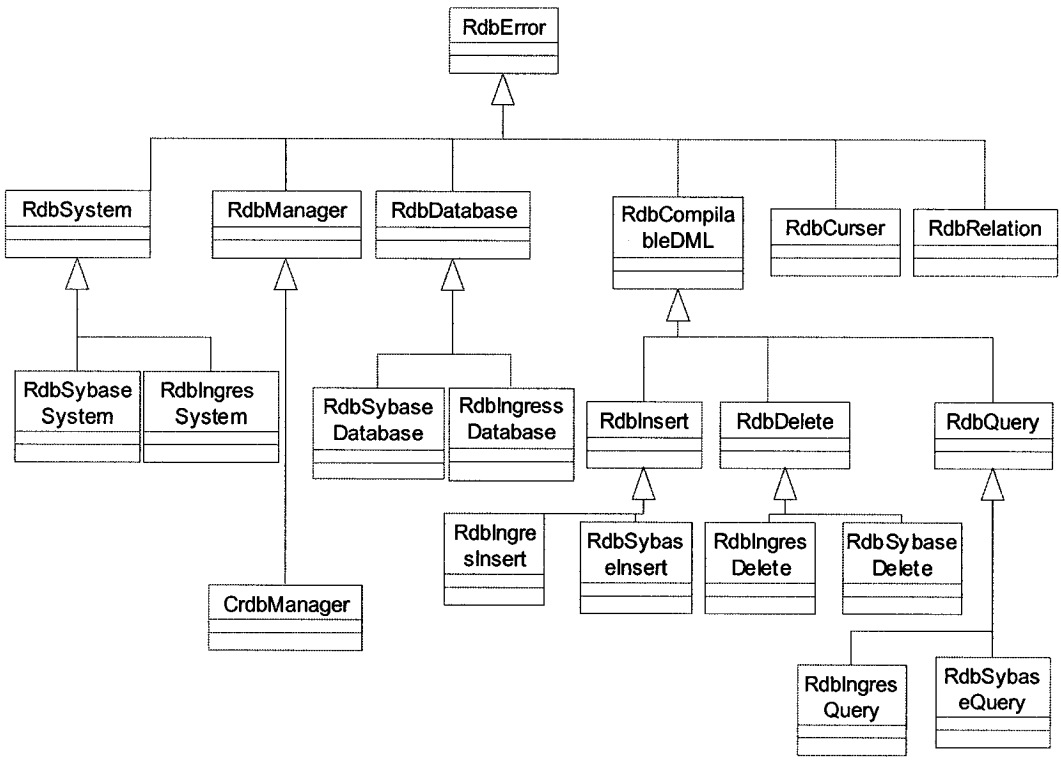

Figure 6.12: RDB Classes

RdbError : is used to manage all possible errors generated by RDB classes.

RdbManager: manages RdbSystems.

RdbSystem : is an abstract class that represents particular relational DBMSs. Now in CORAL, there are two DBMSs supported: Sybase and Ingres. So RdbSystem has two subclass: RdbSybaseSystem and RdbIngreSystem.

RdbDatabase: is an abstract class and provides a virtual interface for child concrete database classes that deal with the extensional databases. It has two subclasses: RdbSybaseDatabase and RdbIngresDatabase.

RdbRelation: stands for all connections to extensional tables. RdbRelation records the informatiom of the backend relation, such as name, column names, and the database in which it resides.

RdbCompilableDML: is a concrete class for any database statements that can be "prepared" and kept around for a while. These statements include insert, delete, and all queries.

RdbInsert : is a class to insert a tuple, it is a abstract class. It has two subclasses: RdbSybaseInsert and RdbIngresInsert.

RdbDelete : is a class to delete a tuple, it is a abstract class. It has two subclasses: RdbSybaseDelete and RdbIngresDelete.

RdbQuery : is a class to query on the table of backend database. It is abstract class. It has two subclasses:RdbSybaseQuery and RdbIngresQuery.

RdbCursor :is an iterator class that will be used during queries on the backend tables. It stores some state of the query.

6.6 Common Subsystem

The Common subsystem contains classes that are used very often. Every other subsystem has dependency to Common subsystem. Mainly it includes Cor_CArg, Cor_ExecutionEnv, Cor_ParserStruct, Cor_SymTabElement, Cor_DatabaseStruct; Cor_SymTable, and Cor_StrRefTable.

The class Cor_CArg is to store CORAL arguments. It can hold a number, string, variable, set or other implemented type.

The class Cor_ExecutionEnv stores the system defaults. For example, It contains system defaults for optimization. Each module inherits these defaults, and they are used to process the module, unless other defaults are specified by the use of annotation.

The class Cor_ParserStruct is a structure to store whatever the parser is reading in. It can store a rule, a query, a fact or an entire module. It can be used to pass information among subsystems.

The class Cor_SymTabElement stores information of a workspace or a relation.

The class Cor_SymTable uses a hash table to store Cor_SymTabElement objects.

The class Cor_DatabaseStruct is used to define workspace. It stores either EDB relations or relations corresponding to predicates exported by modules.

The class Cor_StrRefTable is the reference Counting for strings. It handles the reference counting for strings and deletes them when the references drop to zero.

Chapter 7 Behavioral Model

This section describes the dynamic aspects of CORAL. Four basic scenarios are chosen. Sequence Diagrams are used to describe these scenarios.

7.1 Description of the scenario for Enter a fact to a relation

Brief description of the Scenario:

User input a fact to the system, CORAL searches the corresponding relation and insert the fact as a tuple to the relation. It involves following steps.

1. User inputs a fact at prompt.
2. Cor_Scanner checks if there is any thing illegal. If it is OK, it pass the data to Cor_Parser.
3. Cor_Parser analyses the data and make sure it is a fact. then it chooses an action "Cor_FactFunction" of Cor_ParserActions.
4. Cor_ParserActions checks the RelationTable of current workspace object Cur_DB, whether there is the relation whose name is same with the input fact. If not, it create a relation whose name is the same as the input fact and insert into the RelationTable. After a success message, it will insert the fact as a tuple into the relation.

The Sequence Diagram in Figure 7.1 shows these steps.

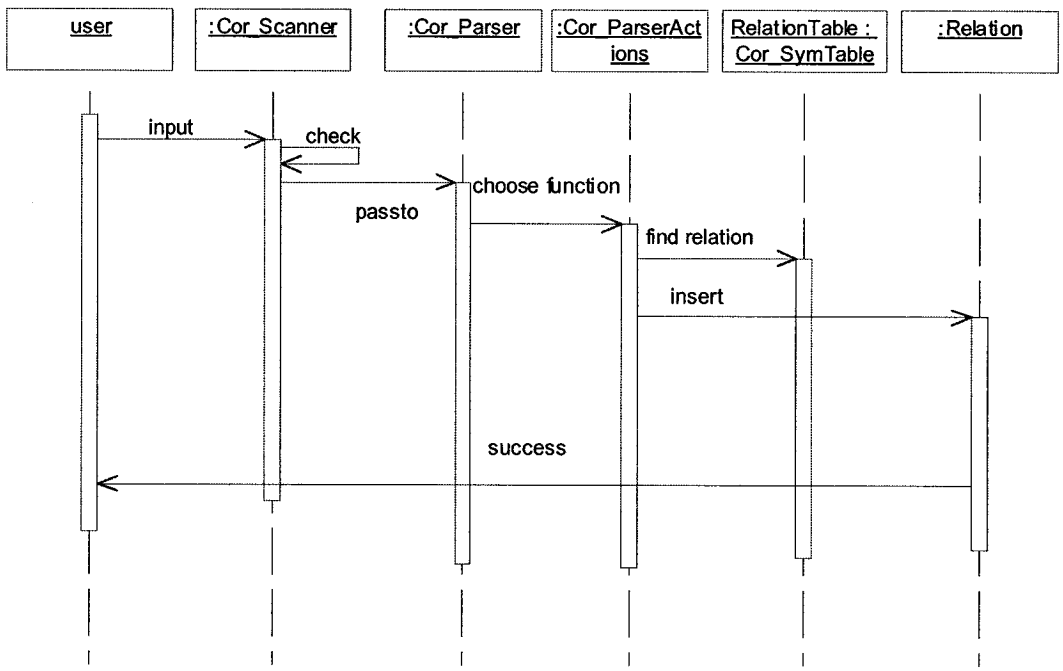

Figure 7.1: Sequence Diagram for the scenario Enter a fact

7.2 Description of the scenario for Input a module

Brief description of the Scenario:

User inputs a module at the prompt, after CORAL receives "end_module." of the module, it performs rewriting process and produces rewritten module. Then it compiles rewritten module.

Note: a module can be either input at prompt or stored in a file. A module stored in a file can be consulted by the system. In this scenario, we just discuss the first case: a module is input at prompt.

The process includes the following steps:

1. User input a module at prompt.
2. Cor_Scanner checks if there is any thing illegal. If it is OK, it passes the data to Cor_Parser.
3. Cor_Parser analyses the data and makes sure it receives "end_module.". Then it chooses an action "Cor_EndModuleFunction" of Cor_ParserActions.
4. Cor_ParserActions checks the PreProcessing of CurModule is "ON" or not.
5. If "ON", Cor_Rewriter calls Cor_perform_rewriting().

   Cor_Rewrite sends the rewritten module to Cor_Scanner.

   Cor_Scanner checks if there is any thing illegal. If it is OK, it passes the data to Cor_Parser.

   Cor_Parser analyses the data and make sure it receives "end_module.", then it chooses an action "Cor_EndModuleFunction" of Cor_ParserActions.

   Cor_ParserActions checks the PreProcessing of CurModule is "ON" or not.
6. If "OFF", Cor_Compiler calls Cor_interpret_module(), after Cor_Compiler finishes, it sends back user message.

Note: First time a module is processed by parser, CurModule->PreProcessing is "ON". After rewriting, it will be set to "Off". The rewritten module will be sent back to Cor_Scanner.

Figure 7.2 illustrates the above steps.

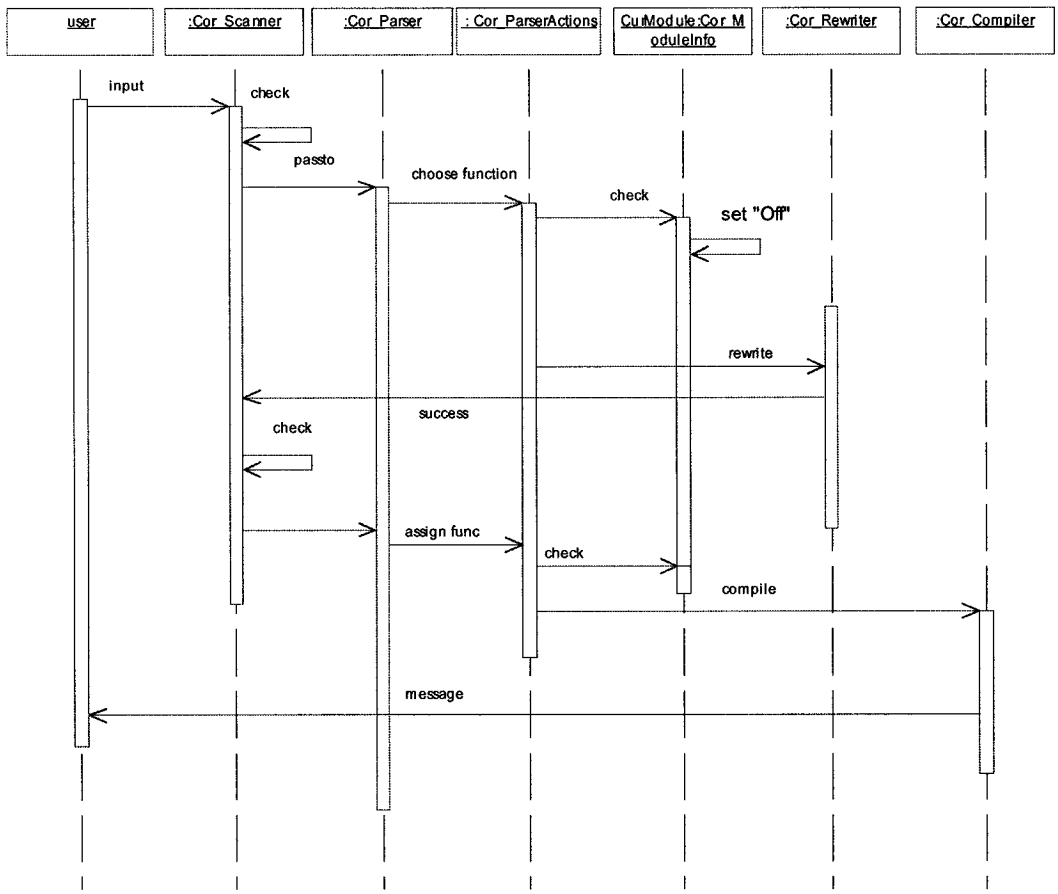

Figure 7.2: Sequence Diagram for the scenario Enter a module

7.3 Description of the scenario for Input a Builtin command

Brief description of the Scenario:

User inputs a Builtin command "consult(mytest.P)", CORAL treats it as the insert of the tuple "(mytest.P)" of arity 1 into the relation "consult". The insert_tuple method is called

on the relation "consult", and this results in the file "mytest.P" being consulted by the system. This process involves following steps.

1. User inputs a Builtin command "consult(mytest.P)" at prompt.
2. Cor_Scanner checks if there is any thing illegal. If it is OK, it passes the data to Cor_Parser.
3. Cor_Parser analyses the data and makes sure it is a fact, then it chooses an action "Cor_FactFunction" of Cor_ParserActions.
4. Cor_ParserActions checks the RelationTable of Cor_BuiltinDB, where there is the builtin relation whose name is "consult". If found, it inserts the tuple `(mytest.P)' of arity 1 into the relation `consult'.
5. Relation object "consult" executes its solver that will load the file into Cor_Scanner.

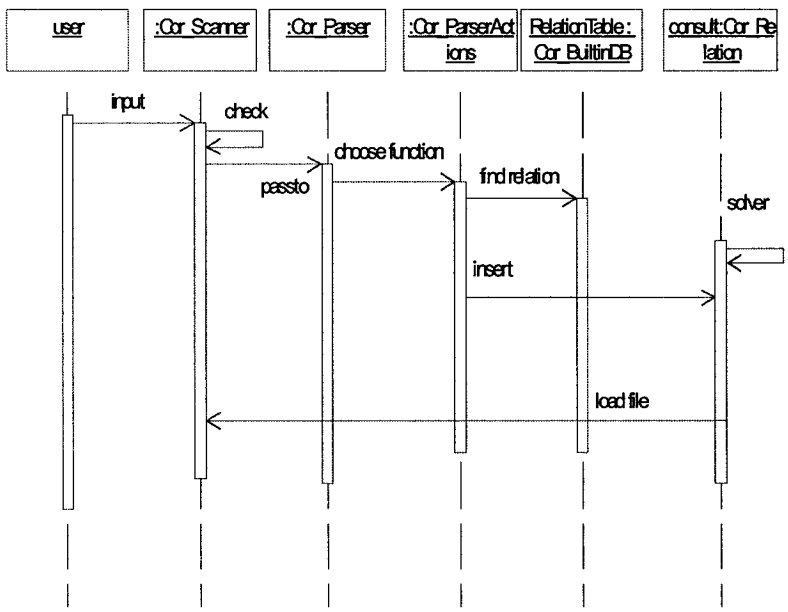

Figure 7.3: Sequence Diagram for the scenario Enter a Builtin command

7.4 Description of the scenario for Input a Query

Brief description of the Scenario:

User inputs a query at prompt, CORAL converts the query into a rule by adding a special predicate called print_bindings which is a Builtin relation. After that, CORAL compiles and evaluates the rule. The results of the query are printed by Builtin relation print_bindings.

The process includes following steps:

1. User inputs a query at prompt.
2. Cor_Scanner checks if there is any thing illegal. If it is OK, it passes the data to Cor_Parser.
3. Cor_Parser analyses the data and makes sure it is a query, then it chooses an action "Cor_QueryFunction" of Cor_ParserActions.
4. Cor_ParserActions uses function Cor_complete_rule() to converts the query into a rule by adding the name of print_bindings, which is a builtin relation.
5. Cor_ParserActions passes the data to Cor_Evaluator.
   Cor_Evaluator creates a Cor_SemiNaive object.
6. Col_compiler performs the function Cor_compile_single_rule() to compile this converted rule.
7. Cor_Evaluator sends an evaluate() message to Cor_SemiNaive. It causes a tuple to be generated and inserted into the builtin relation print_bindings.
8. The insertion of a tuple to print_binding causes the method solver() of this builtin relation to be executed. That is, to print the tuple to the user.

The Sequence Diagram in Figure 7.4 describes above steps.

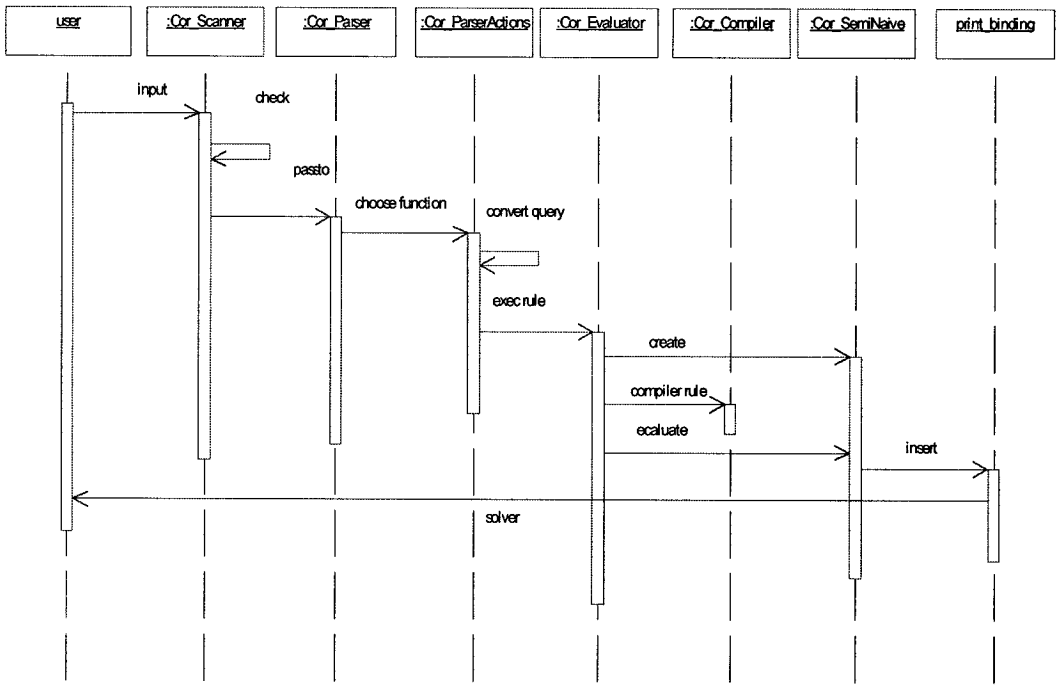

Figure 7.4: Sequence Diagram for the scenario Enter a query

Chapter 8 Conclusion

This report provides documentation for the design of CORAL. UML is used to describe CORAL's architecture, class diagram and basic behavior during execution. The document uses the following parts to illustrate the design of CORAL deductive system:

CORAL

As the design documentation of CORAL, it is necessary to introduce the main idea of the system. It involves:

- Basic concepts of CORAL, including relation, tuple, argument, variable binding, workspace, etc.
- Concepts of declarative language features, such as rules, modules, non_ground facts, negation, sets and multisets.
- Description of how CORAL works. Using an example to illustrate the execution of CORAL code and show the internal structure and values during the execution.
- Discussion of two major evaluation and optimization strategies and explanation of the different kinds of annotations.
- Using an example to illustrate how optimization effects the execution of CORAL code.

System architecture

The architecture concentrates on the design of a single user database system. It consists of four major components: User Interface, Query process system, Data management system, and Persistent data storage management system. The analysis of architecture includes the following works:

- Describe the functionalities for each component.
- List the interactions between components.

Major Data Structure

This part introduces the design of CORAL to represent some key concepts. It includes:

- Representation of argument and how CORAL uses the structure of argument to implement other data structures, such as functor, list, sets, etc.
- Using the example to show how CORAL represents a term.
- Representation of tuple, relation and TupleIterator.
- Description of the rule and module runtime data structures, showing the relationships between these data structures.

Class Diagrams

This part uses class diagrams to describe the structural model of the CORAL deductive database system. It includes:

- Each subsystems
- Subsystem collaboration.
- Some important classes in detail, such as the classes to implement optimization, the important subclasses of relation and the classes to implement the interface of an extensional database system.

Sequence Diagrams

This part chooses four scenarios and uses a sequence diagram for each scenario to illustrate the behavioral model of CORAL. The four scenarios are the main use cases of the CORAL deductive system, they are:

- User input s a fact at prompt.
- User inputs a Builtin command at prompt. It tells how CORAL implement Builtin commands.
- User inputs a module at prompt. It causes the compilation for the module.
- User inputs a query at prompt. It processes the evaluation of a query.

Through the study of CORAL deductive database system and the analysis of its source code, I learned what is a deductive database and what features CORAL has; understood different kinds of query optimization strategies, how they are implemented in CORAL; and how to use UML.

References

[1] R. Ramakrishnan, D. Srivastava, S. Sudarshan, P. Seshadri, "The CORAL Deductive System", *VLDB Journal*, vol 3, no. 2, pp. 161–210, 1994.

[2] R. Ramakrishnan, D.Srivastava, S. Sudarshan, P. Seshadri, "Implementation of the CORAL Deductive Database System", *SIGMOD Conference*, pp. 167-176, 1993

[3] Ivar Jacobson, Grady Booch, James Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

[4] R. Ramakrishnan, P. Seshadri, D. Srivastava, S. Sudarshan, *The CORAL User Manual, A Tutorial Introduction to CORAL*, Computer Sciences Department, University of Wisconsin-Madison, 1993.

[5] Raghu Ramakrishnan, *Database Management Systems* , Boston : McGraw-Hill, 2000.

[6] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.

[7] Tania Kharma, "Reengineering Unification and t-Entailment for Mantra in C++ ", *Master Thesis*, Dept. of Computer Science, Concordia University, 1996.

[8] Kenneth Ross, "Modular Stratification and Magic Sets for DATALOG programs with negation". Proceedings of the ACM Symposium on Principles of Database Systems, pp. 161--171, 1990.

[9] J. Vaghani, K. Ramamohanarao, David B. Kemp, Z. Somogyi, Peter J. Stuckey, J. Harland, "The Aditi Deductive Database System", VLDB journal, vol 3, no. 2, pp. 245-288, 1994.

[10] Marcia A. Derr, Shinichi Morishlta, and Geoffrey Phlpps, "The GLUE-NAIL Deductive Database System: Design, Implenmentation and Evaluation", VLDB journal, vol 3, pp. 123-160, 1994

[11] J. Dix and U. Furbach and A. Nerode, "Logic Programming and Nonmonotonic Reasoning", 1997. LNAI 1265, Springer-Verlag, Berlin, pp. 357-386

[12] G. Butler, L. Chen, X. Chen, A. Gaffar, J. Li, L. Xu, "The Know-It-All project: A Case Study in Framework Development and Evolution", *Domain Oriented Systems Development: Perspectives and Practices*, Taylor and Francis Publishers, UK, 2002.

Appendix

A Data Dictionary

BindEnv: is a binding environment, for a set of variables it indicates what they are each bound to.

Builtin: a special relation, to implement some user command.

Common: a subsystem that contains global classes used by most subsystems.

Compiler : a subsystem that is responsible to perform the optimization on a input module, And transform module into a intermidiate structure.

CORAL : A deductive database system.

Datalog : is a relational query language inspired by Prolog.

DataManager: a subsystem that is responsible for maintaining and manipulating the data in relations.

DDBMS: Deductive Database Management System.

EDB: extensional database that refers to a set of facts.

Evaluator: a subsystem that is responsible to perform the user's request.

EXODUS: a storage manager which has a client-server architecture.

IDB: intensional database that refers to the collection of rules.

Module : a subsystem that includes all the information for a module.

MySql : a relational database system.

PEI : a structure which mainly contains backtracking information and the offset of the corresponding predicate in the Predicates array.

Prolog: a programming language based on Horn clause logic.

Rewrit: a subsystem that is responsible to produce a transformed module in which the rules have been rewritten for optimization.

SemiNaive: an evaluation strategy.

SCC: Strongly Connected Component, is a region of this graph that is as small as possible such that there are no dependency cycles.

UML : Unified Modeling Language.

UserInterface: a subsystem used to provide communication with the user.