DESIGN AND IMPLEMENTATION OF AN OBJECT DEFINITION LANGUAGE

by

Yuan Yao

B. Sc., Peking University, China, 1985

M. Eng. Academia Sinica, China, 1988

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
December 1993

APPROVAL

Name: Yuan Yao

Degree: Master of Science

Title of thesis: Design and Implementation of Object Definition Language

Examining Committee: Dr. Jiawei Han, Associate Professor
Chair

Dr. Wo-Shun Luk, Professor
Senior Supervisor

Dr. Nick Cercone, Professor
Co-Senior Supervisor

Dr. Fred Popowich, Assistant Professor
Supervisor

Dr. William S. Havens, Associate Professor
External Examiner

Date Approved: Dec. 10, 1993

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Design and Implementation of An Object Definition Language.

Author: _____
(signature)

Yuan Yao
(name)

Dec. 10 1993
(date)

Abstract

Object-oriented databases (OODB) have emerged to be a very active research area in database systems. Complex objects and behavior encapsulation are two very important concepts in object-oriented data modelling. In previous approaches, behavioral aspects of complex objects are generally defined directly in the implementation programming language. It is thus difficult for the database users to define application specific semantics for complex objects. In this thesis, we examine the behavioral aspects of complex objects in various OODB applications. In order to enable the database users to specify user semantics, we have designed and implemented a language called ODL (Object Definition Language). By using ODL, the user can define methods for complex objects in terms of methods defined for the constituent objects. An important component of ODL is the implementation of the concept of behavioral constraint, which, contrary to the traditional integrity constraints specified as boolean expressions, represents the relationship between various database operations and user-defined methods. Behavioral constraints can be specified in ODL. We have implemented an ODL code generator, which generates C++ code with ObjectStore DML. This code can be integrated with application programs and in the system thus obtained, the behavioral constraints can be enforced automatically.

Acknowledgements

I am deeply indebted to my Senior Supervisor Dr. Wo-Shun Luk, not only for his technical contribution to this thesis, but also for his patient guidance. His insights of many facets of software systems have always been a source of inspiration, without which this thesis would have not been possible. I wish to express my deep gratitude to my Co-Senior Supervisor, Dr. Nick Cercone, for his generous support and constant encouragement. His advice on research methodology was extremely valuable for me. Many thanks go to Dr. Fred Popowich, who not only guided me in my early research work but also made careful correction to this thesis.

I would like to thank Dr. William S. Havens for serving as the examiner and Dr. Jiawei Han for serving as the Chairman of my defense. Dr. Jiawei Han also gave me several helpful suggestions in the early stage of my research.

I am also grateful to some of my fellow graduate students, especially Jibin Zhan, Charlie Huang and Jiashun Liu for their valuable discussions. Special thanks to Graham Finlayson for his proofreading and important comments.

Finally, I would like to thank my dear wife, Wanli Zhang, for her wonderful love.

Contents

List of Figures

Chapter 1

Introduction

In the last ten years, relational database systems have been the mainstream database management systems, especially for business applications. However, the relational database paradigm is not well suited to many advanced applications. Subsequently various object-oriented database systems (OODBs) have been developed to support these advanced applications such as Electronic Computer Aided Design (ECAD), Mechanical Computer Aided Design (MCAD), Computer Aided Software Engineering (CASE) and Geographical Information Systems (GIS). The data entities in these applications tend to have nested structures, and values of the attributes may be generated by procedures. While it is difficult and inconvenient for the relational data model to capture some of the complex structures and derived values, these structures can be represented easily in an object-oriented database system.

Much research, especially in the academic community, has been devoted to object-oriented database modelling, i.e., capturing the application semantics by means of object-oriented modelling constructs, but little regard has been paid to efficiency.

This research is not directly related to the OODB products in the market. In contrast, OODB vendors are still busy on making their systems fast and robust.

In this thesis, we attempts to bridge the gap between the two communities, admittedly in a small way. We are interested in making it easy for an OODB user to express application semantics in a way accepted by the commercial OODBs.

In this chapter, we will first present the strengths and weakness of the OODBs. After that, the problem areas of OODBs and the objectives of this thesis will be discussed. We will conclude this chapter with the organization of this thesis.

1.1 OODB Concepts

We give a brief overview of the core concepts of the object-oriented database systems, the architecture of the database programming language (DPL) and corresponding applications.

1.1.1 Object-Orientation Basics

With object-oriented data modelling, each data entity is modeled as an *object.* Each object is associated with a unique identifier (*Object Identifier* or OID). Each object has a structural part and a behavioral part. The structural part is a set of attributes and their types. The behavioral part is a set of procedures that operate on the structural part of the object [18].

The objects with same structure and behavior are grouped into a *class* which represents the *abstract data type* (ADT) or template for those objects. The types of

the attributes of an object can also be defined as classes that have complex structures. The nested structure is thus supported. An object with nested structures is usually referred to as a *complex object* or *composite object.* Since the domain of an attribute may be any class, the nested structures of complex objects form a hierarchy, usually called the *class composition hierarchy.*

Object-oriented data modelling allows the database designers to define complicated semantics for objects by associating operations with them. The procedures that generate values for attributes are thus *encapsulated* within the class definition. Procedures associated with the class definition are usually referred to as *methods* or *member-functions.* All the methods defined in a class form a uniform interface by describing the behavioral aspects.

An OODB user may derive a new class from an existing one. The new class is called the *subclass* of the existing class. The subclass *inherits* all the attributes and methods of the existing class, also known as its *superclass.* All the classes in an OODB system form a hierarchy, called a *class hierarchy.*

A database constraint is a condition that any valid database state must observe. In OODB systems, since database operations can be encapsulated with objects and thus stored in the database, the concept of constraint can be extended to include the relationship between operations. Later in this thesis (chapter 2), we show how this concept can be extended to include behavioral aspects.

1.1.2 Database Programming Language

In object-oriented databases, one important modification to the database architecture is the integration of the database with programming language environments. The combination of the object-oriented applications and the object-oriented databases has narrowed the gap between the applications and the databases usually found in relational database systems. A large measure of so called *"impedance mismatch"*, which refers to the transformations between the application data space and the database storage, is also reduced by this combination [5].

In essence, almost all the databases based on the object-oriented data model are what we call *Object-oriented Database Programming Languages* (ODPLs). These systems are based on the object-oriented database programming language architecture: applications are written in a version of an existing object-oriented programming language which has been extended to incorporate database functionalities, such as persistent types, transaction structures, etc. Instead of traditional SQL-like Data Definition Languages (DDLs) and Data Manipulation Languages (DMLs), the databases are defined and manipulated by means of the programming constructs of the host language. Among these ODPLs, the most popular host languages include C++ and Smalltalk. To distinguish the database from the program variables, there are two classes of data for an ODPL: persistent objects and transient objects. The former remains accessible after a user session or application program execution while the latter does not.

Many advantages are derivable from a tight integration of database and programming language environment. In contrast to traditional database systems, an object-oriented database programming language provides a language for database access that is computationally complete, i.e., the database language can perform any operations

that a programming language can.

In relational databases, a data structure (e.g., a linked list) must be converted into a table before it can be stored into the underlying database. In contrast, with an ODPL, the only work required by a programmer to make a data structure persistent is to define it as a member of a persistent class. An ODPL, being object-oriented, is able to associate the procedures with the objects, and store procedures in the database.

As in object-oriented programming languages, the object types are defined as classes. However, apart from the meaning of a user-defined type, a class also represents an *extent*, i.e., the set of objects that belong to this class.

Performance is the most important reason why ODPLs are adopted by many developers of advanced applications. The architecture of ODPLs makes it possible to enhance performance for advanced applications. By using a client cache, the data can be retrieved directly into the application's programming space. This is beneficial especially for the data being used repeatedly.

1.1.3 Problem Areas of Object-oriented Databases

Although current OODB systems achieve very good performance for advanced applications, they are still showing poor performance when used in some traditional applications, such as business applications.

Another problem with current OODB systems is that since they have not been around for as long as the relational database systems have, many aspects of the systems, such as integrity constraints, query languages, and so on, are not very polished. Reliability is also a problem.

By far, the most serious obstacle yet to be overcome by the current generation of OODB is the ease of use. This is the issue addressed by this thesis. By using object-oriented data models, objects in these applications are represented as combinations of various attributes. The operations performed on the attributes are represented as methods (program code). However, the semantics associated with these methods in advanced applications tend to be complicated. With current OODB systems, the complicated semantics of object must be incorporated in the implementation of the methods.

There are several disadvantages to capturing the semantics of objects directly in the code.

- *Extra effort must be made on communication between application users and application programmers.*
- *It is difficult for database users (other than designers) to understand the object semantics.*
- *The implementation of object semantics could be complicated and difficult to integrate with database functionalities.*
- *Any modification to the object semantics must involve manually changing the code by application programmers, which could be complicated and confusing.*
- *Due to individual programming habits of the application programmers, the methods implemented tend to be less organized and may not reflect the object semantics correctly.*

In current OODB systems, these drawbacks are due to the lack of a uniform

language interface for complete object definition. These drawbacks have made object-oriented databases much more difficult for general users to use than relational databases. In this thesis, we will propose a solution for resolving this problem.

1.2 Objectives of the Thesis

In this thesis, we design and implement a language called *Object Definition Language* (ODL) for OODB systems in order to provide a high level and uniform interface for complete object definition.

An object definition can be divided into three parts: the *structural definition*, the *behavioral definition* and the *relationship definition.* Since the structural definition of objects can be specified quite clearly using class definitions as in C++, our ODL will focus on the behavioral definition and the relationship definition of objects. The objectives of designing and implementing ODL are as follows:

- *It should provide a clear and high level definition for object behavior (or methods). It should thus be easier to understand for both the application users and application programmers.*
- *It must be able to incorporate complicated object semantics defined by database designers. Various constraints associated with methods should be easily specifiable.*
- *It must support various behaviors of complex objects. In particular, it must support various relationships between the methods of complex objects and those of their constituent objects.*

- *The implementation of the language should generate program modules that could be integrated easily with the rest of the application programs and the underlying database systems.*

- *The object semantics defined in the language should be correctly reflected and maintained in the corresponding program modules.*

With the database programming language architecture, the database functionalities and the application programs are combined into a single programming space. Application programmers are also responsible for designing the database for the users. Even a very simple database will have to be created by using application programs.

By introducing ODL, we will be able to provide an interface between the database designer and the application programmer. ODL will allow database designers to directly define object behaviors and the associated semantics, especially constraints. In the implementation of ODL, the code for the method will be generated. The constraints specified in ODL will be maintained by the code.

1.3 Thesis Organization

In chapter 2, we discuss the motivation behind this research. By providing a series of examples, we will identify a category of integrity constraint associated with the methods in OODB applications. We will also discuss how to support complex object as views. Issues relating to customized query languages will also be discussed in chapter 2. In chapter 3, we present the design and the syntax of ODL. We will provide several examples illustrating how object semantics and complex object behavior can

be specified in ODL. In chapter 4, we describe the implementation of ODL. We have implemented the ODL on top of ObjectStore, an object-oriented database system. We also discuss the integration of ODL with the rest of application programs. As an example, we have integrated methods specified in ODL code with an application package which supports a customized query language called DSQL. In chapter 5, we give a conclusion and discuss the future work.

Chapter 2

Motivation and Related Work

Many researchers like to divide the short history of database systems into three eras, each of which is characterized by one generation of database system. The first generation database systems can be called generically pre-relational database systems. They are based on network or hierarchical models. There are DDLs and DMLs associated with these systems. The DDLs and DMLs are usually defined as low-level operations performed directly on the physical database. Therefore, such DDLs and DMLs are generally difficult to use. They provide low degree of data independence. For instance, a typical DDL contains many references to the physical organization of the underlying databases.

The second generation of database systems are called relational databases because the data entities are uniformly defined as relations. The definition and manipulation of the databases are simplified to performing operations in SQL on the database tables. In particular, the DDL of a relational database system is very simple. For instance, to define a database in a relational database system is simply to create a set

of relations [8].

The OODBs are known as the third generation database systems. There is no separate language for object definition since the host language is used as the DDL. Clearly, the development of DDL from the second to the third generation database systems is at odds with the development from the first to the second generation. There are obviously many reasons why the DDL has to assume the full facilities as an object-oriented programming language. Some of which have been elaborated in chapter 1. As a result, we need to address fully the question of how a separate high level ODL can be justified.

In this chapter, we discuss our motivation from three perspectives: an object definition language can be used to describe the database constraints; an object definition language can be used to support user views; finally, an object definition language can also be used to support customized query languages.

2.1 Incorporation of Constraints

The term "integrity" refers to the accuracy or correctness of the data in the database. Integrity constraints provide a means [20] of ensuring that changes made to the database by authorized users do not result in a loss of data consistency. Thus, integrity constraints guard against accidental damage to the database.

In relational database systems, changes to the databases are made by using operators defined in SQL. The integrity constraints are associated with operators such as *insert, delete*, and *update.*

With current OODB systems, complicated computations and operations can be defined for objects. However, while all the manipulation and dependencies of an object have been put into the code, the relationships between objects, especially a complex object and its constituent objects are not clear to the user. Besides, almost all the manipulations and the enforcement of the integrity constraints are now left to the user. In this way, the database is only responsible for the storage management. Although this arrangement is quite flexible, this kind of system is potentially very dangerous. The database user has to make great effort to ensure the consistency of the system.

2.1.1 Behavioral Constraints

In OODB systems, changes to the databases are no longer made by using SQL operators. Instead, complicated computation along with database operations are intermixed. As a whole, they are defined as object methods. So the integrity constraints in OODB system are associated with object methods instead of SQL operators.

One of the major differences between OODB integrity constraints and the integrity constraints of relational databases is that the SQL operators are system defined whereas the methods are user defined. In order to see this, let us look at the following example.

Example 2.1.

In MCAD applications, the real world entities can be modeled as 3-D objects. Complex 3-D objects can be divided into simple ones such as `cuboid`, `cylinder`, `pyramid`,

frustum, and sphere. We can define them as classes of 3-D objects and group them into a superclass block. The definition of class block looks as follows:

```
class block{
   private:
      Vertex          block_center;
      STRING          name;
   public:
      void            move(float, float, float);
      boolean         ontopof(block);
}
```

A block has two attributes: block_center of type Vertex representing its center and name of type STRING indicating its name. There are two methods defined for the block. The move will update the position of a block. ontopof is a predicate which tests whether this block is on the top of another block.

Although block only has attributes block_center and name, its subclasses cuboid, cylinder etc. may have additional attributes such as face. Since a block is a complex object which has several other objects as its components (for instance, a cuboid has six face objects), the method move is also responsible for changing the positions of these component objects.

Based on the application semantics, there could be various relationships between the operations move and ontopof. For a block b1 to be moved, if there is currently another block b2 on the top of b1, i.e. b1.ontopof(b2)=TRUE, then a constraint could either be "move b2 as well" or "reject". In the first case, the constraint requires that

block `b2` be moved together with `b1`, thus `b2.move` has to be called. In the second case, the operation of `b1.move` will have to be rejected.

From the example above, we can see that in object-oriented environments the constraints must be associated with the user-defined operations (`move, ontopof`). For the class `block`, the modification of the `block_center` attribute may be implemented using a single database operation. However, the aforementioned two constraints can not be associated with the operation of changing the `block_center` since the modifications of positions for constituent objects (`face` objects) is also involved in the constraints.

There are two important features that make such constraints different from traditional database integrity constraints.

- *They describe the relationships between database operations.*
- *The operations involved are user-defined.*

At any particular moment, all the information about the schema and data stored in a database are referred to as a state of the database. Except for object retrieval, every operation performed on a database state will change the database from the current state to another state. It thus represents a transition of database states. Some of the operations may cause a transition from a valid database state to an invalid database state. In many database applications, such invalid database states are inevitable as interim stages of database transitions. However, these invalid states should not remain in the database persistently. In fact, they must not remain outside a transaction so they are usually eliminated by performing other database operations that cause a transition from an invalid database state to a valid one.

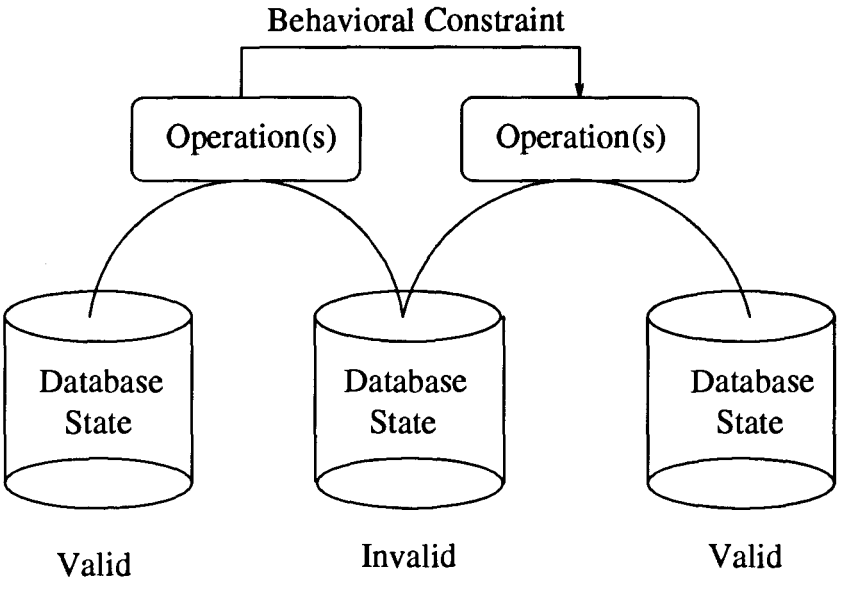

Figure 2.1: Behavioral Constraint

Example 2.1 shows a type of integrity constraint that applies to a user-defined operation that drives the database into an invalid state. The constraint requires that either the operation be rejected or some additional user-defined operations be performed in order to change the database back into a valid state. We call this type of integrity constraints *behavioral constraints* (BC) because they are effective over the behavioral aspects of the the object-oriented data model (Figure 2.1).

2.1.2 Intra-Object and Inter-Object BCs

A constraint that applies to a single object is called *intra-object* constraint. An example of such constraints for a `person` object is that `years-of-schooling` be at least 5 less than `age` [16]. Constraints that apply across objects are called *inter-object* constraints. For example, we may have a constraint that the `age` of a `person` must be at

least 12 greater than the age of any child of person. Similarly, we could also have intra-object behavioral constraints and inter-object behavioral constraints.

Example 2.2.

Suppose we have a class employee in an *Office Information System* (OIS) application defined as follows:

```
class employee{
  private:
      STRING        name;
      int           age;
      POSITIONS     position;
      MONEY         salary;
      ...           ...
  public:
      void          raiseSalary();
      void          raisePosition();
}
```

There are two user-defined methods raiseSalary() and raisePosition(). An intra-object behavioral constraint may require that:

> *"when the* position *of an employee is raised, the* salary *should be raised according to the employee's new position".*

Example 2.3.

In a *Geographical Information System* (GIS) application, we could have classes representing geographical objects. Two such classes are `intersection` and `road`. The attribute `intersections` in class `road` is a set of objects from `intersection` where all the intersections on the `road` object are recorded.

```
class intersection{
  private:
      STRING          name;
      road            rd1;
      road            rd2;
      vertex          location;
      ...             ...
  public:
      intersection    new(road,road,point);
}

class road{
  private:
      SET-INTER       intersections;
      ...             ...
  public:
      void            add_inter(STRING name, point location);
}
```

An inter-object constraint is that:

> *"when a new intersection is created with two roads, this new intersection must be recorded in each of the two roads".*

In this constraint, a method defined in class `intersection` triggers invocations of methods defined in another class `road`. The constraint is therefore an inter-object BC.

2.1.3 BCs for Complex Objects

For any complex object, the difference between its primitive attributes and component objects is significant. For the primitive attributes, the complex object has its own methods of attribute access manipulation. Component objects, on the other hand, are not simple values. Only operations performed on their attributes are meaningful. For those attributes defined as *public*, they can be accessed directly. However, for those attributes defined as *private*, they can only be accessed by invoking the corresponding methods of the class.

The methods of an object represent their behavioral aspects. Because of the *PART-OF* relationships between the complex objects and their constituent objects [19], the complex objects can be viewed as formed by their constituent objects. Therefore, the behavior of a complex object will inevitably affect or be affected by the behaviors of its component objects. This effect is actually determined by the behavioral constraints between the complex objects and their component objects. We now give an example to show this kind of behavioral constraint.

Example 2.4

In an Office Information System (OIS) application, there are two classes `person` and `family` defined as follows:

```
class person{
   private:
      STRING    name;
      INT       age;
      char      sex;
      char      married;
      PERSON*   spouse;
      ...       ...
   public:
      void      marriage_record(char);
};

class family{
   private:
      PERSON    husband;
      PERSON    wife;
      PERSON[]  children;
      float     income;
      ...       ...
   public:
      void      divorce();
```

```
};
```

In class `person`, the attribute `married` indicates the person's marriage status by a single character (`Y`–married, `N`–never married, `D`–divorced, `W`–widowed, etc). The attribute `spouse` is a pointer referencing the spouse object. Because these attributes are declared as private, they can only be accessed by the methods defined in the class. The operation `marriage_record()` is responsible for making modifications to private attributes of class `person`.

Each `family` object is a complex object formed by two objects `husband` and `wife` and a number of their `children`. There is a public member-function `divorce()` which is used to record the divorce transaction for all `family` objects. If the husband and the wife of a family divorce, the method `divorce()` will have to delete the `family` object from the database. However, before doing that, the method has to make sure the modification of the corresponding marriage status information has been changed in the objects `husband` and `wife`. This is a behavioral constraint between a method of class `family` and one of class `person`.

Example 2.5

As another example of a behavioral constraint for complex objects, consider a complex object `cuboid`, which is composed of six objects from class `Face`, while each of the face objects in turn contains four objects from class `Edge`. Each `Edge` object contains two objects from class `Vertex`. The definition for the above database schema is as follows:

```
Class Cuboid {
```

```
  private:
      STRING name;
      STRING color;
      Vertex block_center;
      Face top, bottom, left, right, front, back;
      ...
  public:
      void rotate(coordinate, degree);
      void move(float, float, float);
}

Class Face {
  private:
      Vertex center;
      Vector normal;
      STRING color;
      Edge edge1, edge2, edge3, edge4;
      ...
  public:
      void rotate(coordinate, degree);
      void move(float, float, float);
}

Class Edge {
  private:
```

```
        Vector normal;
        float length;
        Vertex x, y;
        ...
    public:
        void rotate(coordinate, degree);
        void move(float, float, float);
}
```

Operations `rotate` and `move` have been defined for all three classes. Because the faces are components of the complex object cuboid, whenever a cuboid is rotated or moved, the faces that belong to it will also be rotated or moved. Therefore, there is a constraint between the operation of `rotate` of the cuboid and that of its faces. Similar constraints exist between the operations of `Face` objects and that of their constituent `Edge` objects.

2.2 Related Literature on Constraint Specification

In essence, any database constraint can be maintained by implementation code. In relational databases, there are a limited number of operations supported by the system; consequently the constraints defined over the database states can be easily checked in between the executions of these operations.

2.2.1 Relational Database Constraints

Two important integrity maintenance facilities in relational DBMS are attribute-based constraints and database triggers. However, it can be shown that they are not suitable for supporting behavioral constraints.

In relational databases, data are generally simple values associated with attributes in relations. Constraints are usually expressed as boolean expressions representing conditions on data values. For a simple example, in the Sybase relational database system, the users may create a constraint on the `user_id` of one of their tables by defining a rule as follows:

```
create rule user_id
as @user_id in ('1389', '0736', '0877')
```

This rule defines a constraint that the `user_id` has to be one of the three numbers. Any modification to the `user_id` will be checked against this rule. We call this type of constraint *attribute-based* or *state-based* because it is usually expressed over the attributes (although sometimes the attributes may be complex) in the database or the states of the database.

Relational triggers, another type of integrity constraint, specify the relationships among database operations. For example, the syntax for creating triggers in the Sybase relational database is as follows:

```
CREATE TRIGGER [owner.]trigger_name
    ON [owner.]table_name
    {FOR {INSERT, UPDATE, DELETE}
```

```
AS SQL_statements |

FOR {INSERT, UPDATE}
AS
IF UPDATE (column_name)
[{AND | OR} UPDATE (column_name)]
SQL_statements}
```

From this syntax we can see that the triggers may only defined over operations `INSERT, UPDATE, and DELETE`.

In relational databases, the triggers can be maintained and performed by the database management systems. This is because the semantics of all of the operations are well defined by the system. However, in an object-oriented database, the operations performed on the objects are no longer defined by the system. The persistent programming language that a user uses is so powerful that any complicated operations can be specified and performed. While this is very convenient for the database user, the consistency of the database becomes more vulnerable because the application programmer may easily violate the semantics of the objects. Due to the complexity of the object structure, it may be very expensive for the OODB to perform integrity checking. Indeed, it is impossible to perform such a checking if the user semantics of the objects are not known to the OODB.

2.2.2 Constraint Maintenance Mechanisms

There have recently been many proposals for supporting database integrity constraints for advanced applications. In general, they can be classified into three different approaches.

The first approach tries to capture all the integrity constraints by using a uniform rule system [12, 28, 29]. The second approach insists on early detection and high level expression of the constraints. It involves associating the constraints with the class definition or providing a separate algebra for representing the constraint [7, 11, 13, 16, 17, 25, 31]. The last approach heavily relies on the so-called active database model, where operations can act as triggers that cause the execution of other operations [2, 9, 10, 21, 22]. We will review three systems that are representative of each approach: the POSTGRES Rule System, ECA model and Constraint Compilation of Ode.

2.2.3 POSTGRES Rule System

Incorporating a rule system into a DBMS represents the effort of applying techniques of artificial intelligence to database systems.

A successful example of an embedded rule system is the POSTGRES database system (which is based on an extended relational data model[28, 29]). The rule system in the POSTGRES is very powerful. It subsumes not only the general scheme of integrity constraints but also other concepts such as materialized views and special procedures.

In POSTGRES, the general format of the rules is as follows:

```
DEFINE RULE rule-name [AS EXCEPTION TO rule-name]
ON event TO object [[FROM clause] WHERE clause]
THEN DO [instead] action
```

In this format, an **event** can be one of the system defined operations such as **retrieve, replace, delete, append**, etc. An object is either a relation name or a relation name followed by a column name. The semantics of a rule is that when an operation is to be performed on a **CURRENT** tuple (for retrieves, replaces and deletes) and **NEW** tuple (for replaces and appends), all the rules that are defined on the event and the **CURRENT** or the **NEW** tuple will be found and the action parts will be executed. The rules are defined in such a way that actions could be made to the database either as an addition to or as an replacement of the event. If an action part is specified as **instead** then the event part will not be performed.

POSTGRES takes a traditional production system approach to rule handling. There are two ways to enforce the rule semantics in POSTGRES. A *"tuple level rule system"* supports the rule system at execution time. When the execution engine is performing an event on the **CURRENT** tuple and/or the **NEW** tuple, the rule manager is responsible for finding all the rules defined on these tuples. The execution engine will then perform the event and the actions according to the rules. The *"query rewrite implementation"* is a module between the parser and the query optimizer. When a POSTGRES query is processed by this module, it will be transformed into an alternate form according to the applied rules.

The POSTGRES rule system has several limitations on the application and maintenance of its rules:

- The events and the actions must be system defined operations.
- There is an overhead of performing the constraint maintenance since a separate rule system is used to undertake the rule management.
- The rule propagation may happen in POSTGRES system, i.e., an action of a rule may fire another rule. The rule propagation, however, is uncontrolled.

2.2.4 ECA Model of HiPAC

In contrast to relational database where data manipulation is performed in response to explicit requests from applications, an active DBMS is one which automatically executes specified actions when specified conditions arise. An architecture for an active DBMS that supports Event-Condition-Action (ECA) rules as a formalism for active database capabilities has been proposed in the HiPAC project [10, 22].

The major concept in the ECA model is the ECA rule. An ECA rule consists of three parts: an *event*, a *condition*, and an *action*. When the event occurs, the active DBMS will evaluate the condition; if the condition is satisfied, the action will be executed.

Events may be primitive or composite. Primitive events are database operations that can be detected by components of the DBMS, or abstract events that are signaled by external processes. Composite events are constructed from the primitive events using algebraic operations (e.g. disjunction, sequence, closure, etc.).

The condition is simply a collection of queries that are evaluated when the rule is triggered by its event. The action is a sequence of operations. They can be database

operations or external requests to application programs.

There are two coupling modes in the ECA model. The E-C coupling specifies when the condition is evaluated relative to the transaction in which the triggering event is signalled. The C-A coupling specifies when the action is executed relative to the transaction in which the condition is evaluated. Both of the two coupling modes may have three possible options: immediate, separate and deferred.

One of the advantages of the ECA model is that it separates the event and condition parts of rules. The separation is useful because the events and the conditions play different roles: events specify *when* to check if a rule should fire; conditions specify *what* to check.

There are several differences between the POSTGRES rules and the ECA rules. The event of the POSTGRES rule can only be a single database operation whereas the ECA event may be various composite database operations. The action of the POSTGRES rule is also a single database operation whereas the ECA action can be a sequence of operations or even user programs.

Although the ECA model has extended the concepts of event and action, there are still some limitations. The events are still system defined operations. The condition checking can only be performed after the event has been identified. Like the POSTGRES rule system, a separate rule engine is still needed for perform the rule management. Uncontrolled rule propagation is also possible in the ECA model.

2.2.5 Constraint Compilation in Ode

Ode is an object-oriented database system defined, queried and manipulated using the database programming language O++, an extension of the object-oriented programming language C++. Ode provides facilities for associating constraints and triggers with objects [13, 14, 16].

In Ode, constraints are associated with the class definition. The constraints are specified in a *"constraint section"* using the following form:

```
constraint:
        constraint_1 :  handler_1
        constraint_2 :  handler_2
        ...
        constraint_n :  handler_n
```

Constraints are said to be *hard* if the constraint checking is performed immediately after each database operation even though the end of the transaction is not reached. If the constraint checking is deferred to the end of the transaction, then the constraint is said to be *soft*. No structure of nested transactions has been employed in the Ode constraint maintenance mechanism.

A recent version of the Ode constraint maintenance mechanism can accept higher level constraint specifications. For example, a constraint:

> *there is at least one employee whose salary is more than half the department head's salary*

can be specified as:

```
foreach d in Dept (thereis e in d->emps[[]]
    (e->sal() > d->head->sal()/2))
```

The constraint compilation mechanism accepts constraint specifications and transforms them into an object-oriented representation associated with class definition. This representation is localized and more efficient.

In Ode, a trigger is also specified in the class definition. Each trigger has two parts: a *condition* and an *action*. The format of the trigger is quite similar to that of the constraint. However, there is a difference between a trigger and a constraint. A constraint action must maintain database integrity. The handler of a constraint is executed within the transaction where the constraint is violated. The trigger actions, on the other hand, are not executed in such transactions.

Because the trigger actions are executed in a separate transaction, as suggested in [13], the trigger facility can not be used for integrity maintenance. In fact, even if the transaction being triggered has been aborted, the triggering transaction will not be affected. Therefore, the Ode triggers can only be used to specify independent actions which, whether executed or not, will never affect the validity of the database. Such trigger mechanisms are comparatively less meaningful for integrity maintenance.

Compared to the ECA model, the constraints and triggers in Ode only deal with C-A coupling. The events are not explicitly specified. Instead, an execution mechanism will perform the checking just prior to the termination of each member-function. Since each constraint or trigger is associated with a class definition, for each member-function, only those constraints and triggers associated with the class are checked.

One problem with this approach is that the E-C coupling is not explicitly specified,

it may thus be difficult to distinguish which event will trigger which action. For a simple example, if two member-functions f_1 and f_2 are both defined in class C, two constraints c_1 and c_2 are defined for f_1 and f_2 respectively. If c_1 and c_2 have the same condition part, since both c_1 and c_2 will be associated with C, when f_1 is invoked and the condition is true, both actions in c_1 and c_2 will be executed, which could potentially make the database inconsistent.

Since it is not clear to the system how the transaction structure is incorporated with the methods, and also because the checking is performed at the end of each method, a problem may arise when a method is not embedded properly in a transaction.

2.3 Complex Object as Views

The database architecture proposed by the ANSI/SPARC Study Group on Data Base Management Systems (the so-called ANSI/SPARC architecture) [8] is divided into three *levels*, known as the *internal*, *conceptual*, and *external* levels. The *internal level* is the one representing the structure of physical storage. The *external level* represents the users' viewpoint to the data. The *conceptual level* cames between the internal and the external levels — it is an abstract representation of the entire information content of the database. Further it is not confined to any particular language or hardware. The ANSI/SPARC architecture is a general framework for database management systems.

The external level of a database system usually has a language as the interface between the user and the system. Different users of the database may have different languages as their *host languages*. A data sublanguage – i.e., a subset of the total

language that is concerned specifically with database objects and operations, is usually embedded within the host language which is responsible for providing various nondatabase facilities, such as local variables, computational operations, if-then-else logic, etc. Most traditional databases support a *loosely coupled* data sublanguage and host language in the sense that the two languages are clearly and easily separable. In OODB systems, especially those being called database programming languages, the data sublanguage and the host language are *tightly coupled* [5].

OODB systems are generally concerned with objects as data. However, there is no uniform external level interface between the user and the system.

Database views provide the user with various exposures to the same set of data stored in the database. There are several important reasons for using views in the database systems. Firstly, views enhance logical data independence, which means the way a set of data being used by a user is independent of the structure how it is stored in the database. Secondly, views can exclude information that should be hidden from some users. These users would only have access to selected views instead of the whole database concept model.

There has been extensive research work on relational views [3, 4, 6, 26]. In relational databases, a view is simply a derived relation and is specified by a relational query. The view mechanism maintains the definition for the view and will return the view table when the view name is referenced.

Because of the similarities between a relation and the structural aspect of a class, a similar view concept has been defined in terms of a *virtual class* [1] for OODBa. There have been several approaches for supporting views based on this definition.

[1] With different meaning from the "virtual class" used in C++

2.3.1 Class Integration

Several authors have proposed class integration as an approach for supporting object-oriented views [1, 27, 30]. The basic idea of class integration is to insert the virtual class into the conceptual schema, i.e., the class hierarchy.

The problem of building a view mechanism has been tackled at the schema level [30]. Aside from the concept of virtual class, [30] proposed the concepts of the *virtual schemata* and *base schemata.* The virtual schemata are defined as a different classification of the root class which may be chosen by some of the users. This classification corresponds to the intuition of some users. In [30], an infinite set of objects U_∞ is defined to represent the set of all possible objects in a database. Each class C has a qualifying predicate P_C associated with it:

$$P_C : U_\infty \rightarrow \{true, false\}$$

Obviously, objects in U_∞ will not be associated with any class C, rather, any virtual class V in a virtual schemata will choose objects from U_∞ by its qualifying predicate P_V. Although the theoretical model works fine, the approach of [30] may not be applied practically because the object universe U_∞ is difficult to implement in a real system.

In [1], a framework for a view mechanism on top of the O_2 object-oriented data model was proposed. The specifications of views are extended by introducing *class generalization* and the *behavioral generalization* as ways of populating the virtual classes. Two rules were proposed for computing the new class hierarchy for the virtual classes which only contain the existing objects. In other words, when there is a one-to-one mapping between the view objects and the objects in the base classes the view

could be integrated with other base classes to form a new schema.

Suppose a view v is defined by including classes c_1, ..., c_k and objects from classes c_{k+1}, ..., c_n, then the class v will be placed at a position in the class hierarchy such that $c_1, ..., c_k$ are its subclasses, and all superclasses of $c_1, ..., c_n$ will also be its superclasses. The class integration of [1] fails to be a uniform approach because it does not deal with the situations when new or virtual objects, are included in the views. Moreover, because there is no description about the physical structure that the view objects associated with the virtual classes, it is not clear how the integration will benefit the retrieval of the view objects as well as how the consistency between the views and the base classes [2] could be maintained.

Another framework for object-oriented views known as *MultiView* was proposed in [27]. The model is similar to that of O_2 in that it also performs the process of class integration. The purpose of the integration is to find a most appropriate location in the schema for a virtual class. However, the meaning of "most appropriate" is different from that of O_2. For each virtual class VC, a set of direct superclasses and a set of direct subclasses can be defined using an object algebra. Intuitively, S is a direct superclass of a view VC if there is no superclass of VC which is also a subclass of S. Similarly, S' is a direct subclass of a view VC if there is no subclass of VC which is also a superclass of S'.

According to the approach in [27], a virtual class VC will be placed at the location in the schema directly below all of its direct superclasses and directly above all of its direct subclasses. Views with virtual objects could also be treated uniformly in this approach of class integration. But still, the structure of how objects are associated

[2] The classes from which the view is generated

with the classes in the schema is not clear. Therefore, as in [1], the question of how the view objects will be retrieved and how the consistency could be maintained remains unknown.

2.3.2 Function Application

A view mechanism on an object-oriented data model FUGUE has been presented in [15]. FUGUE is an object/functional model based on three primitive types: *Object, Function*, and *Type* as well as on the concept of *function application.* In the FUGUE model, a view is defined as a tuple of two sets, one of types and the other of objects, i.e.

View V = [T:{types}, O:{objects}]

This definition of view is similar to the concept of so-called *type-closed* view in [27]. All views defined in the FUGUE model must satisfy the *type-closed condition*, namely, for all $o \in O$, $type(o) \in T$, for all $t \in T$ and for all $f \in functions(t)$ where f : $signature(f) \rightarrow t'$, $t' \in T$. T contains those types that have already been defined in the base views (base classes) as well as derived types generated from the base views. The derived types are defined as data abstractions over some types in the base view. Similar to the view model in [1], views in FUGUE may also be populated by executing queries over the base view. The system will define the functions representing the mapping between view objects and base objects according to the queries. The system will also generate functions representing the queries that are issued against the views.

By function application, the functions will be applied to the functions representing the view-base mapping and the queries can thus be performed directly on the base view. When a view object corresponds to multiple base objects in the view-base mapping, OIDs (object identifiers) of those base objects will be stored in the tuple representing the view object. The mapping function is responsible for extracting the OIDs from the tuple.

The FUGUE view model depicts the nature of the view-base relationship. The function application mechanism is similar to a query rewrite process in [28]. In order to keep the consistency between views and base, the system only accepts *well-formed* views. A view is said to be well-formed if the functions that are allowed to perform on it are *type-closed, value-closed* as well as *equivalence-preserving.* The problem of consistency still remains when the base is updated. If the mapping functions perform qualifying predicates each time a query for a view is evaluated, then this approach would fall into the same category as a materialized view mechanism. Also, there is no description about how a view could be detected to be well-formed.

2.3.3 Supporting Complex Objects

The traditional view concept reflects part of the external level in the ANSI/SPARC model. However, because of the tight coupling of the data sublanguage and the host language in a uniform language interface, together with the embedding of the behavioral aspects in the object-oriented data model, the traditional view concept is not very useful for advanced applications. In contrast, advanced applications require that complex objects be supported by the system to form various user views at the external level.

In the traditional class composition hierarchy, the arc from a complex class to its constituent classes represents the PART-OF relationship. However, the semantics of the PART-OF arc can be extended by modifying the methods of the complex class. Arbitrary relationships between the complex class and the base classes may be implemented. The qualifying predicate can be implemented in the constructor of the complex object. Destructor and other methods can be implemented similarly. In fact, with the DPL architecture, all user views can be implemented by the application programmer by defining complex objects.

With the ODPL architecture, the concept of views can be replaced by an extension of the complex object. A formal comparison of these two concepts, however, is beyond the scope of this thesis.

In order to keep the database consistent, not only must a user view map between its own structural part and those of its base classes, it must also maintain a mapping between its behaviors and those of the base classes.

Languages have been proposed for constructing virtual classes in [1, 27]. However, none of them can be used for describing the mapping of behaviors. The function application proposed in [15] can only handle limited behavior mapping. With the object-oriented data model, the virtual class construction corresponds to the construction of complex objects. Since a complex object is generally constructed by selecting appropriate constituent objects that satisfy certain semantic requirements (qualifying predicates), corresponding constraint checking must be performed.

Another operation universally defined for user views is *deletion.* The deletion of views requires that the corresponding objects in the source classes must also be deleted. This can be represented as a behavioral constraint between the delete method

of the virtual class and that of the source class.

Complex objects correspond to so-called *updatable views*, i.e., any updates made to an (updatable) view object will be reflected in their base objects. This requires that the mapping between the methods of complex objects and those of its constituent objects must be maintained. Since the mapping can be represented as a behavioral constraint, it is only possible to support updatable views by providing a high level specification which is capable of expressing behavioral constraints for the complex objects.

2.4 Supporting Customized Query Languages

Another motivation for designing an Object Definition Language is to support a Customized Query Language (CQL) for advanced database applications.

There have been complaints about the lack of query languages in OODB systems. Some recent DPLs do provide a query language, however, their capabilities vary greatly. Moreover, for advanced database applications, the declarativeness of a query language is incompatible with the procedurality of a programming language. Further, a query language may compromise the high performance offered by a database programming language. This is the motivation of using *Customized Query Languages* [33].

There are several advantages of CQLs over generic query languages:

- *Embodying Application Semantics*: More meaningful queries may be supported by incorporating object semantics with query language constructs.

- *Customized Query Optimization*: Many spatial and multimedia applications require specialized indexing techniques which can be made transparent in a CQL.
- *Heterogeneous Databases Accessing*: For applications that must access multiple databases of different types, a CQL isolates different application users from the database types.

The main disadvantage of CQLs is that its design and the implementation can be complex where complex objects are involved.

A CQL can be seen as a set of operations defined specifically on a data model. All the operations of a CQL should be supported by methods defined for the underlying objects. The mapping between CQL operators and the methods requires that object semantics be incorporated in object methods. If the object semantics can be correctly incorporated, the CQL may be supported.

As an example, consider the `Cuboid` complex object described in Example 2.5. A simple customized query language could allow operations on cuboids that have been identified by a predicate. This predicate can be a simple one (for instance, color = "Red"). It may also be a complex one (*"on the top of"* a cuboid named "Apollo"). Various computations of so-called "boundary representation" may be involved in order to build the complex predicate "ontopof". The following customized query [32]:

```
move o1 to o2
where
o1: name( o1 ) = "Apollo",
o2: ontopof( o3 ) = o2,
o3: name( o3 ) = "Moon"
```

```
go
```

moves the cuboid named `Apollo` to the top of cuboid named `Moon`.

Building a CQL involves two tasks: implementation of the operations on the complex objects and predicate processing modules. The latter are quite generic, as they can be processed in the same way as the WHERE clauses are processed in SQL. Thus they can be pre-built.

If the operations can be defined by the users for the complex objects in an object definition language, then the task of building a CQL is essentially one of linking the predicate processing modules with the methods for the corresponding operations of the complex objects. Consider the processing of the above query, we need a **move** method on an object identified by the where-clause. If the user defines all the methods via an ODL for the operations on the complex objects, the task of implementing a customized query language on the complex object can be greatly facilitated.

Chapter 3

Object Definition Language — ODL

3.1 Overview

We have discussed motivations for proposing an object definition language in OODB systems. The major advantage of providing such a language is that the user may use a high level representation to specify complete object definition. An object definition has three components:

1. The structural aspects of objects,
2. The behavioral aspects of the objects, and
3. The relationships between objects.

The definition of the structural aspects of objects is similar to the definition of persistent data structures in an OODB system. This component of ODL simply specifies the attributes names and their types. The attributes of a complex object may be an object, or of pointer types. In actual implementations, the structural aspects of objects can be defined as class definitions. Typically, these definitions are written in header files and stored separately from other parts of object definitions.

The signatures of the methods are also included in the structural part of object definitions, however, the actual definitions of the methods are usually provided separately. We will not elaborate further on this component of ODL since they are well defined in most of the OODB systems.

The behavioral component of ODL defines the operations which are on complex objects. Key to the design of behavioral component of ODL is the perception that the behaviors of complex objects can be defined in terms of the behaviors of their constituent objects. The constraint rules are designed such that the constraint checking may be performed either before or after the event occurs. Complicated user semantics can thus be defined as constraint rules.

The relationship component of ODL is needed primarily for the purpose of specifying constraints associated with the behaviors of the objects. As discussed in Chapter 2, the behavioral constraint is a type of database constraint usually found in OODB applications. Behavioral constraints are effectively the relationship definition and are associated with behavioral definition of ODL. By doing so, we are able to define user-defined methods as the event of our constraint rules.

In this chapter, we describe the design and the implementation of the behavioral component of the ODL. In the following sections, ODL is used to refer only to this

part of the object definition language. To justify our design, at the end of this chapter we will show how the ODL is applied to several examples.

3.2 The General Scheme

An ODL specification has two parts: a *head* and a *body*. The ODL head starts with a class name indicating the class for which the method will be specified. The class name is followed by the method name and the parameter list. The class name and the method name is divided by two colons similar to the definitions of member-functions in C++.

Within the parameter list, types are associated with the parameters. By introducing such a parameter list, the head appears similar to a function definition and is thus natural to the programmers. Also, the specification of both manipulation and behavioral constraints in the body can be simplified by using parameters introduced in the head.

An ODL body is specified within a pair of brackets. Each ODL body has two sections: a *do* section and a *with* section. Statements in the *do* section are function calls that represent the decomposition of the method. The functions invoked by these statements are user-defined procedures. This decomposition enables the users to clearly describe the relationships between methods of a complex object and that of its constituent objects.

In the ODL, we not only want to specify the behavior decomposition of methods, we also want to capture the behavioral constraints and express them in higher level

language constructs. The behavioral constraints are expressed as constraint statements that are grouped in the *with* section of the ODL body. The general format of an ODL specification is as follows:

```
<type> <class>::<method> ( <parameter_list> )
        {
        DO:
            <action>;
            <action>;
            ...
            <action>;
        WITH:
            <constraint>;
            <constraint>;
            ...
            <constraint>;
         }
```

By using ODL, the database user is able to define complete contents for a method. After being accepted and processed, the specification will be mapped into a corresponding (possibly nested) transaction structure which not only reflects the database manipulation but also enforces the specified constraints.

3.3 Manipulation

The manipulation (do) section of the ODL specification is an important component of the behavior description. The main purpose of providing such a construct in ODL is that the method can be decomposed into lower-level operations, i.e., the operations defined for the lower-level classes in the class composition hierarchy. The lower-level operations are also defined by the application programmer. The relationship between the method and its lower-level actions are specified more clearly. Moreover, many methods of complex objects are themselves complex operations consisting of methods defined for their constituent objects. The method composition is usually a sequential invocation of other methods or lower-level procedures.

3.3.1 Format

In order to capture the relationship between a method of a complex object and that of its components, an action statement in ODL has the following format:

```
<action> := [ <path> ] [.|->] <method> ( [<arguments>] )
          | FOREACH ( <component>, ..., <component> )
                    <path> [.|->] <method> ( [<arguments>] )
```

The two different formats of the action statement can be applied to different situations of method descriptions. Any method may be invoked by using the first format. The <path> starts from the current class in the class composition hierarchy. It may be omitted if the method is simply another member-function of the current class. The <path> also enables the user to invoke methods defined a number of levels down the

class composition hierarchy.

When different paths, methods and arguments are involved in the actions, the user must invoke them explicitly one after another. However, it is often the case that the same method is defined for several components of the complex object with the same interface (signature). More often, some of the components of a complex object are actually from same class. When such methods are to be invoked for several components in a complex object, the second format can be used to group the components together where the method and arguments can be specified only once.

3.3.2 FOREACH statement

The `FOREACH` statement specifies a group of component objects on which the method is to be performed. The `<path>` in this format represents a path in the class composition hierarchy. It helps the system to find the definition and implementation corresponding to a method. The components within the parentheses must be the constituent objects from the current class. In the code generated, the method `<method>` will be invoked for each of the components.

For example, consider a complex object from class `cuboid`:

```
class Cuboid{
      VERTEX      block_center;
      FACE        top, bottom, left, right, front, back;
      ...
}
```

A method **move** applied on objects in class **cuboid** not only modifies the **block_center** of the cuboid but also moves the six component objects from class **Face**. The method can be represented as follows (**this** is a keyword representing a pointer referencing the current object):

```
void Cuboid::move(float dx, float dy, float dz)
        {
        do:
            this->block_center->move(dx, dy, dz);
            foreach (top, bottom, left, right, front, back):
                    Face->move(dx, dy, dz);
        with:
            /* constraints */
        }
```

In this example, six faces are grouped into the **foreach** statement. A method **move** is also defined in class **Face** and its implementation is different from the one being specified. By using the above specification, we are able to describe the relationship between methods defined on two levels. A **move** is also defined in class **Vertex**, it changes the value of a point object. It is invoked as a method performed on the **block_center** attribute of the current object.

3.4 Behavioral Constraints

In ODL, an important aspect of behavioral description is the specification of behavioral constraints associated with the methods. A method can be seen as signaling an

event that occurs when the method is invoked. The constraints, on the other hand, are actions associated with conditions that can be checked either before or after the event occurs.

Some of the approaches to constraint specification in the literature have combined operations, e.g., disjunction, sequencing and closure, defined as events [22]. However, those combinations of operations are arguably more related to user semantics and as such should be captured by the applications. In fact, since our concept of event is defined as a method of a class, combinations of finer events fit easily into our model. The `do` section of ODL also supports high level specifications of complex events.

3.4.1 General Format

There could be more than one condition associated with one method. The multiple conditions enable the users to set up control over the behavioral constraints. Different constraint actions may be specified for different conditions. Hence, the conditions specified for different constraint action sets are usually disjunct. When two conditions overlap, i.e., when they are both evaluated to be true, the constraint actions will be performed according to the order they are defined. The general format of the behavioral constraint following the keyword `with` is:

```
WHERE [[PRE | POST] <condition> | TRUE]: [[PRE | POST]]
      <action>;
      <action>;
      ...
      <action>;
```

The PRE and the POST appearing before <condition> indicate the time when the condition is to be checked. If neither PRE nor POST is specified, the condition will be checked after the event occurs. The PRE and the POST appeared before the actions determine the orders of invocations of the method and the constraint actions. If neither PRE nor POST is specified, the actions will be performed after the event occurs. It is obvious that a combination of a POST condition and a PRE action is not semantically valid and must be avoid in the specification. The actions in the behavioral constraint are actually function calls that invoke member-functions defined in various classes. The format of an action is as follows:

```
[FOREACH <object> IN <class> THAT [PRE | POST] <condition>:
 { methods }
| [TOP] REJECTION
| methods
```

The optional FOREACH clause plays a role of "filter" and returns a collection of objects. It selects objects from a particular class and examines them against the <condition>. For those objects that are "passed", the corresponding methods are performed. Under some conditions, a REJECTION may be specified as a single action. This action will simply deny the execution of the method because the execution of the method may possibly drive the database into an unrepairable invalid state.

3.4.2 PRE and POST conditions

Traditional constraint checking is usually performed after an event occurs or at the end of a transaction. However, in behavioral constraints, the conditions may not

necessarily be specified over the database states after the event has occurred. In fact, many behavioral constraints require that state of the database should be checked before the event occurs. Moreover, it is also possible that the value of a condition has changed after the event has occurred (due to modifications made within the method execution). For example, consider the following behavioral constraint on the method `move` of the object `cuboid` in a situation where two cuboids are glued together:

> *"If a cuboid is on_top_of a block x, then when the cuboid is moved to another place, move x as well."*

The condition `on_top_of` should be checked before y is moved. If the condition is checked after the `move` operation, the truth value of the condition may not remain the same. In this constraint, x is an object and is referenced by its object identifier. In ODL, the operation of `move` and its constraints can be defined as:

```
void Cuboid::move(float dx, float dy, float dz)
            {
            do:
                this->block_center->move(dx, dy, dz);
                foreach (top, bottom, left, right, front, back):
                        Face->move(dx, dy, dz);
            with:
                where pre on_top_of(this, x):
                {x.move(dx, dy, dz)}
            }
```

The <conditions> field in the **where** clause is basically an extended boolean expression which returns TRUE or FALSE. The format of the <conditions> is as follows:

```
[ conditions AND conditions
| conditions OR conditions
| ( conditions )
| condition
| THEREIS <object> IN <collection> THAT conditions ]
```

The extended boolean expression, i.e., the THEREIS statement, makes it possible to check the boolean condition against the class extent. For example, as mentioned, on different application environment there could be different semantics for the method **move** of the class Cuboid. For example, if a user wants to define the constraint that no cuboid will be moved when there is anything on the top of it, i.e.:

"If there is anything on the top of the cuboid, do not move it".

This constraint can be represented using ODL as follows:

```
void Cuboid::move(float dx, float dy, float dz)
        {
        do:
            this->block_center->move(dx, dy, dz);
            foreach (top, bottom, left, right, front, back):
                    Face->move(dx, dy, dz);
        with:
```

```
        where pre thereis x in block that on_top_of(x, this):
        {rejection};
    }
```

Note that the condition is checked before the event `move` occurs. The keyword `thereis` requires that objects in class `block` be checked until one is found that satisfies the condition `on_top_of(x, this)`.

3.4.3 Constraint Actions

The constraint actions can either be function calls or rejections. There is a `FOREACH` clause that will select appropriate objects in a particular class. For example, consider the following constraint:

> *"When a cuboid is moved, all the blocks that are on top of it must also be moved."*

Before an object from class `Cuboid` is moved, all the objects from a super class `block` are selected and checked to see whether they are on the top of the one to be moved. Obviously, the condition should be specified as `PRE`. Using ODL, this constraint can be represented as follows:

```
void Cuboid::move(float dx, float dy, float dz)
    {
    do:
        this->block_center->move(dx, dy, dz);
        foreach (top, bottom, left, right, front, back)
```

```
                Face->move(dx, dy, dz);
    with:
        where true:
        {
        foreach x in block that pre on_top_of(x, this):
        {x.move(dx, dy, dz)}
        }
```

In this constraint, all of the actions of moving each block that is on top of the cuboid will be put into a single transaction together with the **move** operation. Note the **pre** specifies the condition that is to be checked before the **move** is actually performed.

REJECTION can be specified to express the rejection of a method when some conditions are satisfied. **REJECTION** is different from the usual *"if (condition) then no-op"* construct of programming languages in that it has richer semantics. When a **REJECTION** is specified, the calling transaction must be aborted, the operation performed by the method is thus rolled back. There is also a keyword **TOP** which, if specified, will cause the top level transaction to abort when the condition for **REJECTION** is satisfied. For example, consider the complex objects **person** defined as:

```
class person{
      STRING      name;
      INT         age;
      ...
      *person     spouse;
      ...
}
```

The following constraint:

> *"A person cannot marry with person p, if either of them currently has a spouse."*

expresses that no person is allowed to marry two people simultaneously. This constraint can be expressed in ODL as follows:

```
void person::marry(person p)
        {
        do:
            this.spouse_update(p);
        with:
            where pre ((p.spouse != NULL) || (this.spouse != NULL)):
            {rejection}
        }
```

The spouse_update is a member-function of class person that records the update of the spouse attribute. The REJECTION in this constraint is not specified as TOP because aborting the current transaction should not affect the upper level transactions. We will give an example later in this chapter where the state of upper level transactions may be affected by aborting an inner level transaction.

3.5 More Examples

In previous sections, we proposed our design of the object definition language ODL. Here we present some more examples to show that the language is capable of expressing various complex object semantics for different OODB applications.

3.5.1 Mutual Dependency

As we have seen, the identification of behavioral constraints allows for users to specify database constraints more clearly. Moreover, it also eases the problem of constraint checking. The following example extracted from [13] shows a problem of constraint checking and how it can be handled with the ODL specification.

Example

Suppose we have the following class definition:

```
class person
{
private:
    ...
    persistent person *spouse;
public:
    ...
}
...
```

```
person p1, p2;
```

A constraint specifies that

> *"if a person has a spouse, then the spouse's spouse must be the person himself or herself"*

can be expressed in a traditional boolean expression as follows:

```
(spouse == NULL) || (this == spouse->spouse);
```

Initially, we suppose that the persons p_1 and p_2 were not married. Now we consider the transaction to record the fact that p_1 and p_2 have married each other. Because the `spouse` attribute is a private member in class `person`, the operation of modifying the `spouse` for a `person` is thus associated with the class itself. The moment p_2 is made the spouse of p_1, the above constraint will be violated because the spouse field of p_2 has not been updated. The reverse problem occurs if p_1 is first made the spouse of p_2. In either case, the transaction will be aborted. In fact, whenever a pair of complementary relationships has to be maintained between two objects, the same problem will occur.

[13] suggests a deferred or transaction-level constraint checking mechanism in order to solve the above problem. However, while a deferred constraint checking mechanism will be expensive to implement, it is much easier if we could extend the scope of the transaction to include behavioral constraints. This can be achieved easily by describing the behavior of `marry` in ODL. After the specification is accepted and processed by the system, the generated transaction will have a constraint maintenance part.

Instead of being performed always at the end of the transaction, the checking will be performed either before or after the method is invoked according to the specification. Constraint maintenance actions will be performed if violations are detected.

Suppose the operation of recording the marriage for a person is defined as a method `marry(person)` in class `person`. Expressed in ODL, the behavior of `marry` will be:

```
void person::marry(person p)
        {
        do:
            this.spouse_update(p);
        with:
            ...
            where post (p.spouse != this):
                  {p.marry(this)};
            ...
        }
```

Only one method is invoked in the "do" section. The constraint will be checked after the attribute `spouse` is updated. If the `spouse` of the spouse is not the same as `this`, then a repairing action `p.marry(this)` is performed. The checking and the repairing action will both be included in the transaction of `marry`. By using the ODL specification and the corresponding code generation mechanism, the problem of "complementary aborting" encountered when using traditional constraint specification is avoided. This is because temporary violation of consistency is allowed with our approach and the user-specified repairing actions can be performed by the system when such violation occurs.

It can be observed that a "recursive checking" problem will occur if the condition (`p.spouse != this`) in the **where** clause is specified as a **pre** condition. Suppose p_1 is to marry p_2, if the condition is specified as **pre**, then p_2`.spouse !=` p_1 will be checked before the **spouse** attribute is updated and it will return true. Since the repairing action involves recursive call of **marry**, the condition p_1`.spouse !=` p_2 will also be checked before any update is performed. This condition will also return true. A second repairing action will be performed and a recursive checking is incurred. The essence of this problem is that the repairing actions must not be performed before the violation happens.

By providing such a counter example, we point out that even though ODL is capable of expressing complicated semantics of behavioral constraints, the users must still be careful about the specification.

3.5.2 Recursive Checking

The example in the above sub-section shows that some of the constraints with behavioral properties are very difficult and expensive to maintain when expressed in traditional constraint specifications. However, they can be easily specified and maintained when expressed as behavioral constraints in ODL.

Another example extracted also from [13] shows that the behavioral aspects of complex objects can be captured more easily in ODL. The maintenance of constraints can also be more natural and powerful. In CAD applications, constraints often involve other objects such as neighbors. Consider a row of adjacent cells on a chip that are placed next to each other. Except for the end cells, each cell has two neighbors. The

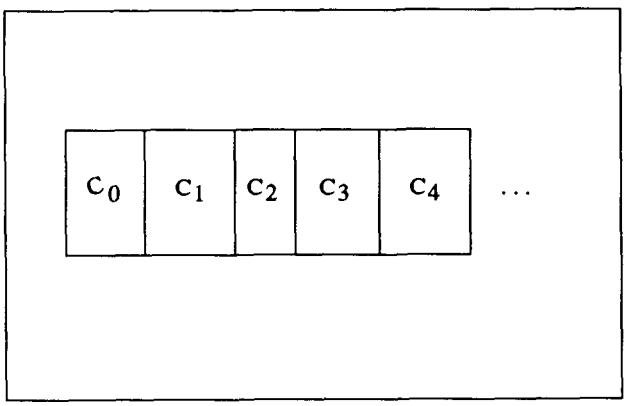

Figure 3.1: The Domino Effect

cells can be represented as complex objects that indirectly reference the neighboring objects (Figure 3.1):

```
class cell {
      int       x, y;     //center coordinates
      int       w, h;     //width height
      cell      *left;    //left neighbor
      cell      *right;   //right neighbor
      cell      cell(int x1, int y1, int width, int height);
      cell      shift(dx);
      }
```

A cell must always satisfy the following constraints:

1. *It must be on the chip.*
2. *It should be adjacent to but must not overlap its left neighbor(if any).*
3. *It should be adjacent to but must not overlap its right neighbor(if any).*

These conditions must be satisfied when a cell is created and when a cell is moved. Using the specification proposed in [13], the constraints are expressed in the **constraint** section of the class definition:

```
class cell {
        ...
    constraint:
            x-w/2 >= XMIN && x+w/2 < XMAX;

            (right==NULL)||x+(w+right->w)/2==right->x:
                  right->shift((w+right->w)/2-(right->x-x));

            (left==NULL)||x+(w+left->w)/2==left->x:
                  left->shift((w+left->w)/2-(left->x-x));
    }
```

Because the constraints 2 and 3 involve behavioral properties, it is very difficult to specify them in traditional constraint formula. The above specifications are inevitably complicated and not very clear. However, if we introduce a predicate **adjacent** defined as:

```
boolean adjacent(cell a, cell b){
        if (a == NULL) return(FALSE);
        if (b == NULL) return(FALSE);
        if ((a.x + a.w/2 = b.x - b.w/2) ||
           (a.x - a.w/2 = b.x + b.w/2))
```

```
        return(TRUE)
        else return(FALSE);
    }
```

and a method update_center as updating the coordinates of the center for a cell, by using the specification of ODL, the constraints can be easily expressed as follows (note here the constraint is associated with the method shift instead of the whole class):

```
void cell::shift(int dx)
    {
    do:
        update_center(this.x + dx);
    with:
        where post ((this.x + this.w / 2 > XMAX)
                or  (this.x - this.w / 2 < XMIN))
        : {top rejection};

        where pre  adjacent(this, this.right)
        : pre {this.right.shift(dx)};

        where pre  adjacent(this, this.left)
        : pre {this.left.shift(dx)};
    }
```

In this example, a constraint violation domino effect occurs from the fact that moving

a cell violates the constraints and the repairing action also involves moving another cell. When a cell is moved, in order to ensure that its constraints are satisfied, the related neighbors of the cell must also be moved. This will in turn violate the neighbor's constraints.

One problem with the constraint specification proposed in [13] is that if any cell is moved outside the chip (x-coordinate of left-end is less than `XMIN` or x-coordinate of right-end is greater than `XMAX`) during the repairing action, the resulting constraint violation cannot be repaired. This is because the updates of previous operations have already been recorded into the database and the corresponding transactions have been committed. The aborting of the current transaction will not be able to repair the damages made to the database.

However, this problem can be eliminated by using our approach. Since our behavioral constraint enforcement mechanism associates the constraints with appropriate transaction structures, each event `shift` and the corresponding repairing actions will be put into the same transaction. The generated code will be similar to the following (`do_transaction(){}` is the transaction specification in the DML of ObjectStore):

```
do_transaction(){
        if        adjacent(this, this.right)
        then      this.right.shift(dx)
        else if   adjacent(this, this.left)
             then this.left.shift(dx);

        update_center(this.x + dx);
```

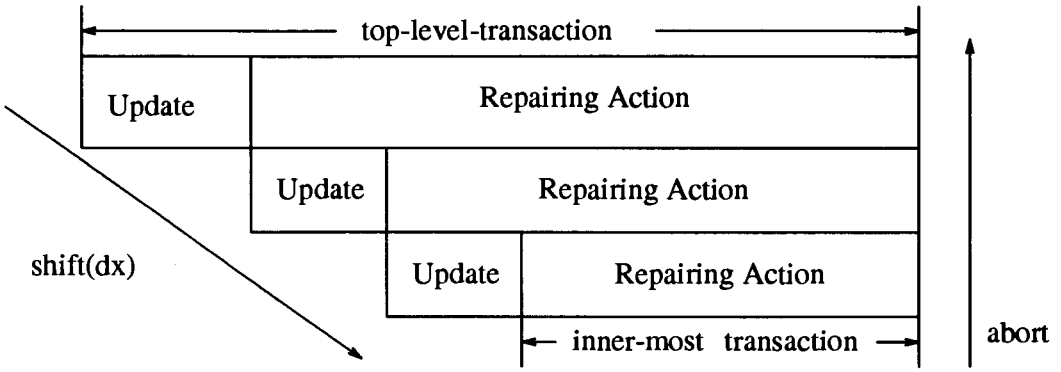

Figure 3.2: The Generated Transaction Structure

```
        if    ((this.x + this.w / 2 > XMAX) ||
               (this.x - this.w / 2 < XMIN))
        then  {
              os_transaction::abort_top_level();
              /* abort the top level transaction */
              }
}
```

The above listing shows the code when the system procedures defined in ObjectStore are used. `os_transaction::abort_top_level()` is an ObjectStore system procedure which aborts the top level transaction [23]. When using other databases, the corresponding system routines can be used in the places of aborting actions.

When the method `shift` is invoked in execution for a cell with several neighboring cells, a nested transaction structure will be built up because of the domino effect. Each level of the nested transaction corresponds to the operation on one particular cell. If any cell is moved outside of the chip, the top level transaction will be aborted because `top` is specified in the constraint. Therefore the whole transaction will be aborted

eventually and the constraints can be maintained, i.e., when any cell is moved outside the chip in a repairing action, the very first invocation of the method **shift** will be rejected.

This example shows that in our approach the rule propagation will only occur in a controlled manner. All the rule propagations are pre-defined. They will be generated as appropriate transactions of updating the repairing (Figure 3.2).

In this chapter, we have described the syntax of our ODL. We have also show that ODL can be used to defined objects in various applications. However, we do not define the formal semantics of the ODL since the semantics of our ODL is dependent on the semantics of user defined operations and a formal discussion of the semantics of the ODL will be out of the scope of this thesis. Because the ODL is intended to be a high level language for database designer and application programmers, its syntax and informal semantics are clear enough.

Chapter 4

Implementation

In the previous chapters, we have described the language ODL for expressing the object definition, especially the behavioral aspects including the behavioral constraints of objects. In this chapter, we discuss the implementation issues for supporting ODL on top of an object-oriented database, namely, ObjectStore. We want to show that ODL can be used for specifying the object definitions, especially the behavior of objects and that the behavioral constraints expressed in ODL can be easily enforced.

We have implemented a parser for ODL using YACC (Yet Another Compiler Compiler) and LEX. The code generator generates C++ code with appropriate ObjectStore functions. Since we did not provide a full computing capability with ODL, the methods that involve various computation should be defined separately from those involving complex semantics. The generated code will then be integrated with the user's class definition and be compiled with the ObjectStore compiler.

The system module supporting ODL specifications is implemented on the application programming level. The ultimate goal of the system is to guarantee that the manipulation of the database be handled in accordance with the user semantics. One of the reasons that we pursue such a strategy of implementation is that we believe the programming interface between an application and the database management system is powerful enough to support various operations and the associated integrity constraints. In fact, by embedding appropriate transaction structures to the application programs, the integrity of the database can be maintained, the behaviors of complex objects can also be properly supported.

4.1 ObjectStore

ObjectStore is a commercialized OODB system [23, 24]. It is, in essence, a database programming language. ObjectStore combines the data query and management capabilities of traditional databases with the flexibility and power of the C++ object-oriented programming language. It supports persistence orthogonal to type, transaction management, and associative queries.

One of the ObjectStore's most important features is that the persistence is specified on a per-instance basis and is independent of the types. Each type may have both persistent and non-persistent instances. The member-functions can operate on both persistent or non-persistent data. This feature has made it very easy to integrate application programs with the underlying database services. Complex objects that are transient in the application programming space can be defined as object-oriented views that correspond to some persistent objects in the database.

ObjectStore supports nested transactions. There are two kinds of transaction aborts in ObjectStore. When a system procedure

```
os_transaction::abort()
```

is called with no arguments, the innermost transaction is aborted. The outermost transaction may also be aborted if the system procedure

```
os_transaction::abort_top_level()
```

is called from within a subtransaction.

Another kind of abort is performed automatically by the system. The system-performed abort can occur in two circumstances. If a network failure affecting the current process occurs, the system usually aborts the current transaction of the process. All the sub-transactions nested within that current transaction are also aborted. Another situation where a transaction can also be aborted by the system is deadlock. ObjectStore is able to detect deadlocks and in the case of deadlock, one process involved will be picked to be aborted.

4.2 Outline of the System

We have implemented a system module that supports ODL on the top of ObjectStore. The module consists of an ODL parser and a code generator. A grammar of ODL is written and provided to YACC and LEX. The parser is generated by YACC and LEX based on the ODL grammar. When accepting the object definitions written ODL,

the parser will build a tree representation of the method specified. The code generator will then traverse through the tree representation and generate corresponding code in C++ and ObjectStore DML. The generated code corresponds to the method implementation (Figure 4.1).

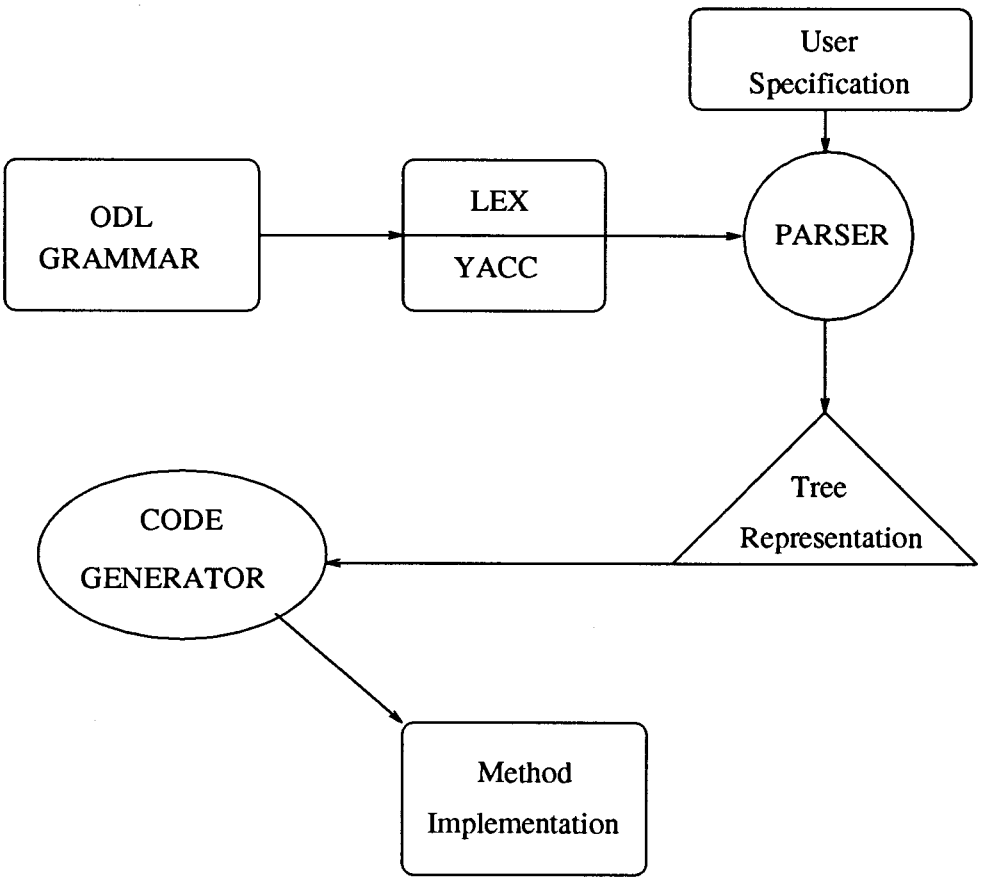

Figure 4.1: The Architecture of ODL Compiler

The implementation of the method can thus be integrated with other user definition and implementations to form a complete application. In order to present a uniform interface for the application programmer, the method implementation may be either put into a static library or put into a *Dynamic Linking Library* (DLL).

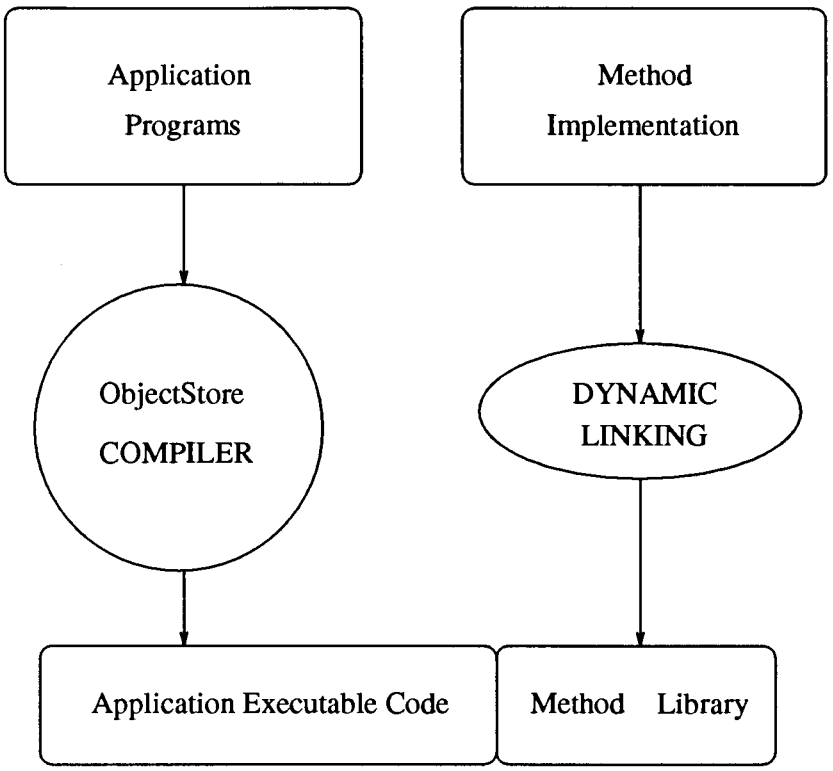

Figure 4.2: Implementation Integration

As long as there is a consistent interface in the application program for the method (including the declaration), any modification on the ODL specification will not affect other parts of the application program. Therefore, any changes made to the semantics of the methods will be transparent to the application programmer. The application code will be simply compiled by ObjectStore compiler. The integration of the code is realized dynamically (Figure 4.2). This means the system module of the ODL can maintain the object definitions by itself for the underlying OODB.

The ODL specification captures all function-calls and behavioral constraints explicitly. When using ODL to describe the behaviors for objects, the users need to provide definitions for lower-level operations.

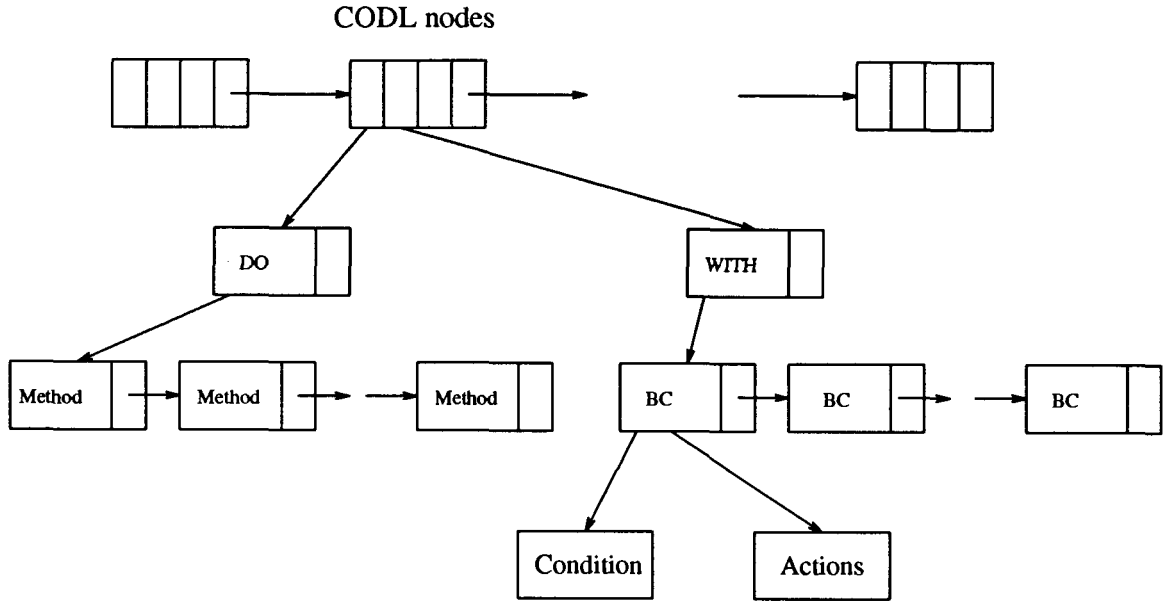

Figure 4.3: The Structure of ODL Tree

One of the important functionalities of the ODL module is to enforce behavioral constraints. The central part of an enforcement mechanism for behavioral constraints is the detection of the *event*. In our system, an event can be thought of as the operations specified in the "do section". After an event has been properly detected, the behavioral constraints associated with the event will easily be identified and enforced by the execution of the repairing actions.

4.3 Code Generation

The ODL parser generates a tree representation from the user specification. The code generator will then generate code based on the tree. The structure of the tree is showed in Figure 4.3. For example, in chapter 3, we gave an ODL specification for

the method `shift` of the class `cell`:

```
void cell::shift(int dx)
      {
      do:
          update_center(this.x + dx);
      with:
          where post ((this.x + this.w / 2 > XMAX)
                  or  (this.x - this.w / 2 < XMIN))
          : {top rejection};

          where pre  adjacent(this, this.right)
          : pre {this.right.shift(dx)};

          where pre  adjacent(this, this.left)
          : pre {this.left.shift(dx)};
      }
```

the code generator will generate the following code when traversing the tree generated by the ODL parser:

```
cell::shift(int dx)
      {
      do_transaction()
      {
      if adjacent(this, this.right)
```

```
        this.right.shift(dx);
if adjacent(this, this.left)
        this.left.shift(dx);
update_center(this.x + dx);
if ((this.x + this.w / 2 > XMAX) ||
    (this.x - this.w / 2 < XMIN))
        {
        os_transaction::abort();
        os_transaction::abort_top_level();
        }
}/* transaction */
}
```

The conversions from ODL to the generated code are mostly quite straightforward. The functions that are invoked in the *do* section in ODL can be found in the system. They can be either generated from another ODL specification or defined directly by the user. Many low-level database manipulations are not included in ODL code so that it can be easily ported to other database systems. However, the users still need to implement the low-level operations themselves. These operations can thus be invoked by the ODL generated code.

In the code generated from ODL specifications, both the operations for manipulating the database and the code for integrity maintenance are included in the same transaction. This is consistent with the semantics of the ODL specification. Since the database operations (such as `update_center(this.x + dx)` in the above example) may violate the consistency of the database, the repairing actions should be attached

to them.

4.3.1 Condition Checking

In code generation, the code for enforcing behavioral constraints has to be combined with the code for the "do section". One of the tasks for generating constraint code is to put the constraint checks in the proper places. Since the user may define a condition as `PRE` or `POST`, the condition checking may either appear before the "do section" or after.

For those conditions that need to be checked before a "do section", the actions may be performed after the "do section". We introduce some new variables to store the results of the checking. These variables can later be used conveniently for determining the corresponding enforcement actions. In ODL, there is a `THEREIS` statement that specifies the existence of objects satisfying some conditions. In order to perform the checking correctly, we need to check every single object of the collection against the condition specified. The code generated thus contains a `foreach` statement of the ObjectStore DML which performs an iteration of checking. For example, the code generated for the following statement in ODL:

```
where pre thereis x in block that on_top_of(x, this):
pre {rejection}
```

is as follows:

```
foreach(x, block)
    {
```

```
            if on_top_of(x,this)
                    _C2:=TRUE;
        }
    if (_C2 == 1)
        {
          os_transaction::aborts();
        }
```

where _C2 is a newly introduced boolean variable. These two statements will be put before the "do section" because of the pre specified in the where clause.

4.3.2 FOREACH Statement

A constraint enforcing action in ODL can be a function-call, a rejection or a foreach statement. When it is a foreach statement, a collection from which objects will be selected and a condition which sets up the criterion for object selection must also be specified. In code generation, the foreach statement is also realized using the ObjectStore foreach statement. As an example, a foreach statement in ODL stating that a move operation be performed for all objects from block that satisfy an on_top_of condition is as follows:

```
where true:
        {
          foreach x in block that pre (on_top_of(x, this)):
          pre {x.move(dx, dy, dz)}
        }
```

will be translated into the following code:

```
foreach(x,block){
     if (on_top_of(x,this))
              {
                x.move(dx, dy, dz);
              }
     }/* foreach */
```

4.4 Application Integration: An Example

As an example of applying ODL in applications of an object-oriented database, we have integrated ODL with a customized object-oriented query language known as *Dynamic Spatial Query Language* (DSQL) [33].

4.4.1 DSQL

There have been efforts of developing customized query languages for various application areas of object-oriented databases to complement the limited query capabilities of some database programming languages. DSQL represents one such effort. DSQL is a customized query language for manipulating geometric objects in three dimensional (3-D) space. By using DSQL, the database user may perform retrieval, update and deletion on any spatial or non-spatial attributes of the objects stored in the database.

One of the sample data domains of DSQL is called *Block World*, which consists of a collection of blocks (e.g. *spheres, cylinders*) in 3-D space. Various operations may

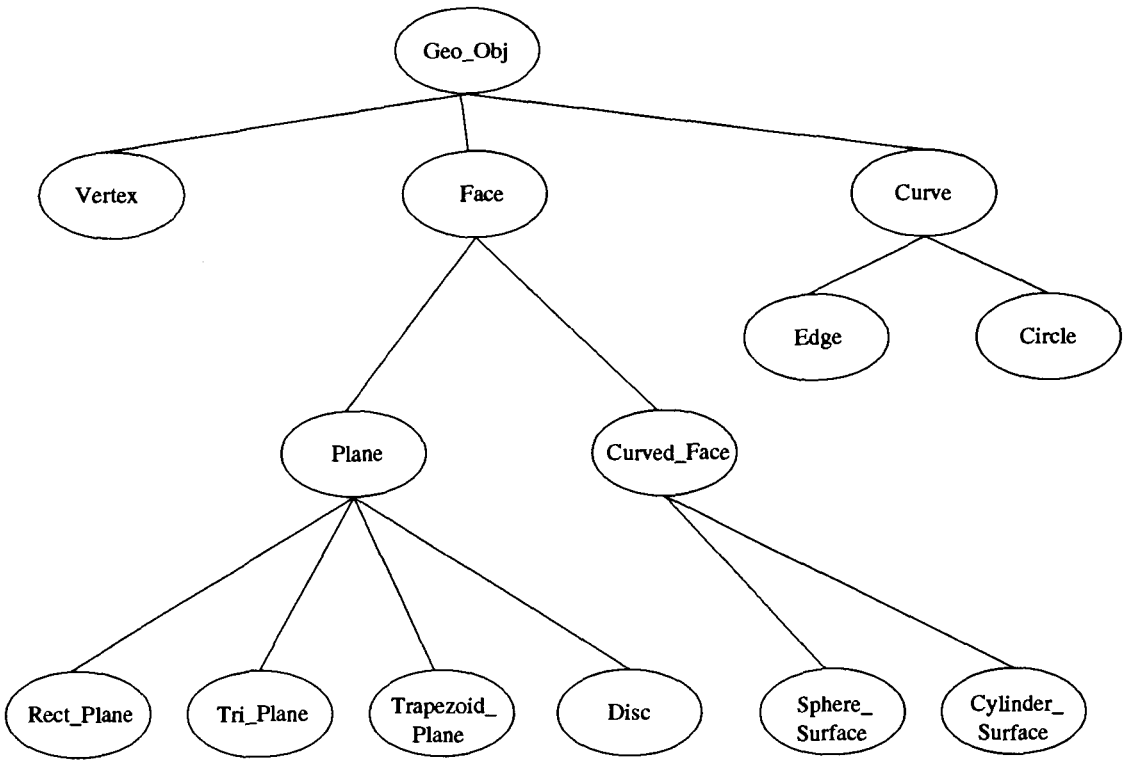

Figure 4.4: The DSQL Geometric Schema

be applied to the blocks (e.g. *move, rotate*). The schema used for the Block World is a bi-level schema which is composed of a geometric object schema and a block object schema.

Objects in the geometric schema are grouped into various classes with a class *Geo-Obj* as their superclass. Examples of such objects are *vertices, edges*, and various *faces*. The classes of geometric objects are grouped into a class hierarchy rooted at *Geo-Obj* (Figure 4.4).

Objects in the block schema are grouped into various classes with a class *Block* as their superclass. Examples of such objects are *sphere, cuboid, frustum*. These classes of block objects are also grouped into a class hierarchy. The class hierarchy is rooted

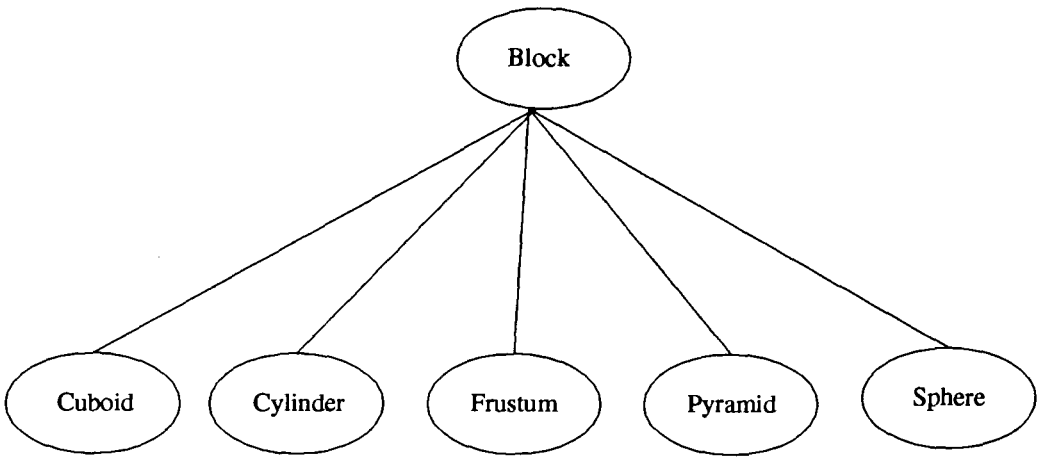

Figure 4.5: The DSQL Block World Schema

at Block (Figure 4.5).

In this bi-level schema, the *Geo-Obj* is defined as the lower-level schema and the *Block* is defined as the upper-level schema. All the objects in the upper-level schema are constructed with objects in the lower-level schema. For instance, a cuboid is constructed with six faces.

DSQL is an English-like, application-specific query language associated with spatial and non-spatial predicates. DSQL is also similar to SQL in that a number of high level operations such as *insert* and *select* are provided for database manipulation. However, unlike the SQL, which only provides generic query operators, DSQL also provides customized query operators such as *move* and *rotate*.

4.4.2 Code Integration

Because the DSQL is previously implemented [32] on top of an object-oriented database system, namely, ObjectStore, it can be seen as an application built on ObjectStore.

For the schema of the block world, various user semantics can be associated with the operations on block objects. Behavioral constraints can also be defined for those operations. The block world schema can be seen as a simplified MCAD schema. The importance of describing object behaviors in such an environment is to show that our approach can be applied in advanced database applications.

In order to combine ODL with the implementation of DSQL, we must specify in ODL the methods involved in the DSQL queries. We can then generate the code for those methods. Since the DSQL query processing involves invocations of lower-level methods, the user semantics specified in ODL will be enforced when the generated code is integrated with DSQL implementation.

We have implemented an example by providing ODL descriptions for *move* and *rotate*, two of the major methods defined for block objects as well as geographical objects. In particular, we have defined in ODL the method `move` for class `Cuboid`. In order to fully develop the `move` operation for `Cuboid`, we also provide the descriptions for the corresponding `move` methods for the components of the cuboid. A cuboid is composed of six objects from class *Rect_Plane*, which is a subclass of class *Plane*. Class *Plane* is a subclass of the class *Face*. Each *Rect_Plane* in turn consists of four objects from class *Edge* which is a subclass of class *Curve*. We have defined the virtual method function `move` for classes *Face*, *Plane* as well as *Curve*. For classes Cuboid, *Rect_Plane* and *Edge*, we have defined the method `move` in ODL. The ODL generator then generates the implementation code for those member functions.

Not only have we implemented the method `move` for these classes by providing the specifications in ODL and integrating the generated code with other parts of the

system, we have also added the code for enforcing behavioral constraints by specifying the behavioral constraints in the ODL specifications. Figure 4.6 and Figure 4.7 illustrate how the behavioral constraints are enforced by the generated code.

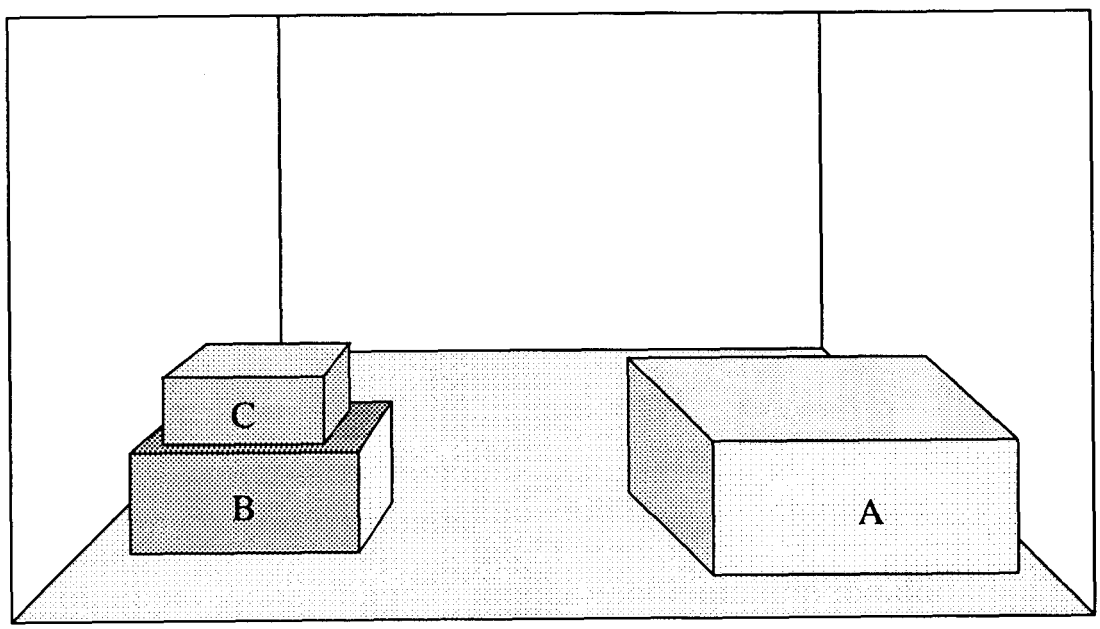

Figure 4.6: The Database State Before *move*

In a database, there are three cuboid objects, A, B and C. Figure 4.6 shows the database state before the operation *move* is performed. One of the constraints is:

> *"When an object is to be moved, any object on top of it must also be moved altogether."*

Figure 4.7 shows the database state after the *move* operation is performed on object B. The behavioral constraint is enforced since object C has also been moved with the object B.

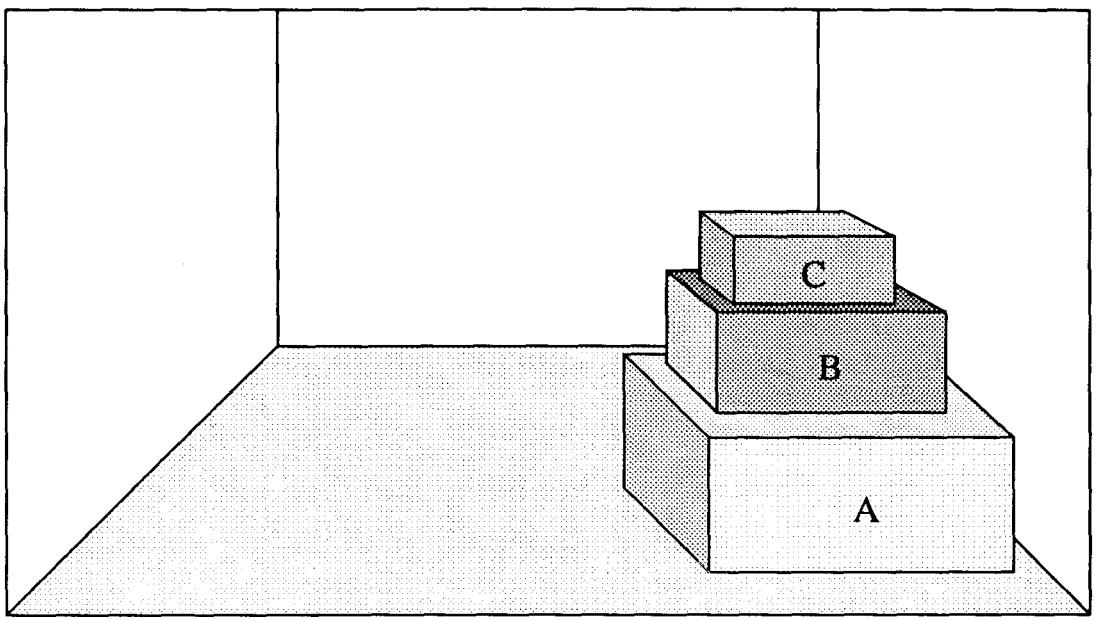

Figure 4.7: The Database State After *move*

4.4.3 Comparison of Implementations

In the original implementation of the Block World [32], a bi-level block world schema was defined. A customized query language DSQL was also designed and implemented. The system can accept DSQL queries and perform corresponding operations on the database. There is also an X/Windows display mechanism that can be used to check the current database state. Although the implementation generated by the ODL generator performs similar database manipulation, it is different from the original implementation in several aspects.

Firstly, the semantics of the move operation are implemented more clearly. This can be seen by examining the ODL specification. The `move` operation defined for the `Cuboid` simply consists of the moving of the `block_center` and six faces. The `move` operation of each face in turn consists the moving of the face center and the four

edges. The moving of an edge is simply the moving of the center.

In the original implementation, the schema was designed to reflect the bi-level data modeling. For example, although a cuboid has six faces as its component objects, the faces are actually sharing the edges as their components. So in the original schema, a cuboid only has 12 edges. In our new implementation, however, we have constructed the data model differently. In a cuboid, even though the edges of two adjacent faces may have same value, we still created two different objects to represent them. So in our schema, a cuboid has 24 edges. By using such a schema, we are able to divide a complex object more easily into its component objects. This model also reflects the concept of the object identifier, i.e., even though two objects may have the same value, they are still different objects if their OIDs are different.

Secondly, by specifying the behavior of `move` in ODL, we are able to define the behavioral constraints associated with it. The code generated by the ODL generator enforces the constraints automatically. In original implementation, there is no code for enforcing behavioral constraints. In fact, although behavioral constraints can be implemented by programmers, a manual implementation of complex behaviors has drawbacks compared to an ODL implementation. It lacks a formal specification of the behavioral constraints and is thus difficult to verify. It is also difficult for the programmer to include the code for enforcement if the operations are not defined strictly according to its user semantics.

For instance, in the original implementation of DSQL, the display function is invoked from within the `move` to achieve the effect of displaying the database state after a block has been moved. However, the action of displaying the database state is not part of the `move`. Instead of putting the display in the move, a more clear

implementation should display the database state after each query has been processed.

In summary, since we have provided ODL as a uniform specification for complex object behaviors, the implementation generated by the ODL generator is highly consistent with the user semantics. The code for integrity maintenance is also generated and combined with the methods automatically. By putting the methods generated from ODL into a dynamic linking library, the other parts of the application program do not need to be recompiled. Any changes made to the semantics of the behaviors can be easily added into the application code. The object definition written in ODL can also be solely maintained by the ODL module using the dynamic linking library.

Chapter 5

Conclusion and Future Work

5.1 Thesis Summary

In this thesis, we examine the behavior aspects of object definitions, especially complex objects in various applications of OODB. In particular, we examine the phenomena of behavioral constraints associated with object methods.

In order to provide the database user with a more convenient and efficient way of supporting various complicated application specific semantics, we have designed a language ODL for object definition, especially for describing object methods and the associated behavioral constraints.

In ODL, an object method is defined in terms of function calls which invoke other functions, especially methods defined for the constituent objects. A behavioral constraint is defined as a number of conditions and actions associated with a method. The execution of the method can be seen as an event. The checking of the conditions

can be performed either before or after the event happens. The actions may either be performed on a single object or on a set of objects that satisfy certain conditions.

Compared to previous approaches for constraint specification and enforcement, our behavioral constraint specification is more powerful since the user-defined methods are regarded as events. The user can put various database manipulations or event signals into their methods. The condition checking of our approach is more flexible. The user may also change the boundary of the methods to adjust the points of checking. Because the maintenance of our constraint rules do not need a separate rule engine, the enforcement is thus more efficient than some of the previous approaches. By associating the method definitions with appropriate (possibly nested) transaction structures in the code, we are able to solve some problems encountered by some previous approaches of constraint maintenance.

We have implemented a parser and a code generator for ODL. The parser accepts specifications written in ODL and generates an internal data structure to store the information. The code generator is designed to generate application code which uses ObjectStore as the underlying database management system.

As an example, we have implemented methods for objects in a database called Block World using an ODL specification. In the method implementation generated by the ODL code generator, the enforcement of the behavioral constraints has been successfully integrated with other parts of the system. The implementation employs an interface between the class definition and the dynamically linked implementation of methods.

5.2 Contributions

In this thesis, we present the design and the implementation of an object definition language, which can be used to specify objects, especially the behavioral aspects including constraints of objects, for advanced applications of OODB systems. The major contribution of this thesis can be described as follows.

In current OODB systems, especially those DPLs, the only interface between the user and the DBMS is the host language (Persistent C++ or Smalltalk). There is no high level interface for the users to define their databases. Our ODL provides a high level language for object definition.

We have investigated a particular category of database constraints called *behavioral constraints*, i.e., the constraints associated with database manipulation. Moreover, in an object-oriented database environment, the constraints can be associated with the user-defined object methods, especially methods of complex objects. By using our ODL, behavioral constraints can be specified more clearly.

We have combined the concepts of the ECA model and the constraint compilation technique in Ode into the design and the implementation of our ODL. In our approach, methods are identified as events. We have also designed language constructs to allow the user to assert their condition checking either before or after the event occurs. The condition checking can be performed either on single objects or on collections of objects.

We have implemented a prototype of the ODL. The ODL code generator generates C++ code associated with ObjectStore DML. By executing the ODL code for the methods, the constraints associated with the methods can be enforced automatically.

In order to show the usage of our ODL, we have used ODL as the definition language for defining objects in an application called Block World. The ODL code for the methods are generated and integrated with a customized query language DSQL. We also show that the implementation with ODL definition is clearer than the original implementation.

5.3 Future Work

In this thesis, we tried to include the behavioral aspects especially behavioral constraints in the object definition of OODB systems. We tried to provide the database user with a high level language for specifying objects. At the same time, the code generated from an ODL specification is also capable of performing corresponding constraint enforcement. However, in order to provide the database user with a easy to use, more clear, and more efficient object definition language, a lot of work still needs to be done in the areas of object definition and user semantics maintenance.

Currently, the structural aspects of objects are defined separately from definitions of behavioral aspects. A more complete ODL could include both parts in a uniform language interface.

Further investigations of the semantics in various advanced applications are required to gain a thorough understanding of the application specific constraints.

In order to construct user views as complex objects, methods must be defined for automatic populating of the virtual class. A lot of work is still needed in order to support user views using complex objects.

Since one of the objectives of providing an ODL is to make it easy for the user to define the database, a more practical approach would be building a *Graphical User Interface* based on the ODL.

As a long term goal, much work has to be done on the building of an integration of the ODL with the predicate processing modules so that a customized query language with operations on complex objects can be automatically generated once the ODL definition has been provided.

Appendix A

Syntax of ODL

```
odl             ::= method ;
                  | odl method ;

method          ::= TYPE-ID CLASS-ID :: METHOD-ID ( parameters ) { do with }

parameters      ::= parameter
                  | parameters , parameter

parameter       ::= TYPE-ID ARG-ID

do              ::= DO : doactions

doactions       ::= doaction ;
                  | doactions doaction ;
```

```
doaction        ::= path . METHOD-ID ( arguments )
                  | path -> METHOD-ID ( arguments )
                  | FOREACH ( components ) : CLASS-ID . METHOD-ID ( arguments )
                  | FOREACH ( components ) : CLASS-ID -> METHOD-ID ( arguments )

arguments       ::= ARG-ID
                  | arguments , ARG-ID

components      ::= ATTR-ID
                  | components , ATTR-ID

with            ::= WITH : bcs
                  | WITH :

bcs             ::= bc
                  | bcs ; bc

bc              ::= WHERE PRE conditions : PRE { cactions }
                  | WHERE PRE conditions : POST { cactions }
                  | WHERE POST conditions : { cactions }
                  | WHERE TRUE : { cactions }

conditions      ::= conditions AND conditions
                  | conditions OR conditions
```

```
                | ( conditions )
                | condition
                | THEREIS OBJ-ID IN CLASS-ID THAT conditions

condition       ::= expr OP expr
                | expr
                | NOT ( expr )

expr            ::= path
                | NUM
                | predicate

path            ::= CLASS-ID
                | path . CLASS-ID
                | path INFER CLASS-ID

predicate       ::= path ( arguments )

cactions        ::= caction
                | cactions ; caction

caction         ::= FOREACH OBJ-ID IN CLASS-ID
                    THAT  POST conditions : { methods }
                | FOREACH OBJ-ID IN CLASS-ID
                    THAT PRE conditions : { methods }
```

```
                | TOP REJECTION
                | REJECTION
                | methods

methods         ::= method
                | methods ; method

method          ::= CLASS-ID . METHOD-ID ( arguments )
                | CLASS-ID -> METHOD-ID ( arguments )
                | CLASS-ID . METHOD-ID ( )
                | CLASS-ID -> METHOD-ID ( )

OP              ::= && | || | > | < | >= | <= | != | / | + | - | ==

ID              ::= {letter}{letter_or_digit}*

NUM             ::= {digit}+

letter          ::= [a-zA-Z_*]

digit           ::= [0-9]

letter_or_digit ::= [a-zA-Z_0-9]
```

Bibliography

[1] Serge Abiteboul and Anthony Bonner. Objects and views. In *Proceedings of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, 1991.

[2] Catriel Beeri and Tova Milo. A model for active object oriented database. In *Proceedings of the 17th International Conference on Very Large Data Bases*, September 1991. Barcelona, Spain.

[3] J. Blakeley. Efficiently updating materialized views. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, June 1986. Washington, D.C.

[4] José A. Blakeley, Neil Coburn, and Per Åke Larson. Updated derived relations: Detecting irrelevant and autonomously computable updates. In *ACM Transactions on Database Systems*, September 1989. Vol.14, No.3.

[5] R.G.G. Cattell. *Object Data Management: object-oriented and extended relational database systems.* Addison-Wesley, 1991.

[6] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the 17th International Conference on Very Large Data Bases*, September 1991. Barcelona, Spain.

[7] I-Min Amy Chen and Dennis Mcleod. Derived data update in semantic databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, 1989. Amsterdam.

[8] C.J. Date. *An Introduction to Database Systems.* Addison-Wesley, 1990.

[9] U. Dayal. Queries and views in an object-oriented data model. In *International Workshop on Data Base Programming Languages 2*, 1989.

[10] Umeshwar Dayal, Meichun Hsu, and Rivka Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of the 1990 ACM SIGMOD Int. Conf. on Management of Data*, May 1990.

[11] Lois M. L. Delcambre, Billy B. L. Lim, and Susan D. Urban. Object-centered constraints. In *Proceedings of the IEEE Data Engineering Conference*, 1991.

[12] Oscar Diaz, Norman Paton, and Peter Gray. Rule management in object oriented databases: A uniform approach. In *Proceedings of the 17th International Conference on Very Large Data Bases*, September 1991. Barcelona, Spain.

[13] N. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, September 1991. Barcelona, Spain.

[14] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proc. of ACM-SIGMOD 1992 Int'l Conf. on Management of Data*, 1992.

[15] S. Heiler and S.B. Zdonik. Object views: Extending the vision. In *Proceedings of IEEE Data Engineering Conference*, pages 86–93, Feburary 1990.

[16] H. V. Jagadish and Xiaolei Qian. Integrity maintenance in an object-oriented database. In *Proceedings of the 18th VLDB Conference*, July 1992. Vancouver, B.C., Canada.

[17] Anton P. Karadimce and Susan D. Urban. A framework for declarative updates and constraint maintenance in object-oriented databases. In *Nineth International Conference on Data Engineering*, April 1993. Vienna, Austria, pp.391-398.

[18] Won Kim. Object-oriented databases: Definition and research directions (invited paper). In *IEEE Transactions on Knowledge and Data Engineering*, 1990. Vol.2, No.3.

[19] Won Kim, Elisa Bertino, and Jorge F. Gazza. Composite objects revisited. In *ORION Papers*, 1988.

[20] Henry F. Korth and Abraham Silberschatz. *Database System Concepts.* McGraw-Hill, Inc., 1991.

[21] Ling Liu and Robert Meersman. Activity model: A declarative approach for capturing communication behaviour in object-oriented databases. In *Proceedings of the 18th VLDB Conference*, July 1992. Vancouver, B.C., Canada.

[22] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active data base management system. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, May-June 1989. Portland, Oregon, pp.213-224.

[23] Inc. Object Design. *ObjecStore User Guide, Release 2.0 For UNIX Systems.* Objeect Design, Inc., October 1992.

[24] Jack Orenstein, Sam Haradhvala, Benson Margulies, and Don Sakahara. Query processing in the objectstore database system. In *Proceedings of the 1992 ACM SIGMOD Int. Conf. on Management of Data*, June 1992.

[25] Tore Risch. Monitoring database objects. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, 1989. Amsterdam.

[26] N. Rousoupoulis. The incremental access method of view cache: Concepts, algorithms, and cost analysis. Technical Report CS-TR-2193, University of Maryland, February 1989.

[27] Elke A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proceedings of the 18th VLDB Conference*, 1992. Vancouver, B.C., Canada.

[28] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. On rules, procedures, caching and views in data base system. In *Proceedings of ACM-SIGMOD Conference on Management of Data*, May 1990. Atlantic City, NJ.

[29] Michael Stonebraker and Greg Kemnitz. The postgres next-generation database management system. In *Communications of the ACM*, pages 78–92, October 1991.

[30] K. Tanaka, M. Yoshikawa, and K. Ishihara. Schema virtualization in object-oriented databases. In *Proceedings of IEEE Data Engineering Conference*, pages 23–30, 1988.

[31] Susan D. Urban and Lois M.L. Delcambre. Constraint analysis: A design process for specifying operations on objects. In *IEEE Transactions on Knowledge and Data Engineering*, December 1990. Vol.2, No.4.

[32] Ju Wu. Implementation and evaluation of dynamic spatial query language. Technical report, Master's Thesis, Simon Fraser University, September 1992.

[33] Ju Wu and W.S. Luk. Implementation of a customized query language using a database programming language and its performance analysis. Technical Report CSS/LCCR TR92-18, Simon Fraser University, 1992.