Coral++: Adding Object-Orientation to a Logic Database Language

Divesh Srivastava*
University of Wisconsin, Madison

Raghu Ramakrishnan
University of Wisconsin, Madison

Praveen Seshadri
University of Wisconsin, Madison

S. Sudarshan†
AT&T Bell Labs, Murray Hill

Abstract

Coral++ is a database programming language that integrates Coral [23] with the C++ type system. The data model allows arbitrary C++ objects in database facts, and the declarative query language extends Coral with C++ expressions in rules. Coral++ also supports an imperative rule-based sub-language that is integrated with C++, providing support for updates.

The design and implementation of Coral++ incorporates several important decisions: the data model is based on C++, and class definitions and method invocations are handled entirely by the C++ compiler; the notion of classes is kept orthogonal to the related notion of class extents; and declarative Coral++ programs can be largely understood in terms of standard Horn clause logic with C++ method invocations treated as external functions. The implementation outline illustrates that extending an existing deductive system to incorporate object-oriented features in the data model is feasible, and is orthogonal to the techniques used for object storage and retrieval.

1 Introduction

In recent years considerable research has been done in extending relational database languages, such as SQL, which have proven inadequate for a variety of emerging applications. Two main directions of research in database programming languages have been object-oriented database languages and deductive database languages, and the issue of combining the two paradigms has received attention recently.

Object-oriented database languages, such as O++ [1] and O_2 [7], among others, enhance the relational data model by providing support for abstract data types, encapsulation, object identifiers, methods, inheritance and polymorphism. Such sophisticated features are very useful for data modeling in many scientific, engineering, and multimedia applications. *Deductive database languages*, such as LDL [19], Coral [23] and Glue-Nail! [21], among others, enhance the declarative query language by providing a facility for generalized recursive view definition, which is of considerable practical importance. However, data models for deductive databases are typically structural, and do not have the richness of object-oriented data models.

One of the contributions of this paper is to demonstrate that the advantages of object-oriented database languages and deductive database languages can indeed be combined in a clean and practical manner. Our proposal, Coral++, has the following objectives:

*The work of Raghu Ramakrishnan, Divesh Srivastava and Praveen Seshadri was supported by a David and Lucile Packard Foundation Fellowship in Science and Engineering, a Presidential Young Investigator Award with matching grants from DEC, Tandem and Xerox, and NSF grant IRI-9011563. The addresses of the authors are Computer Sciences Department, University of Wisconsin, Madison, WI 53706, USA, and AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA. The authors e-mail addresses are {divesh,raghu,praveen}@cs.wisc.edu and sudarsha@research.att.com.

†Part of the work of this author was done while the author was at the University of Wisconsin.

Proceedings of the 19th VLDB Conference
Dublin, Ireland 1993

- To combine an object-oriented data model with a deductive query language.

  This permits the programmer to take advantage of the features of both object-oriented database languages and deductive database languages in developing applications. The Coral++ query language is significantly more expressive than the object-oriented extensions of SQL ([8, 6], for instance). A non-operational semantics is however maintained, and this makes Coral++ more amenable to automatic query optimization than imperative languages for object-oriented databases ([7, 1], for instance) that have similar expressive power.

- To cleanly integrate a declarative language with an imperative language.

  Such an integration is extremely useful since several operations such as updating the database in response to changes in the real world, input/output, etc., are imperative notions, whereas one can easily express many complex queries declaratively. A clean integration allows the programmer to do tasks in either the declarative style or the imperative style, whichever is appropriate to the task, and mix and match the two programming styles with minimal impedance mismatch.

- To keep object storage and retrieval orthogonal to the rest of the design, so that techniques developed for implementing object stores can be used freely in conjunction with other optimizations in the declarative query language.

1.1 Overview of the Coral++ Design

The central observation is as follows: Object-oriented features such as abstract data types, encapsulation, inheritance and object-identity are essentially extensions of the data model. We can achieve a clean integration of these features into a deductive query language by allowing the deductive language to draw values from a richer set of domains, and by allowing the use of the facilities of the deductive language to maintain, manipulate and query collections of objects of a given type.

In relational query languages such as SQL, values in fields of tables have been restricted to be atomic constants (e.g. integers or strings). In logic programs, values can be *Herbrand terms*, which are essentially structured values. In Coral++, values can additionally be of any class definable in C++ [29]. (We chose C++ since it provides a well-understood and widely used object-oriented type system.)

Coral++ provides support for maintaining *extents* or collections of objects of a given type, either in a simple manner that reflects the inclusions associated with traditional IS-A hierarchies, or in a more sophisticated way through the use of declarative rules. The idea is to automatically invoke code that handles extent maintenance whenever objects are created or destroyed, and provide constructs to use these extents in Coral++. We also provide support for creating and manipulating various types of collections: sets, multisets, lists and arrays, whereas traditionally only sets and multisets are provided.

Coral++ separates the querying of objects from the creation, updating and deletion of objects, and provides separate sub-languages for these two purposes. This methodology stems from the view that querying is possible in a declarative language, whereas creation, updates and deletion should be performed only using an imperative language.

In summary, the proposal is simple, combines features of C++ and Coral—two existing languages—with minimal changes to either, and yields a powerful combination of the object-oriented and deductive paradigms. The essential aspects of the integration are not specific to our choice of C++ and Coral, and the ideas can readily be used to combine other object-oriented and deductive systems in a similar manner.

1.2 Related Proposals

The Coral++ data model and query language have been influenced by other proposals for integrating object-oriented data models and declarative query languages. Some of the important aspects of our design are:

- Coral++ uses the type system of an existing object-oriented programming language (i.e., C++) as an object data model rather than inventing yet another object data model (e.g., [28, 8, 13, 9, 10, 17, 2]) This approach benefits from the support for data abstraction, inheritance, polymorphism and parametrized types already available in C++.

  Other query languages that use the C++ type system include CQL++ [6], ZQL[C++] [3] and ObjectStore [20].

- The Coral++ declarative query language supports the combination of Coral rules with C++ expressions in a clean fashion. This approach can effectively utilize the Coral implementation and the C++ compiler.

  Noodle [17] and ObjectStore [20], for instance, take the alternative approach of inventing new syntax to query an object data model.

- Coral++ is more expressive than most of the other proposals ([28, 8, 6, 9]). In particular, it provides a facility for generalized recursive view definition in the query language. It also supports unordered relations (i.e., sets and multisets) and ordered relations (lists and arrays), which are useful in applications involving sequence data [27].

- The Coral++ query language can be largely understood in terms of standard Horn clause logic, unlike Noodle [17] which is based on HiLog [5] and XSQL [13] which is based on F-logic [14]. Bottom-up evaluation of HiLog and F-logic programs is not as well understood as the evaluation of Horn clause programs and is likely to be more expensive.

- Our proposal includes a detailed implementation design that clearly demonstrates the practicality of extending Coral (an existing deductive system) with object-oriented features of C++ (a widely-used object-oriented type system). An implementation based on the run-time system of the Coral implementation [24] is already underway.

The rest of this paper is structured as follows. We start with example programs written in Coral++ that demonstrate several features of the data model and query language. We describe the object-oriented features of the Coral++ data model in Section 3. The Coral++ declarative query language is presented in Section 4 and the Coral++ imperative rule-based language is described in Section 5. We discuss in detail how Coral++ can be implemented using the existing Coral run-time system in Section 6. Finally, in Section 7, we compare our proposal with some of the related proposals in more detail.

2 Motivating Examples

Example 2.1 (A university database)
A university database maintains information about various departments as well as information about students and employees. Some Coral++ type declarations for this database are given below:[1]

```
class department {
public:
  employee   *head ;
  int   budget ;
} ;
class person {
public:
  char   *name ;
  int   age ( ) ;
} ;
class employee : public person {
public:
  int   salary ;
  department   *dept ;
  employee   *supervisor ;
  void   update_salary (int) ;
} ;
class student : public person {
public:
  department   *dept ;
} ;
```

Consider the query: "*Find all departments where the sum of the salaries of the employees has exceeded the department budget?*" If the collection of all employee objects is maintained as a relation called employee, the corresponding Coral++ program is:

```
budget_exceeded (D) : −  sum_of_salaries (D, S),
            S > D→budget.
sum_of_salaries (D, sum (< S >)) : −  employee (E),
            D = E→dept, S = E→salary.
```

Coral++ programs use a rule-based syntax, similar to Coral and logic programming languages. A difference is that Coral++ allows the use of C++ expressions (for instance, E→dept) in rules to access attributes and invoke methods. Each rule can be read as an "if-then" statement in logic. For instance, the meaning of the first rule is "if the sum of salaries in a given department D is S and S is greater than the budget of department D, then the budget of department D has been exceeded"; the second rule gives a way of computing the sum of the salaries of the employees in a given department.

[1] Coral++ type declarations have the same syntax as C++ class declarations.

Consider the query: "*Find all students in departments where the head of the department is named John.*" The corresponding Coral++ program is:

```
jd_students (D, < S >) : −  j_dept (D), student (S),
          S→dept = D.
j_dept (D) : −  department (D), D→head→name = "John".
```

The second rule in the above program is used to find out all departments where the head of the department is named "John", and the first rule collects all the students in such departments.

These two queries can also be expressed in object-oriented extensions of SQL, and are provided here to primarily illustrate the difference in program specification styles. □

Example 2.2 (An engineering application) An engineering database for a manufacturing company stores information about the various parts manufactured, along with information about composition of parts. Some Coral++ type declarations for this database are given below.

```
class part {
private:
   int    functionality_test ;
   int    connection_test ;
public:
   char    *ptype ;
   int    tested ( );
} ;
class connection {
public:
   part    *from ;
   part    *to ;
   char    *ctype ;
};
```

The following Coral++ program can be used to express the bill-of-materials problem: "*Find all subparts of a given part*". This problem is of considerable practical importance in inventory control, and other applications.

```
subpart (P1, P2) : −  connection (C), P1 = C→from,
          P2 = C→to, C→ctype = "subpart".
subpart (P1, P3) : −  connection (C), P1 = C→from,
          P2 = C→to, C→ctype = "subpart",
          subpart (P2, P3).
all_subparts (P1, < P2 >) : −  subpart (P1, P2).
```

Consider the following query from [26]: "*Find if a given part is working, where a part is known to be working either if it has been (successfully) tested or if it is constructed from smaller parts, and all the smaller parts are known to be working*". This can be expressed in Coral++ as follows:

```
working (P) : −  part (P), P→tested ( ) = 1.
working (P) : −  connection (C), C→from = P,
          C→ctype = "subpart", not has_suspect_part (P).
has_suspect_part (P) : −  connection (C), C→from = P,
          C→to = P1, C→ctype = "subpart",
          not working (P1).
```

Neither of these two queries can be expressed in SQL or its object-oriented extensions because of the recursive definitions of subpart and working. □

3 Coral++: Data Model

In database languages such as SQL, values in fields of relational tables are unstructured, i.e., restricted to be of a basic type supported by the system (e.g. integers or strings). In deductive database languages such as LDL [19] and Coral [23], values can be *Herbrand terms*, which are essentially structured values. However, data modeling in many scientific and engineering applications require support for more sophisticated features such as abstract data types, encapsulation, methods and inheritance. To support the data modeling needs of such applications, the Coral++ data model enhances the untyped Coral data model [23] with the C++ class facility. Values in Coral++ can additionally be of any type definable in C++, which can be manipulated using only the corresponding methods, supporting encapsulation. This allows a programmer to effectively use a combination of C++ and Coral, with minimal impedance mismatch.

One of the goals of Coral++ was to integrate Coral with an *existing* object data model, instead of inventing yet another object data model. By using the C++ type system as an object model, our approach is able to benefit from the support of data abstraction, inheritance, parameterized types, and polymorphism already available in C++. The choice of C++ was based on practical implementation considerations (Coral is implemented in C++), but we believe that our approach can also be applied to extending Coral with an alternative object-oriented data model.

3.1 Overview of the Coral Data Model

The formal definitions of constants, variables, terms, tuples, facts and relations are available in logic programming texts such as [15]. We informally describe these features of the Coral data model. The following facts could be interpreted as follows: the first fact indicates that John is an employee in the "Toys for Tots" department who has been with the company for 3 years and makes 35K. The second fact indicates that Joan has worked for the same department for 2 years and makes 30K.

```
works_for (john, "Toys for Tots", 3, 35)
works_for (joan, "Toys for Tots", 2, 30)
```

In order to express structured data, complex terms are required. In Coral, function symbols are used as record constructors, and such terms can be arbitrarily nested. The following fact can be interpreted as: John lives in Madison, and has a street address with a zip of 53606.

```
address (john, residence ("Madison",
        street_add("Oak Lane", 3202), 53606))
```

Sets and multisets are allowed as values in Coral; {peter, mary} is an example of a set representing the children of John, {60, 35, 35, 30} is an example of a multiset representing the salaries of employees in the "Toys for Tots" department.

Coral permits variables within facts. A fact with a variable in it represents a possibly infinite set of ground facts. Such facts are often useful in knowledge representation, natural language processing and could be particularly useful in a database that stores (and possibly manipulates) rules. There is another, possibly more important, use of variables — namely to specify constraint facts [12, 22].

However, the Coral data model does not allow values of (arbitrary) user-defined types in facts. These are extremely useful in several applications, especially when the user-defined types have behavioral components.

3.2 Overview of the C++ Type System

C++ allows the specification of user-defined types using the class definition facility. An implementation of a C++ class is a combination of the attributes that specify the "structure" of the class along with the implementation of the methods that specify the "behavior" of the class. Attributes and methods of a class may be specified as either "public," "private," or "protected," providing different levels of encapsulation. Classes can be organized in an inheritance hierarchy in C++, and a class can have more than one subtype as well as more than one supertype (i.e., C++ supports multiple inheritance). The type declarations of Examples 2.1 and 2.2 illustrate some of these features.

By integrating the C++ object model with Coral, the Coral++ user benefits from having sophisticated data modeling and manipulation capabilities.

3.3 Relations in Coral++

Coral supports relations that are multisets (i.e., unordered collections) of tuples. Typically, current database systems support only multisets of tuples, and the utility of these collections can be seen from the variety of applications written in SQL. However, for many applications (see [25, 27], for instance) involving sequence data and spatial data, for example, ordered collections of list-type and array-type are more natural. Hence, Coral++ also supports list-relations and array-relations, in addition to multiset-relations.

Each of these relation types supports the operation of iterating through the elements in the collection. The difference is the *primary* mode of access to elements in the collection. Multisets support unordered access, lists support ordered access in the total order of the list elements, and arrays support access in the array index order. In addition, each collection type can have value-based indexed modes of access, where the index can be on specified attributes or patterns.

Example 3.1 (Stock market data)
A stock market database maintains daily information about the stocks traded for each company. A small fragment of such type information is shown below:

```
class DailyStockInfo {
public:
  double   low ;
  double   high ;
  double   average ( ) ;
  int   volume_traded ;
} ;
```

Stock market information for individual companies can be naturally represented as array relations, which results in extremely efficient querying and manipulation of such information, as is demonstrated in the Mimsy system [27]. □

4 Coral++: Query Language

The Coral++ query language is modular, declarative and provides support for generalized recursive view definition. It is based on the Coral query language [23] which supports general Horn clauses with complex terms, set-grouping, aggregation and negation. Coral++ extends the Coral query language by allowing C++ expressions for accessing attributes and invoking (side-effect free) methods of objects in program rules.

Coral++ incorporates several important design decisions in the way the data model interacts with the declarative query language:

- The notion of a class as an encapsulation of data and methods is kept *orthogonal* to the related notion of class extents. This is achieved by providing the Coral++ programmer considerable flexibility in explicitly defining and maintaining collections of objects of a given class. We describe this in more detail in Section 4.3.

- The C++ expression truth semantics is kept distinct from the Coral++ predicate truth semantics. This is achieved by allowing C++ expressions to appear *only* in the argument positions of predicates (including evaluable predicates such as $=, <=$, etc.). This is discussed in more detail in Section 4.4.

- Declarative rules in Coral++ do not create new objects (instances of C++ classes), although they can create facts describing relationships between existing objects. The rationale for this decision is discussed in Section 4.5.

One of the major advantages of our proposal is that evaluation of Coral++ programs is based on the existing Coral run-time system, which facilitates implementation considerably. More generally, it suggests that optimization techniques developed for deductive and for object-oriented database languages can be combined cleanly. In this section, we first give an overview of the Coral query language and the evaluation of Coral queries, and then discuss the Coral++ design decisions.

4.1 Overview of the Coral Query Language

The formal definitions of constants, variables, terms, atoms, literals, facts and rules are available in logic programming texts such as [15]. We briefly describe some of these features here. Rules in Coral take the form:

$$\mathsf{head} :- \ \mathsf{body}_1, \mathsf{body}_2, \ldots, \mathsf{body}_n.$$

where head is a positive literal, each body_i is a (positive or negative) literal, and $n \geq 0$. Informally, each rule can be read as a statement of the form "if <body is true> then <head is also true>". (In particular, a fact is just a rule with an empty body.) In the absence of rules with negation, set-generation and aggregation, the meaning of a program can be understood by reading each of the rules in the program in this manner, with the further understanding that the only true facts are those that are either part of the input database or that follow from a repeated use of program rules.

Coral supports and efficiently evaluates a class of programs with negation that properly contains the class of non-floundering left-to-right modularly stratified programs ([26]). Intuitively, this class is one in which the subgoals and answers generated during program evaluation involve no cycles through negation. The keyword '*not*' is used as a prefix to indicate a negative body literal. (A query in Example 2.2 illustrates the use of negation.)

There are two ways in which sets and multisets can be created using Coral rules, namely, multiset-enumeration ({ }) and multiset-grouping (<>). The following illustrate the use of these constructs:

```
children (john, {mary, peter, paul}).
p (X, <Y>) :- q (X, Y, Z).
```

This second rule uses facts for q to generate a multiset S of instantiations for the variables X, Y, and Z. For each value x for X in this set it creates a fact p $(x, \pi_Y \sigma_{X=x} S)$, where π_Y is a multiset projection (i.e., it does not do duplicate elimination). Thus with facts q (1,2,3), q (1,2,5) and q (1,3,4) we get the fact p (1, {2,2,3}). The use of the multiset-grouping construct in Coral is similar to the grouping construct in LDL, except that it constructs a set (as opposed to a multiset) in LDL.

Coral requires that the use of the multiset-grouping operator be left-to-right modularly-stratified (in the same way as negation). This ensures that all derivable q facts with a given value x for X can be computed before a fact p (x,_) is created.

Coral provides several standard operations on sets and multisets as system-defined predicates. These include member, union, intersection, difference, multisetunion, cardinality, subset, and makeset. Coral also allows several aggregate operations on sets and multisets: these include count, min, max, sum, product, average and any. Some of the aggregate operations can be combined directly with the multiset-generation operations for increased efficiency (see [23] for further details).

Modules provide a way, as the name suggests, to modularize Coral code. In developing large applications, incremental program development and testing is critical, and modules in Coral provide the basis for this kind of programming. A module in Coral consists of a collection of rules defining a collection of predicates. A subset of these defined predicates are named as exported predicates, and other modules can pose queries over these predicates. The query forms permitted for each exported predicate are also indicated in the export declarations. Non-exported predicates are not visible outside this module and this provides a way of encapsulating the definition of Coral predicates.

4.2 Evaluating Coral Queries

The evaluation of a Coral module, given a query on an exported predicate of a module, is determined by the control annotations in the module, and the expert user can control the evaluation in several ways. We refer the interested reader to [23] for a discussion of these annotations and their effect on module evaluation. In the absence of any user-specified annotations, the Coral system chooses from among a set of default evaluation strategies.

For declarative modules, Coral evaluation, using these default strategies, is guaranteed to be sound, i.e., if the system returns a fact as an answer to a query, that fact indeed follows from the semantics of the declarative program. The evaluation is also “complete” in a limited sense — as long as the execution terminates, all answers to a query are actually generated. It is possible however, to write programs that do not terminate; in some such cases (e.g., programs without negation, set-grouping or aggregation) Coral is still complete in that it enumerates all answers in the limit.

During the evaluation of a rule r in module A, if we generate a query on a predicate exported by module B, a call is set up on module B. The answers to this query are used iteratively in rule r; each time a new answer to the query is required, rule r requests a new answer from the interface to module B. The module interface makes *no assumptions* about the evaluation of the module. Module B may contain only database facts, or may have Coral rules that are evaluated in any of several different ways. The module may choose to cache answers between calls, or choose to recompute answers. All this is transparent to the calling module. Similarly, the evaluation of the called module B makes no assumptions about the evaluation of calling module A. This orthogonality permits the free mixing of different evaluation techniques in different modules in Coral and is central to how different executions in different modules are combined cleanly.

4.3 Class Extents in Coral++

Coral++ keeps the notion of a class (as an encapsulation of data and methods) orthogonal to the related notion of class extents (i.e., the collection of all objects of the given class). Although maintaining class extents is necessary for iterating over all objects of a given class (as in Example 2.2), Coral++ does not automatically maintain class extents since doing so is very expensive, and one does not always need to iterate over *all* objects of a given class.

Coral++ provides the programmer considerable flexibility in explicitly defining and maintaining collections of objects of a given class. Collections of objects can be maintained either in a simple manner that reflects the inclusions associated with traditional IS-A hierarchies, or in a more sophisticated way through the use of declarative rules. Coral++ provides functions that can be explicitly invoked from class constructors and destructors, and that handle extent maintenance. Such class extents are maintained as Coral++ relations, and can be used as literals in the bodies of Coral++ rules. For example, the following literal can be used to iterate over the extent of class part, in the body of a Coral++ rule:

```
part (P)
```

The variable P is successively bound to (pointers to) objects in the extent. (See Example 2.2 for further uses of class extents.)

4.4 C++ Expressions in Rules

By integrating the C++ object model with the Coral data model, Coral++ allows facts to contain objects of C++ classes. Such objects can be manipulated using only the corresponding methods, supporting encapsulation. This is achieved by allowing C++ expressions in Coral++ rules to access attributes and invoke methods of objects. For example, one can iterate over all tested parts using:

```
part (P), P→tested ( ) = 1.
```

Such C++ expressions can appear *only* in the argument positions of predicates (including evaluable predicates such as $=, <=$, etc.). This ensures that the Coral++ predicate truth semantics is kept distinct from the C++ expression truth semantics. (The C++ type system does not include a boolean type; any arithmetic expression is considered false if its value is zero, and true otherwise.) For instance, the following are not legal Coral++, although the C++ conditional expressions can each be used in the C++ if-statement:

```
part (P), P→tested ( ).
department (D), D→head→name.
```

4.5 Creating Objects in Coral++

Objects can be created using constructor methods (specified along with the class definition), and deleted using destructor methods (also specified along with the class definition). Coral++ requires that objects that are instances of C++ classes be explicitly created *only* using C++; the database can be populated with such objects only from C++. However, the Coral++ declarative language can create facts describing relationships between existing objects in the database.

Rules in Coral++ are deliberately restricted to avoid creating new objects, since this is an issue that is *not* yet well-understood despite work by Maier [16], Kifer et al. [13], and others. A number of issues, notably the resolution of conflicts when rules generate distinct objects with the same object identifier, remain unclear, especially in the presence of partially specified objects (e.g. some fields are variables, in the Coral++ context).

Similarly, updating and deleting objects that are instances of C++ classes should be performed only using the imperative language. This methodology stems from the view that the query language has to be *declarative*, whereas creating, updating and deleting objects are *operational* notions.

5 Coral++: Imperative Language

The Coral++ declarative language cannot be used to create objects that are instances of C++ classes, delete such objects, or update such objects. We view these as *operational* notions and hence provide an imperative language for this purpose. This imperative language consists of C++ augmented with new types and constructs to effectively deal with collections. For instance, it provides the Coral++ user with the ability to iterate through collections. This can be extremely useful in performing database updates, for instance, where the order in which the updates are performed in a collection may be critical. The imperative language also supports imperative Coral++ rules, which can be of the following forms, as in Coral:

$$\begin{array}{l} \mathsf{head}. \\ \mathsf{head} := \ \mathsf{body}_1, \mathsf{body}_2, \ldots, \mathsf{body}_n. \\ \mathsf{head} += \ \mathsf{body}_1, \mathsf{body}_2, \ldots, \mathsf{body}_n. \\ \mathsf{head} -= \ \mathsf{body}_1, \mathsf{body}_2, \ldots, \mathsf{body}_n. \end{array}$$

where each body_i is a literal and head is an atom. An imperative Coral++ rule can also be of the form:

$$\mathsf{head} *= \ \mathsf{body}_1, \mathsf{body}_2, \ldots, \mathsf{body}_n.$$

where each body_i is a literal and head is a C++ expression. This imperative Coral++ rule corresponds to the invocation of arbitrary methods on objects; the arguments to such a method could depend on the body of the rule. The C++ control structures are used to provide sophisticated control on the order in which imperative Coral++ rules are applied. Here we give an example, and present details in the full version of the paper.

Example 5.1 (Updates)
The following rule increments the salaries (using the method update_salary of class employee) of all employee objects named "divesh" by 10%.

```
E→update_salary (NewVal) *= employee (E),
        E→name = "divesh", E→salary = OldVal,
        NewVal = 1.1 * OldVal.
```

The body of the rule is treated as a query and evaluated to bind the variables E and NewVal. For each E and NewVal pair, the method update_salary (...) is invoked. (For this operation to make sense, the query in the body must have at most one binding for NewVal for each value of the variable E. However, ensuring this is left to the user, and Coral++ does not check this.)

With updates, often the order in which updates are performed can affect the final outcome. Consider the following rule which updates the salary of each employee to the salary of the employee's supervisor.

```
E→update_salary (NewVal) *= employee (E),
        E→supervisor→salary = NewVal.
```

Clearly, the order in which updates are performed would affect the final salaries of the employees. All such operations in Coral++ have *deferred semantics*, described below.

- First, the body of the rule is evaluated as a query to obtain *all* answers to the query, in particular bindings for all variables in the head of the imperative rule.
- Next, the C++ expression specified in the head of the imperative rule is evaluated for each query answer.

One can also update a collection by adding elements to them using Coral++ imperative rules.

```
multi_level (P1, P2) := connection (C), C→from = P1,
        C→to = P2, C→ctype = "subpart".
multi_level (P1, P2) += connection (C), C→from = P1,
        C→to = P3, C→ctype = "subpart",
        multi_level (P3, P2).
```

Note that this is a sequence of two imperative rule applications. It adds pairs of parts P1 and P2 to the multi_level collection such that either P2 is a subpart of P1, or there is a part P3, which is a subpart of P1 and of which P2 is a subpart. Note that it *does not* compute the entire subpart collection, as is done by the declarative rules in Example 2.2. □

Example 5.2 (Deletions)
The Coral++ user can delete elements from collections as follows.

```
multi_level (P1, P2) −= connection (C), C→from = P1,
        C→to = P2, C→ctype = "subpart".
```

If this rule is applied after the two rules in Example 5.1, the effect is to remove all pairs of parts P1 and P2 from the multi_level collection such that P2 is a subpart of P1. However, this *does not* delete the parts referenced by P1 and P2 themselves; it only deletes the record describing this relationship from the collection multi_level.

Deleting objects in a collection can also be affected by the order in which the delete operation is performed. As with updates, we have a *deferred semantics* for deletes. □

6 Implementing Coral++

One of the fundamental design decisions of our proposal is to use the run-time system of the Coral implementation [24] as much as possible in the implementation of Coral++. Several design decisions are a practical consequence of this:

- The notation for class definitions in Coral++ is the same as in C++. This allows the Coral++ class definitions to be handled by the C++ compiler directly.
- All variables in Coral++ rules have to be coerced to the appropriate type before invoking a method or accessing an attribute. This permits Coral++ to avoid inferencing types at compile-time.

As a consequence of these design decisions, the evaluation of a Coral++ program augmented with class definitions proceeds as follows. First, the class definitions and the method definitions (if any) provided by the user are compiled using the C++ compiler. These are compiled along with the basic Coral++ system to create an enhanced Coral++ system that "knows" about these new classes. (See Section 6.1.) Second, Coral++ program modules go through a translation phase for handling attribute accesses and method invocations. (See Section 6.2.) Finally, the translated programs are directly evaluated using the Coral++ interpreter. Figure 1 depicts the Coral++ program compilation process pictorially.

6.1 Implementing Classes and Extents

We briefly describe the Coral run-time system with a view to describing the implementation of Coral++. The Coral system is implemented using C++ and all Coral data types are represented as C++ classes. The root of all data types is the virtual class CoralArg; specific types such as complex terms and multisets are all

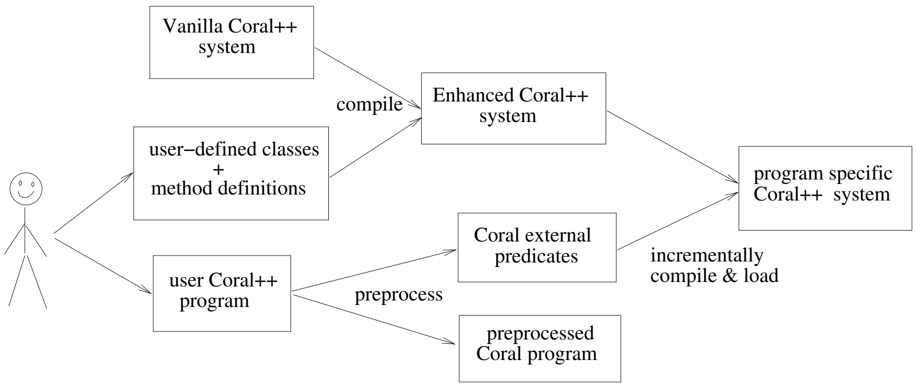

Figure 1: Coral++ Program Compilation

subclasses of the class CoralArg. The class CoralArg defines a set of virtual methods that must be defined for each Coral data type; this includes methods such as the method equals, which is used to compare whether two instances of CoralArg are the same, and the method print, which is used to display the value to the Coral user.

Our approach for a practical implementation of Coral++ classes is summarized as follows:

- User-defined classes in Coral++ have the same syntax as C++ classes. All class definitions including method definitions are completely handled using the C++ compiler. Subtyping (including multiple inheritance) in Coral++ is automatically implemented using the inheritance mechanism of C++.

- All user-defined Coral++ classes should be subclasses of the root class CoralArg. Because all values used in Coral++ rules at run-time are of a type derived from CoralArg, Coral++ does not have to perform any dynamic type inferencing and type conversion to determine the methods that need to be invoked during rule evaluation.

- We provide C++ macros that can be used to maintain class extents. The user has to explicitly insert these macros into the definitions of the constructor and destructor methods of each class whose extent has to be maintained.

6.2 Program Evaluation in Coral++

Program evaluation in Coral++ requires modifying the existing program evaluation strategy in Coral to access named attributes and invoke methods of objects, instead of simply accessing relation field values using position notation. Given a Coral++ program, these requirements can be satisfied as follows:

- First, for each attribute access and method invocation in a Coral++ rule, the Coral++ preprocessor generates external (C++) predicates that perform the appropriate attribute access or method invocation at run-time. This code can be separately compiled and incrementally loaded.

  This approach relegates the task of binding the method name with the actual code to invoke the method to the C++ compiler. The alternative approach of invoking the C++ methods directly from the Coral++ interpreter would involve duplicating some of the tasks of the C++ compiler including maintaining symbol tables and virtual function tables, which would be quite impractical.

- Second, the program is translated to replace all occurrences of method invocations and attribute accesses by the appropriate external predicates.

  Appropriate indexes are also created at this time for providing associative access to relations containing objects.

- Finally, the translated program is evaluated using the Coral interpreter for evaluating rules, modules and programs.

  The evaluation of Coral++ modules can use the query-directed rewriting optimizations as well as the various optimizations of the existing Coral run-time system.

The decision to relegate the task of determining which code is to be evaluated at method invocation to the C++ compiler results in the following practical design decision for methods invoked in Coral++ rules: All uses of rule variables must be coerced to the appropriate type before accessing an attribute or invoking a method. This must be done to avoid sophisticated type inferencing by Coral++, which would involve considerable implementation effort.

7 Related Work

There are many proposals in the literature ([28, 4, 6, 8, 13, 9, 20, 10, 17, 3, 2, 11, 30], among others) for integrating object-oriented data models and declarative query languages. Typically, these proposals support features such as complex objects, data abstraction, inheritance and polymorphism in their data model, and the ability to pose queries on collections of objects using a suitable query language. We presented a summary of the differences between our proposal and these other proposals in Section 1.2. We now examine some of the closely related proposals in more detail.

Proposals Based on C++

ZQL[C++] [3] and CQL++ [6] are the proposals most closely related to Coral++ since they are also based on the C++ object model.

The Coral++ query language is more expressive than CQL++ or ZQL[C++], which are based on SQL. However, each of these proposals is integrated with a computationally complete imperative language: CQL++ with O++ [1], and Coral++ and ZQL[C++] with C++.

CQL++ has a syntax similar to SQL syntax for class definition. These classes do not have any facility for data abstraction (i.e., all class members are *public*). Further, accessing an attribute or invoking a method in a CQL++ query uses the 'dot notation' of SQL, i.e., the user does not have to deal with explicit pointer dereferencing. In Coral++ and ZQL[C++], class definitions can use all the features of C++ including data abstraction, and C++ expressions can be used for accessing attributes and invoking methods in a query.

In Coral++ and CQL++, path expressions are treated as values that can be arguments to boolean-valued predicates. ZQL[C++], on the other hand, allows C++ expressions to serve directly as predicates. Since ZQL[C++] also allows SQL subqueries to appear as predicates, it does not distinguish between the predicate truth semantics and the C++ expression truth semantics, unlike Coral++ and CQL++.

Proposals Based on Deductive Languages

The COMPLEX data model [9] is a structural, typed data model that adds features such as object identity, object sharing and inheritance to the relational model. It does not support abstract data types, encapsulation, or methods; consequently, the data model is not as rich as the Coral++ data model. The query language of COMPLEX is C-Datalog which can be automatically translated to Datalog, and evaluated using an engine for evaluating Datalog programs. This translation is possible because of the lack of behavioral features and polymorphism in the data model. It is not clear how the translation approach generalizes once we introduce behavioral features in the model.

LDL++ [2] is a deductive database system whose type system extends that of LDL [19] with an abstract data type facility that supports inheritance and predicate-valued methods. However, it does not support object sharing or ADT extents, and its support of encapsulation and object identity is limited. consequently, the data model is not as rich as the Coral++ data model. Further, LDL++ methods can be defined only using LDL++ rules; however, this can be done more naturally than in Coral++.

Proposals Based on Non-Horn Logics

XSQL [13] extends SQL by adding path expressions that may have variables that range over classes, attributes and methods. This facilitates querying schema information as well as instance-level information in object-oriented databases, using a single declarative query language. Noodle [17, 18] is a declarative query language for the Sword declarative object-oriented database. Unlike Coral++ and XSQL, Noodle does not use path expressions to access attributes and invoke methods on objects. Instead, Noodle uses a syntax reminiscent of HiLog [5] for this purpose. Noodle also has a number of built-in classes to facilitate schema querying. Orlog [10] combines the modeling capabilities of object-oriented and semantic data models, and is similar to Noodle in that its logic-based language for querying and implementing methods uses a higher-order syntax with first order semantics.

In Coral++, methods and other aspects of data abstraction borrowed from C++ are viewed as being outside the scope of the deductive machinery, notably the unification mechanism. A more comprehensive treatment of features like path expressions (e.g., as in XSQL [13]) may well enable more efficient (i.e., set-oriented) processing of certain queries. We make no attempt to give these features a logical semantics; we simply borrow the C++ semantics, in order to enable ease of implementation.

The semantic foundations of XSQL, i.e., F-logic [14], Noodle, i.e., HiLog, and Orlog have features that are difficult to support efficiently, at least in a bottom-up implementation. In particular, variables can get bound to predicate names only at run-time, and this causes problems with analysis of strongly connected components (SCCs) and can make semi-naive evaluation inefficient. In contrast, one of the design motivations of Coral++ was to have a language that is rich in expressive power *and* can be efficiently evaluated within the framework of existing evaluation techniques.

There are several other interesting proposals for combining semantically rich data models with deductive databases that are less closely related to Coral++. ConceptBase [11] and Quixote [30] are two such systems. ConceptBase is based on the Telos knowledge representation language, and allows the specification of methods using deductive rules and integrity constraints. Quixote is a knowledge representation language that allows subsumption constraints, knowledge classification and inheritance and query processing for partial information databases.

8 Conclusions

We described Coral++, an object-oriented extension of Coral. The Coral++ data model extends the structural data model of Coral by integrating it with the C++ type system. The Coral++ query language extends Coral by allowing C++ expressions for accessing attributes and invoking methods of objects. The Coral++ query language is much more expressive than object-oriented extensions proposed for SQL, while remaining declarative at the same time. Consequently, a variety of rewriting and evaluation-time optimizations can be performed to improve efficiency; in particular, the optimizations performed for Coral programs are applicable to Coral++ programs as well. The Coral++ imperative rule-based language can be used to create, update and remove objects from the database. It is cleanly integrated with C++, providing the user the ability to program in a combination of programming styles, with minimal impedance mismatch.

We proposed an implementation strategy for Coral++ that effectively uses the existing Coral run-time system [24] and the C++ compiler to implement object-oriented features of the data model and query language. This, in our view, is one of the strong points of our proposal, and distinguishes it from many proposals in the literature describing query languages for object-oriented databases. The implementation strategy is orthogonal to issues such as object clustering, caching, indexing, storage management, etc. Although we described the implementation using the C++ class hierarchy, Coral++ does not depend on C++ implementation techniques for classes and class instances; it could also be implemented on top of a typed, persistent object store. We believe that Coral++ is a realistic and useful proposal for engineering, scientific and multi-media applications that can benefit from object-oriented data models and high-level data access and manipulation capabilities.

Acknowledgements

We would like to thank Naser Barghouti for his comments on an earlier draft of the paper, and Shaul Dar for his extensive comments and criticisms, which helped improve the content and presentation of this paper considerably.

References

[1] R. Agrawal and N. H. Gehani. Ode (Object Database and Environment): The language and the data model. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Portland, Oregon, June 1989.

[2] N. Arni, K. Ong, S. Tsur, and C. Zaniolo. The LDL++ system: Rationale, technology and applications. (Submitted), 1993.

[3] J. A. Blakeley. ZQL[C++]: Integrating the C++ language and an object query capability. In *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 138–144, Washington, D.C., May 1993.

[4] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 225–236, Atlantic City, New Jersey, May 1990.

[5] W. Chen, M. Kifer, and D. S. Warren. Hilog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North Amer-*

ican Conference on Logic Programming, pages 1090–1114, 1989.

[6] S. Dar, N. H. Gehani, and H. V. Jagadish. CQL++: An SQL for a C++ based object-oriented DBMS. In *Proceedings of the International Conference on Extending Database Technology*, Vienna, Austria, Mar. 1992. (A full version is available as AT&T Bell Labs Technical Memorandum 11252-910219-26).

[7] O. Deux. The O_2 database programming language. *Communications of the ACM*, Sept. 1991.

[8] L. J. Gallagher. Object SQL: Language extensions for object data management. In *Proceedings of the ISMM First International Conference on Information and Knowledge Management*, pages 17–26, Baltimore, Maryland, Nov. 1992.

[9] S. Greco, N. Leone, and P. Rullo. COMPLEX: An object-oriented logic programming system. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):344–359, Aug. 1992.

[10] M. H. Jamil and L. V. S. Lakshmanan. ORLOG: A logic for semantic object-oriented models. In *Proceedings of the ISMM First International Conference on Information and Knowledge Management*, pages 584–592, Baltimore, Maryland, Nov. 1992.

[11] M. Jarke, S. Eherer, R. Gallersdoerfer, M. Jeusfeld, and M. Staudt. ConceptBase – a deductive object base manager. (Submitted), 1993.

[12] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 299–313, Nashville, Tennessee, Apr. 1990.

[13] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 393–402, San Diego, California, 1992.

[14] M. Kifer and G. Lausen. F-logic, a higher-order language for reasoning about objects, inheritance and schemes. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1989.

[15] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

[16] D. Maier. A logic for objects. Technical Report Technical report CS/E-86-012, Oregon Graduate Center, Beaverton Oregon 97006-1999, November 1986.

[17] I. S. Mumick and K. A. Ross. An architecture for declarative object-oriented databases. In *Proceedings of the JICSLP-92 Workshop on Deductive Databases*, pages 21–30, Washington, D.C., Nov. 1992.

[18] I. S. Mumick and K. A. Ross. The influence of class hierarchy choice on query language design. In *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 152–154, Washington, D.C., May 1993.

[19] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*. Principles of Computer Science. Computer Science Press, New York, 1989.

[20] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in the ObjectStore database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 403–412, San Diego, California, 1992.

[21] G. Phipps, M. A. Derr, and K. A. Ross. Glue-NAIL!: A deductive database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 308–317, 1991.

[22] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programs. In *Proceedings of the International Conference on Logic Programming*, pages 140–159, Seattle, Washington, August 1988.

[23] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[24] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993.

[25] J. Richardson. Supporting lists in a data model (a timely approach). In *Proceedings of the International Conference on Very Large Databases*, pages 127–138, Vancouver, Canada, 1992.

[26] K. Ross. Modular Stratification and Magic Sets for DATALOG programs with negation. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–171, 1990.

[27] W. G. Roth. Mimsy: A system for analyzing time series data in the stock market domain. Technical Report (To appear), University of Wisconsin at Madison, 1993.

[28] L. A. Rowe and M. R. Stonebraker. The POSTGRES data model. In *Proceedings of the Thirteenth International Conference on Very Large Databases*, pages 83–96, Brighton, England, Sept. 1987.

[29] B. Stroustrup. *The C++ Programming Language (2nd Edition)*. Addison-Wesley, Reading, Massachusetts, 1991.

[30] K. Yokota, H. Tsuda, and Y. Morita. Specific features of a deductive object-oriented database language QUIXOTE. In *Proceedings of the Workshop on Combining Declarative and Object-Oriented Databases*, pages 89–99, Washington, D.C., May 1993.