

Integration of Persistent and Relational Database Objects in CORAL

Barrett R. Bryant*

Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, Alabama 35294, U. S. A.
InterNet: bryant@cis.uab.edu

Daniel T. Chang and Taejae Lee
Data Base Technology Institute
IBM Santa Teresa Laboratory
San Jose, California 95161, U. S. A.
InterNet: {dtchang, tjlee}@stlvm14.vnet.ibm.com

Abstract

CORAL is an object-oriented language for parallel and distributed application programming. Fully integrated into the language are persistent classes which allow permanent storage of complex objects and database classes which provide a transparent interface with a relational database management system. The combination of object-orientation, parallelism and distribution, and persistence distinguishes CORAL among database programming languages. Furthermore, the coupling of an object-oriented programming system with relational databases provides unique flexibility and expressivity.

1 Introduction

During the past three decades, file systems and database systems have been used by applications which require access to some form of persistent data. Among those in use today, re-

lational databases have emerged as the most prevalent because of their simplicity of use and automation of design, facilitated by the declarative nature of relational query languages. Unfortunately, relational query languages have significant limitations in their capabilities as a result of their simplicity and it is often difficult to manipulate complex data structures consisting of many integrated components, as found in modern data-intensive applications such as software engineering environments, VLSI circuit design, graphical data manipulations, or other scientific applications. These limitations are often overcome by developing specialized software using an interface to the relational database from a general-purpose programming language, a task which requires in-depth knowledge and expertise since, among other things, the data structures provided by programming languages and database systems do not match well.

A major goal of object-oriented programming is to increase software productivity and quality. Unlike traditional procedure-oriented programming, object-oriented programming provides many high-level abstractions to ease the programming effort. The development of applications which require access to persistent

*This research was performed while the author was a Visiting Scientist at the Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, California.

data in object-oriented programming should be as productive and result in the same quality as the development of any other applications. To accomplish this goal, persistent objects should be supported in object-oriented programming systems in a transparent manner such that they can be defined and manipulated like transient objects with little additional effort. For relational database objects, because of the limitations of existing relational database management systems and because of other considerations, total transparency is not feasible and may not be desirable. However, the impedance mismatch problem should be kept to a minimum.

This paper describes CORAL, a Concurrent Object-oriented Application Language developed at the IBM Palo Alto Scientific Center, which permits the integration of persistent objects and relational database objects into a unified framework. These objects may be defined and manipulated by the user in a convenient manner transparent of their actual storage representations. Furthermore, CORAL is designed for parallel and distributed applications allowing manipulation of concurrently accessed data or distribution of data over a network of heterogeneous computer systems. In the next section, we will survey related work in the area of persistent data management. Section 3 describes CORAL in more detail and distinguishes it from other database and object-oriented programming languages. Persistent objects and relational database objects are discussed in Sections 4 and 5, respectively. Finally, we conclude with Section 6.

2 Related Work

Applications involving databases are often limited to what the database query language being used has to offer. To solve this problem, the field of database programming languages has arisen to integrate general purpose programming languages with databases. This integra-

tion must be more than an interface accomplished through procedure calls or an embedding of a query language in a host language. To accomplish complete integration requires that programming languages be able to manipulate data that persists beyond program execution. This requires a uniform type system capable of supporting object identity, inheritance, and polymorphism [Atki87].

Developments in object-oriented database systems [Kim89] also moved the programming language and database fields closer together. Object-oriented programming languages became the focal point for object-oriented database management system development, with C++ forming the basis for the O++ database programming language [Agra90] and the Gemstone database management system based upon Smalltalk [Butt91], to name a couple. Each of these efforts succeeds by adding persistence to the base language. Other database programming languages, both object-oriented and non-object-oriented, are described in [Banc89], [Banc92], and [Hull90]. Implementation of persistent objects is discussed in [Rose89] and [Dear90].

Many object-oriented database research projects have continued to build upon the relational model (e.g., see [Malh90]). The combination of these two approaches appears to offer many advantages over one or the other paradigm singly [Catt91]. Many of the hindrances in achieving the desired integration involve inadequate language constructs for manipulating both object-oriented and relational data effectively. There have been some reported developments of database programming languages supporting both of these paradigms [Melt92].

Another emerging technology in database systems is the capability of managing large amounts of data distributed over many different physical locations [Ozsu91]. Presently there are working techniques for handling distributed relations but there have not been many solutions for distributing objects, largely because of a failure to standardize upon an

object-oriented data model capable of supporting distributed processing. A major effort, however, has been undertaken by the Object Management Group to address this need [OMG91].

3 The CORAL Programming Language

CORAL (Concurrent Object-oriented Application Language) [Chan90] is a programming system designed to facilitate the construction and execution of sequential, parallel and distributed applications in a transparent manner. In addition to providing essential features (with enhancements) which are present in other object-oriented programming systems, CORAL has the following unique characteristics:

- Subtyping and subclassing are integrated and a type hierarchy is maintained as a sub-hierarchy of its class hierarchy
- Full concurrency is supported among objects and within objects
- Constraints may be used both to ensure correctness and for concurrency control
- Direct interfaces are provided to conventional languages such as FORTRAN and C.

Furthermore, CORAL supports persistence of complex objects and relational database objects, the primary focus of this paper.

A CORAL program is a collection of class definitions and/or object definitions. A class definition is a collection of definitions of attributes, methods, class attributes, and class methods. An object definition is a collection of definitions of attributes and methods. A class attribute and a class method characterize the class as a whole rather than its individual instances.

In CORAL object persistence is specified through classes. There are three kinds of classes: a persistent class, a relational database class, and a transient class, with the latter being the default class. Here we illustrate the definition of transient classes with an example from [Chan91]. Persistent and relational database classes will be discussed further in later sections.

```
class particle
{
    attribute double x;
    attribute double y;
    attribute double z;
    void update
        (in double dx, in double dy, in double dz)
    {
        . . .
    };
};
class:
    attribute double x;
    attribute double y;
    attribute double z;
    void update
        (in double dx, in double dy, in double dz)
    {
        . . .
    };
};
```

This example defines a particle class with attributes x, y, and z. An update method (function) is provided, with no return value (i.e. void return type), and input parameters dx, dy, and dz. There are similar class (aggregate) attributes to represent the geometric center position of all particles (instances of the particle class) and an update class (aggregate) method to update these class attributes.

A class may be defined with generic types for attributes, methods, parameters, or local variables. Generic types must be instantiated with specific basic types or class types at object creation. Similarly, a class may be defined with generic parameters, which may be used in the class definition as variables and which must

be instantiated with specific values at object creation.

CORAL maintains a network-structured class hierarchy with the system-defined Object class as the parent class of all other classes. Classes may be defined as subclasses of one or more classes (i.e., multiple inheritance is supported), in which case they inherit the properties of their superclasses. Inherited properties can be renamed or redefined. An example of this type of behavior is shown below:

```
class binaryGen: randomGen
{
    attribute double percentTrue;
    void Init()
    {
        . . .
    };

    superInit
    from randomGen
    named Init;
};
```

The class `binaryGen` inherits all attributes and methods of `randomGen` and has its own `percentTrue` attribute and `Init` method. The `Init` method inherited from `randomGen` is renamed as `superInit`.

Not every subclass is a subtype of its superclass since the redefinition of its inherited properties may violate the subtype conformity rules [Chan92]. CORAL maintains a type hierarchy in addition to its class hierarchy to provide safety in polymorphism.

In CORAL an object may be defined individually. The advantages are that it is not necessary to define separate classes for objects which have only a single instance and an object may be defined as an instance of a class but with its own unique characteristics. Objects can be defined with attributes and methods. For example,

```
Object particle_1: particle
```

```
{
    attribute double mass;

    double get_weight()
    {
        . . .
    };
};
```

Objects are encapsulated units and can only communicate with each other through message passing, which is represented syntactically using the customary dot notation, `object_name . attribute_name` or `object_name . method_name`. The tight encapsulation of CORAL objects enhances parallel and distributed processing by treating each object as an independent computational unit which can be executed as a single process, thus facilitating program partitioning as well as process communication, synchronization and allocation. The amount of concurrency is limited only by the number of objects and/or available processors. It is important to note that concurrency is transparent as the same CORAL program may be executed sequentially, concurrently, or distributedly without modification. The only additional information required for parallel and distributed execution is a separate process allocation file which specifies the allocation of objects to processes and processes to processors.

CORAL also allows the definition of methods which call external routines written in conventional languages, thereby supporting alternative programming paradigms and legacy software. Data conversions between CORAL and the supported languages are performed automatically.

In addition to the object-oriented paradigm, CORAL supports a number of sub-paradigms within the object-oriented framework. These include the procedural paradigm (through methods), constraint-based paradigm (through constraints and pre/post-conditions), and rule paradigm (through trig-

gers).

Further details about CORAL will be discussed in the following with reference to its features to support persistent objects and relational database objects.

4 Persistent Objects

CORAL achieves the goal of supporting persistent objects in a transparent manner by extending the programming construct of classes with the property of persistence. In CORAL one can designate any class to be a persistent class. A class without specific designation is, by default, a transient class. All instances of a persistent class are persistent objects whose state data are stored in files. At run-time, prior to the execution of a program, CORAL automatically creates all these instances and retrieves their state data from the file. Later, after the execution of the program, CORAL automatically stores their updated state data to the file. If any instance was destroyed during the program execution, CORAL will remove its state data from the file. Also, if new instances are created during the program execution, CORAL will add their state data to the file.

4.1 Data Types

All data types in CORAL are supported for persistent classes. Currently, there are 12 data types as listed in the following:

- boolean
- long
- double
- string
- complex
- class (object identifier)

- array of boolean
- array of long
- array of double
- array of string
- array of complex
- array of class

4.2 Data Definition and Manipulation

Persistent classes are defined in the same manner as transient classes. The only exception is the designation of persistence through the use of the keyword *persistent* as shown in the following:

```
class employee
  - persistent
{
  attribute string name;
  attribute long number;
  attribute double salary;
};
```

Persistent objects can be manipulated like any transient objects. Therefore, the full programming capabilities of CORAL are supported for persistent objects. In addition to navigational accesses through object references which are common in object-oriented systems, CORAL provides the following set-oriented access functions based on classes and objects:

1. class hierarchy
ancestorClassOf, descendantClassOf, classOf, classInstanceOf, subclassOf, superclassOf
2. type hierarchy
ancestorTypeOf, descendantTypeOf, typeOf, instanceOf, subtypeOf, supertypeOf

3. attribute access
attributeNames, attributeTypes,
attributeValues, setAttributeValues

An example is shown in Figure 1. This CORAL program segment declares an object `t_employee` with a function `Init` to create, access, destroy and print employee objects, as defined earlier. The local variable `'e'` is declared as an employee object, and then twice assigned to a newly created employee object. Each invocation of the `Create` method instantiates the object of the given class, `employee`, and two different employee objects are created. The `numberOfInstances` method applied to the class `employee` returns the number of employee objects in existence, hence `emp` will be an array of exactly the size needed to hold the employee objects. The application of the `instanceOf` method will then instantiate the array to the set of employee objects in existence. The `Destroy` method destroys the object associated with `emp` [1]. Since each invocation of the function `Init` creates two new employee objects and destroys one (the “first” existing one), executing this program repeatedly will increase the number of persistent employee objects by one each time.

4.3 Schema Evolution

An important consideration in supporting persistent objects is that the definitions of persistent classes may change over time. Therefore, the persistent objects may become inconsistent with the new definitions. CORAL provides a schema evolution mechanism to address this issue.

The schema evolution mechanism is designed to provide both transparency and user control. If the new schema of a persistent class is correct and a user confirms it, CORAL will perform automatic adjustments and conversions on the outdated persistent objects to make them consistent with the new schema. On the other hand, if the new schema is in er-

ror, a user can override the automatic transformation process such that the original schema and persistent objects remain intact.

The structure of the schema evolution mechanism is shown in Figure 2. The schema evolution mechanism consists of four components: schema analyzer, user confirmation, object transformer, and object re-organizer.

The schema analyzer analyzes the difference between the old schema and the new schema. If any of the following is detected, it indicates that automatic transformation is required:

- added new attributes
- deleted attributes
- changed attribute names
- changed attribute types
- changed order of attributes

The user confirmation component interacts with a user to confirm that the automatic transformation process should be carried out on the affected persistent objects.

The object transformer performs automatic adjustments and conversions on the affected persistent objects such that they become consistent with the new schema. Type conversion is done if needed.

The object re-organizer re-structures the file such that it is consistent with the new schema.

5 Relational Database Objects

Applications using relational databases to store persistent data have been developed for more than a decade and a large amount of data has been accumulating in the relational databases. Traditionally in order to access this

data from an application, one must use the programming language interface facility provided by the appropriate relational database management system. This requires considerable expertise and also the interface facility is very much database-vendor dependent.

CORAL provides a transparent means for accessing and storing persistent data stored in relational databases through the use of database classes. Any class in a CORAL program may be designated as a database class. All instances of a database class are persistent and their state data (or part of their state data) are stored in a relational database.

In general the relational database already exists and is shared with other applications. As such, a database mapping file must be defined which specifies the mapping between each CORAL database class to one or more (in the case of join) relational tables. In addition, the mapping between the attributes of each database class to the columns of its corresponding table(s) must also be defined.

If, on the other hand, the relational database is not shared with other applications, the database mapping file is not required. In such cases CORAL will automatically create the relational tables based on the definitions of the database classes, if they do not already exist.

5.1 Data Types

Due to the limitation of commercial relational database management systems, only attributes of data types which are supported by the relational database systems can be made persistent. These data types include:

- long
- double
- boolean
- string

A database class, however, can be defined with attributes which are of other data types, in which case these attributes are transient in nature.

5.2 Data Definition and Manipulation

Database classes are defined in the same manner as transient classes. The only exception is the designation of persistence through the use of the keyword *database* as shown in the following:

```
class employee
    - database
{
    attribute string name;
    attribute long number;
    attribute string dept;
    attribute double salary;
};
```

In the case where a database mapping file is needed, it must specify the mapping between each CORAL database class to one or more (in the case of join) relational tables. In addition, the mapping between the attributes of each database class to the columns of its corresponding table(s) must also be defined. An example is illustrated in the following:

```
database
class employee
    table emp, dept
    condition
        "emp . deptno = dept . deptno"
    attribute
        name
            table emp
            column ename ;
        number
            table emp
            column empno ;
        dept
            table dept
            column dname ;
        salary
```

```

        table emp
        column sal ;
end

```

Relational database objects are treated similarly to any other CORAL object. In addition to navigational accesses through object references which are common in object-oriented systems, CORAL provides many set-oriented access functions based on classes and objects. These have been described earlier in relation to persistent objects.

Since relational databases are likely to be shared with other applications, CORAL provides high-level database operations to allow a user explicit control over the definition and manipulation of relational databases associated with database classes. The operations related to data definition are:

- existence check
- create
- delete
- type check

and those related to data manipulation include:

- lock
- retrieve
- update
- commit
- abort

An example for illustrating the use of data manipulation operations is shown in Figure 3. The actions being taken by this CORAL program segment are the same as for the example given with persistent classes earlier. However, to interface with the relational database in a concurrent environment, we first lock the

database object using `db_lock` and then retrieve it using `db_retrieve`. After making the necessary modifications locally, `db_update` updates the database to reflect these modifications and `db_commit` commits to the transaction. The CORAL db commands are translated into corresponding SQL commands to control the relational database being manipulated.

6 Summary and Conclusions

CORAL has been shown to be a powerful programming system by making persistent object management an integral part of a programming system. Since both persistent objects of a complex nature and relational database objects are included, CORAL programs may use both object-oriented and relational database paradigms within a single integrated framework.

A uniform interface is provided for the definition and manipulation of all objects, including transient, persistent and relational database objects. For persistent and relational database objects, CORAL takes care of all the details regarding the storage and retrieval of their state data in files or databases. This makes all storage aspects of CORAL objects transparent to the applications programmer.

The integration of relational database technology into CORAL using the established SQL paradigm allows CORAL applications to be constructed on top of existing SQL systems, thereby extending those systems with support for complex object manipulation.

CORAL is a stand-alone system for database processing but with its interface to other programming languages, it is also an attractive addition to any existing system which has need for database processing. We expect that integration of existing systems with CORAL would be very convenient in contrast with integrating other database management

systems.

Support for parallelism and distribution among CORAL objects also allows concurrent and distributed access to databases using CORAL. Any CORAL objects, including persistent objects and relational database objects, can be distributed among remote nodes of a multi-computer network for execution. One can therefore develop applications which utilize distributed persistent objects and distributed relational database objects which access relational databases, working cooperatively to solve computational problems. We believe that CORAL is the only existing database language which integrates object-orientation, relational database technology, and parallel and distributed computing.

Presently we are designing the necessary concurrency control mechanisms to manage persistent object distribution in the appropriate way. Distribution of relational database objects is currently handled through the underlying SQL database manager that CORAL interfaces with.

We are standardizing the CORAL language syntax and semantics to be compatible with CORBA IDL, the Common Object Request Broker Architecture Interface Definition Language [OMG91], thereby keeping CORAL at the forefront of modern database programming languages. It is also our plan to extend CORAL by providing capabilities for inferencing and deductive rule-based computation to support knowledge-based applications.

Acknowledgements. The authors are grateful to Peter M. Hirsch and Frank C. Tung for managerial support during this project. Yeturu Aahlad, Steve Balzac, Dick Damian, Alex Hurwitz, Young Rho, Jeong Seo, and Yaron Wolfsthal have contributed to the development of CORAL.

References

- [Agra90] Agrawal, R. and Gehani, N. H., "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++," in [Hull90], 1990, pp. 25-40.
- [Atki87] Atkinson, M. P. and Buneman, O. P., "Types and Persistence in Database Programming Languages," *ACM Computing Surveys* 19, 2 (June 1987), 105-193.
- [Banc89] Bancilhon, F. and Buneman, P., *Advances in Database Programming Languages*, ACM Press, New York, 1989.
- [Banc92] Bancilhon, F., Delobel, C., and Kanellakis, P., *Building An Object-Oriented Database System. The Story of O2*, Morgan Kaufmann, San Mateo, CA, 1992.
- [Butt91] Butterworth, P., Otis, A., and Stein, J., "The Gemstone Object Database Management System," *Communications of the ACM* 34, 10 (October 1991), 64-77.
- [Catt91] Cattell, R. G. G., *Object Data Management: Object-Oriented and Extended Relational Database Systems*, Addison-Wesley, Reading, MA, 1991.
- [Cham76] Chamberlin, D. D. et al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control," *IBM Journal of Research and Development* 20, 6 (1976), 560-575.
- [Chan90] Chang, D. T., "CORAL: A Concurrent Object-Oriented System for Constructing and Executing Sequential, Parallel and Distributed Applications," *Proceedings of the*

- 1990 ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming, 1990, pp. 26-30.
- [Chan91] Chang, D. T., "Type Systems for Object-Oriented Languages: The CORAL Instance," Proceedings of the 1991 IBM Programming Language Technology Inter-Technical Liaison Conference, 1991, pp. 119-134.
- [Chan92] Chang, D. T., "Productivity through Object-Oriented Programming: The CORAL Approach," Proceedings of the 1992 IBM Software Engineering Inter-Technical Liaison Conference, 1992.
- [Dear90] Dearle, A., Shaw, G., and Zdonik, S., eds., *Implementing Persistent Object Bases: Principles and Practice*, IEEE Computer Society Press, Washington, D. C., 1990.
- [Hull90] Hull, R., Morrison, R., and Stemple, D., eds., *Proceedings of the Second International Workshop on Database Programming Languages*, Morgan Kaufmann, San Mateo, CA, 1990.
- [Kim89] Kim, W. and Lochovsky, F. H., eds., *Object-Oriented Concepts, Databases, and Applications*, ACM Press, New York, 1989.
- [Malh90] Malhotra, A., et al., "Object Oriented Access to a Relational Database," Proceedings of the 1990 IBM Advanced Database Technology Inter-Technical Liaison Conference, 1990.
- [Melt92] Melton, J., ed., *Database Language SQL3, ISO/ANSI Working Draft Document, X3H2-92-155/DBL CNB-003*, July 1992.
- [OMG91] Object Management Group, *The Common Object Request Broker: Architecture and Specification, Revision 1.1*, December 6, 1991. OMG TC Document 91.12.1.
- [Ozsu91] Ozsu, M. T. and Valduriez, P., *Principles of Distributed Database Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [Rose89] Rosenberg, J. and Koch, D., eds., *Proceedings of the Third International Workshop on Persistent Object Systems*, Springer-Verlag, London, 1989.

```

Object t_employee
{
    void Init()
    {
        employee e;

        e = employee . Create();
        e . name = "Susan";
        e . number = 1000;
        e . salary = 2500.5;
        e = employee . Create();
        e . name = "John";
        e . number = 2000;
        e . salary = 3500.5;

        {
            long i;
            long n = numberOfInstances (employee);
            employee emp[n];

            emp = instanceOf (employee);

            employee . Destroy (emp [1]);

            for (i = 1; i <= n; i = i + 1)
                print [label] emp [i] . name, emp [i] . number, emp [i] . salary;
        }
    };
};

```

Figure 1: Persistent Object Example


```

Object t_employee
{
    void Init()
    {
        employee e;

        db_lock employee;
        db_retrieve employee;

        e = employee . Create();
        e . name = "Susan";
        e . number = 1000;
        e . salary = 2500.5;
        e = employee . Create();
        e . name = "John";
        e . number = 2000;
        e . salary = 3500.5;

        {
            long i;
            long n = numberOfInstances (employee);
            employee emp[n];

            emp = instanceOf (employee);

            employee . Destroy (emp [1]);

            for (i = 1; i <= n; i = i + 1)
                print [label] emp [i] . name, emp [i] . number, emp [i] . salary;
        }
        db_update employee;

        db_commit;
    };
};

```

Figure 3: Relational Database Object Example