I hereby state that this thesis contains only my own original work except where explicit reference has been made to the work of others.

A KNOWLEDGE-ORIENTED DATA MODEL AND ITS LANGUAGE

Dan Zhou

A thesis submitted for the degree of
Doctor of Philosophy
of The Australian National University

© Dan Zhou
JULY 1993
Statement

I hereby state that this thesis contains only my own original work except where explicit reference has been made to the work of others.

Dan Zhou

Dedicated to my parents, Jianchang Zhou and Qi Guo
Acknowledgments

This work could not have been completed without help, support and encouragement from many people.

Dedicated to my parents, Jianchang Zhou and Qi Guo

many invaluable suggestions and comments. Many of the ideas presented in this work originated in the discussions with Mike. Without his guidance, this work would not have been possible.

I would like to express my gratitude to Dr. Brendan McKay for his continuous encouragement, help, unlimited patient reading of this draft at every stage, and his kindness and friendship one can count on.

I have also benefited from help and discussions with Professor Robie Stair and Dr. Vicki Peterson. I would also like to thank all the staff members and research students in the Department of Computer Science for creating and maintaining such a nice research environment.

My special thanks to all my friends at University House for the love and support that they have given me.

And a very special thanks to my family for being there.
Acknowledgments

This work could not have been completed without help, support and encouragement from many people.

My sincere thanks go first to my supervisor, Professor Mike Papazoglow, for his many invaluable suggestions and comments. Many of the ideas presented in this work originated in the discussions with Mike. Without his guidance, this work would not have been possible.

I would like to express my gratitude to Dr Brendan McKay, for his continuous encouragement, help, unlimited patient reading of this draft at every stage, and his kindness and friendship one can count on.

I have also benefited from help and discussions with Professor Robin Stanton and Dr Vicki Peterson. I would also like to thank all the staff members and research students in the Department of Computer Science for creating and maintaining such a nice research environment.

My special thanks to all my friends at University House for the love and support that they have given me.

And a very special thanks to my family for being there.
Abstract

In this thesis we propose a data model and its language which aim at capturing more semantics and representing the naturalness of the real-world. It concentrates on a set of goals which have been attracting the attention of both researchers and practitioners for many years. These include: representing knowledge in the database domain to reflect the static and dynamic aspects of data, supporting relationships in the object-oriented data model in a generic manner, and providing a uniform programming language at the conceptual level. The thesis describes the design and implementation of a knowledge-oriented data model called KOM which is implemented by tight integration of the expert system shell CLIPS and the object-oriented database system ONTOS.

KOM is composed of some primitive concepts: object and type, and a set of programming statements to perform operations. Every object holds a type and the semantics of the object is described by this type. A type in KOM has four components: attributes, constraints, methods and triggers. Attributes describe the states of objects and constraints restrict the possible states of objects. Here, both passive and active constraints are allowed and some additional logical constructs are provided. With active constraints, one can enforce restrictions on causally related objects. This means that the modification of one party of objects can cause the constraints of other parties to be checked, even though the involved objects belong to different types. Methods are interface functions which are allowed to operate on the attributes of objects. Finally, triggers are defined as a set of rules which act on events that include updates of attributes, catching of messages and violation of constraints. The actions of triggers are classified as “before”, “after” and “deferred” actions. The execution of triggers depends on the rule firing strategies and the semantics defined for transactions. More general conditions and actions compared with previous trigger mechanisms are provided. Arbitrary programming statements can be used in the action bodies of triggers.

The design objective of the KOM relationship model is to provide a common theory for defining binary or n-ary relationships. This theory supports relationships as first-class objects that possess all the semantics held by ordinary objects. In addition, the concepts of slices and slice types are developed in order to capture semantics which is unique to relationships. The instances of a relationship type are not merely tuples, but rather sets of tuples. New constructs are introduced to describe the properties for both the relationship instances and the tuples defining the instances. Finally, complete rules for inheritance of relationship types are provided.

The final contribution of KOM is to support the concept of transactions in its language. This permits objects to be temporarily protected from constraint checking.
during a critical sequence of operations. Also, the incorporation of rules into KOM requires the system to distinguish when and where to check constraints or fire rules or continue the execution of operations. Both explicitly defined transaction statements and implicitly generated ones are provided by the language. Execution semantics are given for single transactions, nested transactions, dynamically derived transactions and rescue statements.
Contents

1 Introduction
   1.1 Overview .................................................. 1
   1.2 Structure of the Thesis ............................... 3

2 Background of the Research
   2.1 Introduction .............................................. 5
   2.2 Architectures of Expert Database Systems .......... 7
     2.2.1 Loose Coupling EDSs .............................. 9
     2.2.2 Tight Coupling EDSs ............................. 12
     2.2.3 Summary of Previous Systems ................ 15
   2.3 Issues in Knowledge-Based Data Modeling and Languages 15
   2.4 Early Experimental Prototypes .................... 18
     2.4.1 Architectures Used ................................ 19
     2.4.2 Data Models and Languages Supported ......... 21
     2.4.3 Analysis of Transformation Alternatives .... 22
     2.4.4 Discussion ......................................... 26
   2.5 Introduction to the Knowledge-Oriented Data Model 27
     2.5.1 Design Considerations ........................... 27
     2.5.2 The Knowledge-Oriented Model ................. 29

3 The Knowledge-Oriented Model — KOM 31
   3.1 Introduction ............................................. 31
   3.2 Type Specification .................................... 33
     3.2.1 Object States, Attributes and Representation 34
### 3.2.2 Object Behavior, Methods and Polymorphism

- 3.2.2.1 Introduction
- 3.2.2.2 Methods
- 3.2.2.3 Polymorphism

### 3.2.3 Object Semantics, Rules and Knowledge

- 3.2.3.1 Introduction
- 3.2.3.2 Passive Semantics
- 3.2.3.3 Active Semantics
- 3.2.3.4 Rule Evaluation

### 3.3 Constraints: Ensuring Semantic Integrity of Objects

- 3.3.1 Introduction
- 3.3.2 Passive Constraints
- 3.3.3 Active Constraints
- 3.3.4 Constraint Evaluation

### 3.4 Trigger Mechanism: Dynamic Maintenance of Object Semantics

- 3.4.1 Motivation
- 3.4.2 Syntax and Semantics of Rules
- 3.4.3 Run-time Semantics

### 3.5 Inheritance

- 3.5.1 Introduction
- 3.5.2 Inheritance of Attributes
- 3.5.3 Inheritance of Constraints
- 3.5.4 Inheritance of Methods
- 3.5.5 Inheritance of Triggers
- 3.5.6 Dynamic Binding

### 3.6 Comparison

### 4 General Functionality of the KOM Language

#### 4.1 Programming Statements

- 4.1.1 General Statements
- 4.1.2 Database Operations

#### 4.2 Transactions

- 4.2.1 Motivation
- 4.2.2 Transaction Syntax
- 4.2.3 Execution Semantics of Transactions
- 4.2.4 Failure Handling
- 4.2.5 Comparison

#### 4.3 Functions and Methods

- 4.3.1 Introduction
- 4.3.2 Function Syntax
- 4.3.3 Method Syntax
- 4.3.4 Execution Semantics of Functions
- 4.3.5 Execution Semantics of Methods
- 4.3.6 Failure Handling
- 4.3.7 Comparison
4.3.1 KOM Programs
4.3.2 Construction of Objects
4.3.3 Methods for Accessing Object States
4.3.4 Constraint Methods
4.4 Query Statements — Part I
  4.4.1 Single Query Statements
  4.4.2 Embedded Queries
5 Relationships: Description of Associations among Objects
  5.1 Overview
  5.2 Definitions and Semantics of Relationships
  5.3 Modeling Relationships
    5.3.1 Slice Types
    5.3.2 Attributes
    5.3.3 Constraints
    5.3.4 Methods
    5.3.5 Triggers
  5.4 Generic Relationship Type
  5.5 Inheritance
    5.5.1 Introduction
    5.5.2 Inheritance of Participants
    5.5.3 Inheritance of Slices
    5.5.4 Inheritance of Attributes, Constraints, Methods and Triggers
    5.5.5 Dynamic Binding
  5.6 Query Statements — Part II
  5.7 Discussion
6 Implementation of KOM and its Language
  6.1 Introduction
  6.2 Interface between Clips and Ontos
  6.3 Transformation from KOM Programs to C++ Functions
  6.4 Enforcement of Constraints
6.5 Trigger Mechanism ................................................. 127
6.6 Implementation of Transactions and Rescues ................. 128
6.7 Construction of Relationships ................................... 129
6.8 Inheritance Mechanism ............................................ 133

7 Conclusion ...................................................................... 138
7.1 Summary ................................................................. 138
7.2 Future Research ........................................................ 140

A Examples of KOM Programs ............................................. 141
A.1 Introduction ............................................................. 141
A.2 Type Lecturer Definition ............................................. 141
A.2.1 The lecturer.def File ............................................. 142
A.2.2 The lecturer.fun File ............................................. 143
A.2.3 The lecturer.h File ................................................. 143
A.2.4 The lecturer.c File ................................................. 144
A.2.5 The lecturer.rule File ............................................. 149
A.2.6 The lecturerfun.c File ............................................. 149
A.3 Relationship Type take Definition .................................. 152
A.3.1 The take.def File ................................................. 152
A.3.2 The take.fun File ................................................. 153
A.3.3 The take.h File .................................................... 154
A.3.4 The take.c File .................................................... 155
A.3.5 The take.rule File ................................................. 160
A.3.6 The takefun.c File ................................................. 161
A.3.7 The take.slices.h File ............................................. 164
A.3.8 The take.slices.c File ............................................. 166

Bibliography .................................................................. 169
List of Tables

2.1 Summary of Features of Previous Systems ........................................... 15
2.2 Comparison and Summary of Features of Prototypical Systems ............... 23
3.1 Syntax of Types ....................................................................................... 34
3.2 Syntax of Attributes ............................................................................... 35
3.3 Syntax of Methods ................................................................................... 36
3.4 Syntax of Constraints ............................................................................. 40
3.5 Syntax of Passive Constraints .................................................................. 41
3.6 Syntax of Active Constraints ................................................................... 45
3.7 Syntax of Triggers .................................................................................. 49
4.1 Syntax of General Programming Statements ......................................... 69
4.2 Syntax of Database Operations ............................................................... 71
4.3 Syntax of Object Operations ................................................................. 72
4.4 Syntax of Transactions .......................................................................... 74
4.5 Syntax of Rescues .................................................................................. 77
4.6 Syntax of Functions and Methods ............................................................ 82
4.7 Syntax of Programs ................................................................................ 82
4.8 Syntax of Constructors .......................................................................... 82
5.1 Syntax of Relationship Types .................................................................. 93
5.2 Syntax of Slice Types ............................................................................ 96
5.3 Syntax of Relationship Attributes .......................................................... 97
List of Figures

2.1 Architecture of Loose Coupling EDSs ........................................ 10
2.2 Architecture of Tight Coupling EDSs ........................................ 13
3.1 Definitions of Type Person and Lecturer .................................... 32
3.2 Attributes of Type Person ....................................................... 36
3.3 Example of Casual Related Constraints ..................................... 44
3.4 Example of Active Constraints ................................................ 45
3.5 Local Variables in Rule Definitions .......................................... 52
3.6 An Event and its Transaction .................................................. 54
3.7 Partnerships ............................................................................. 55
3.8 KOM Type Hierarchy ................................................................ 57
3.9 Dynamic Binding of Attributes ................................................ 63
3.10 The Order of Dynamic Binding ................................................. 65
4.1 Salary Transaction .................................................................... 74
4.2 Illustration of Transaction Execution ....................................... 76
4.3 Rescue Statement in Method newSalary() .................................. 78
4.4 Constructor for Type Lecturer ................................................... 83
4.5 Example of Query Statements .................................................... 85
4.6 Example of Embedded Query Statements ................................... 86
5.1 Relationship of Students Taking Courses on Some Majors ............ 92
5.2 Relationship Type take ............................................................. 95
5.3 Method Specifications ............................................................. 101
5.4 Transactions on Relationships ................................................ 102
5.5 Relationship Type Hierarchy .................................................... 106
5.6 Querying Participants of Relationship Type take ......................... 113
Chapter 1

Introduction

1.1 Overview

The work described in this thesis was originally part of a large project which was planning to investigate issues in the area of Knowledge-Based Database Systems and design a new-generation information system featuring by knowledge and object-orientation. It started with the investigation of new architectures of expert database systems and later on went to the integration of DBMSs with expert system shells. Several important research topics were proposed in the area, including reflective architecture, meta-level specification and reasoning, representation of knowledge and data under one unified scheme, etc. Two trial prototypes have developed and the feasibility of designing such systems was shown. As the research proceeded, interest switched from the architecture and meta-object issues to more fundamental ones, data modeling and database language. We found that before any of these new ideas could be realized, we needed to know what are the essential constructors for a data model, whether and how they can be represented and implemented as a real database system. Thus, capturing more semantics and knowledge in a database model became the centre issues in this research. This is finally accomplished by the construction and implementation of a knowledge-oriented data model KOM and its language.

In Chapter 2, the background of the research is reviewed. One can still see the outline of previous research and the results of prototype systems. The reason that we include these into the thesis is that those issues affect the way that we ended with...
designing KOM and its language, specially the features provided by the KOM system. Like most of the expert database systems which try to capture knowledge into their data models, KOM pursued the same aim but resulted in a slightly different direction. The difference is that KOM uses an object-oriented language to represent its model. Unlike traditional database languages, the new one is an enhancement of the integration of object-oriented programming languages and expert systems.

The KOM system supports persistent storage of objects. That is all the objects defined by the KOM language are persistent, except for temporary variables. In general, DBMSs allow multiple users and thus mechanisms for concurrency control and transactions are provided. However, the implementation of KOM only supports a single user at this stage. Many issues like rule firing and transaction execution are only discussed under the single-user assumption. This results in the execution environment of KOM being rather limited.

The concept of rules is used very often in the whole thesis. In KOM, two components of an object type can be regarded as rules. One is constraints which restrict the semantic domain of objects. Since constraints imposed on one object may be related to or inherited from other objects, the checking of constraints in different orders may result in different evaluation results. To avoid non-determinism of constraint checking, rules or orders must be enforced and they should be consistent with inheritance and rule firing.

The other type of rule in KOM is the trigger. Triggers behave like rules in expert systems and they are fired when certain conditions are satisfied. Different from triggers in relational databases which fire rules according to the update of a set of data, rules here are defined for objects and thus react regarding of individual object’s behavior. In other words, triggers are activated before or after object updates or method execution. However, they are only fired when the system decides so. Questions like when rules’ conditions are checked and fired, how rules’ actions are executed, and more fundamentally how to specify the behavior of rules are answered. The solutions to these problems are proposed under the KOM execution assumption. More desirable features or extension to the solutions are left for future research.

In the following of the thesis, we will introduce the background of the research and
Chapter 1. Introduction

the work that has been done for designing KOM. Before that, we conclude this chapter by the structure of the thesis.

1.2 Structure of the Thesis

In Chapter 2, the review of the literature is given and three types of architectures used for building expert database systems are compared. In addition, some important issues in this area are investigated and possible solutions are described. The basic design considerations for the knowledge-oriented data model and an overview of KOM were shown at the end of the chapter.

Chapter 3 starts with the introduction of basic concepts used for KOM type definitions and the syntax. Three major issues are discussed in detail, namely constraints, triggers and inheritance. The semantics and precise uses of constraints and triggers are described. At the end of the chapter, inheritance between types is elaborated in the context of KOM.

Chapter 4 introduces the details of the KOM language which is defined in this unified environment. Some specific operations allowed on KOM databases are introduced. In addition, methods generated by the KOM system for user-defined types are described. The central parts of the language including the transaction and rescue statements are introduced. Semantics of transactions on conceptual levels is defined and several approaches for execution of them are described.

To support full object-orientation for the data model, associations of objects are regarded as first class objects in KOM. Chapter 5 investigates the problems and missing semantics of in current models or definitions, and discovers a series of completely new constructs to model n-ary relationships. This is a generic model which can be applied to any object-oriented systems. Query language and inheritance related to relationship objects and types are also proposed.

Chapter 6 reports the implementation results from the experiments which have done to build the KOM system. Two basic systems contributed to our experiments are introduced, which are the expert system shell Clips and the object-oriented database system Ontos. Integration of the two systems is accomplished through the C++ lan-
Chapter 1. Introduction

guage, and extensions of Ontos are made to achieve the persistent knowledge base. In particular, issues such as constraints, triggers, transaction and rescue statements, inheritance and relationship constructions are mainly discussed in this chapter. The transformation from the KOM language to C++ is introduced and realized by means of the software tools: Yacc++ and Lex++.

Chapter 7 provides a summary of the whole thesis and further discussion on building of the knowledge-oriented data model and its language. Future directions of work are proposed to meet the challenge of new generation of knowledge-based database systems.

Finally, Appendix A gives the KOM programs for the examples used in this thesis. These are followed by the C++ code which are generated by the KOM compiler and are used to help readers to understand the implementation of KOM.
Chapter 2

Background of the Research

2.1 Introduction

A primary research objective in the database area is to develop more advanced database languages and richer data models to accommodate complex applications. The emergence of numerous data models has demonstrated that new generation database systems should provide multiple functionalities, high level representation and combined techniques from different disciplines. This demands that characteristics from AI and programming paradigms are incorporated into data modeling constructors. Several types of systems have been developed in the past few years which claim that both knowledge and data semantics of objects from the real world should be captured at a conceptually natural level.

Combining AI technology, i.e., knowledge or rules, into data models is regarded as the most common and feasible approach for achieving advanced functionalities of database systems. In general, rules can be integrated into database systems in two general forms[59]. Firstly, a declarative rule takes the form of if A is true, then B is true, which states that the hypothesis A implies the hypothesis B. When the facts of A exist, the new information B is deduced. Database systems integrated with logic programming languages like Prolog are often designed in this way [24] [95] [109]. Secondly, an imperative rule holds the form of if A is true, then do C, which means that actions C must be taken when conditions A becomes true. The new actions C can be statements which enable or disable conditions, or cause other actions to be executed.
Chapter 2. Background of the Research

This is very useful for database systems to monitor and control users’ operations, and maintain the integrity of data [50] [98] [108]. For example, updates of database objects may result in the changes of objects’ states and constraints are thus violated. If rules are attached to objects in the database, actions can be automatically taken to repair the constraint violation. A sequence of events can be conducted through rule firing.

Rules from the AI paradigm offer a key contribution to building new types of database systems [2] [70] [89]. In addition, programming languages, particularly object-oriented programming languages, are incorporated into the design of current database systems to assist the implementation of procedure-based applications [54]. The categories of database systems can be classified through the ways of how database systems integrate with rules or knowledge. For example,

1. **Semantic Databases:** Semantic data models allow database designers to represent objects of interest and their relationships within an application in a manner that closely resemble the view that the user has of these objects and relationships [13] [47] [81]. Since the integrity and inter-consistency between objects are part of the semantics of objects, semantic models provide mechanisms for the definition of integrity constraints and at the same time allow them to be maintained to get correct view of databases. Here, rules are used to specify and enforce constraints on objects.

2. **Active Databases:** In active database systems, rules are used as a general mechanism to support timely critical situations and monitoring the changes of system’s states. Constructors such as assertions, alerters, triggers and database procedures are adopted to provide timing constraints, access control and inferencing [20] [62] [80]. Basically, rules are used to act as the specified changes in data or other events occur.

3. **Deductive Databases:** These usually refer to those systems which use logic programming languages as the main vehicle to specify, manipulate and reason about objects in databases [40] [104]. The logic programming paradigm supports deduction and new information can be derived from reasoning on existing data in the databases [4] [83] [105].
4. **Knowledge-Based Databases:** Knowledge-based database systems are characterized by having declarative languages and supporting principle capabilities of database systems [104]. The functionalities of knowledge-based database systems include putting rich knowledge representation schemes into data models, providing deductive reasoning facilities, and improving performance of data and knowledge management. Rules in knowledge-based database systems can be used to express different types of semantics and functionalities, and emphasize different perspectives of applications [16] [34] [84]. The enhanced systems could be a combination of some types of the systems mentioned above.

The systems which are more interesting to us are the knowledge-based database systems since they capture most promising and essential features for the next generation of information technology. Sometimes, this type of system is called an **expert database system** (EDS) because it is commonly implemented by integrating expert and database systems. In the following chapters of this thesis, we investigate the problems existing the current expert database systems and propose a new data model and its language based on our research.

This chapter is organized into four sections. It starts by reviewing the architectures of expert database systems and is followed by the summary of previous systems. In Section 2.3, several important issues related to the knowledge-based database systems are pointed out and possible new directions of developing the systems are described. Section 2.4 shows the earlier experimental prototypes developed by the author and how they affected the design of our new model and its language. The last section gives the principles of design and a brief description of KOM.

**2.2 Architectures of Expert Database Systems**

The motivation for building expert database systems primarily came from two requirements, including (i) efficient management and access to large amounts of knowledge for reasoning; (ii) intelligent processing of data [52] [69] [78] [93]. In general, this type of system requires the expertise to reside within database systems to provide intelligent question answering, to automatically enforce integrity constraints and to
combine knowledge- and data-driven search techniques into efficient inference schemes. Alternatively, expertise may reside outside the system in knowledge-based applications, with database systems simply utilized to store data. In this case, expert systems interpret data retrieved from databases, make decisions based on them and send the results back to the databases. Since the behavior of expert database systems strongly depends on the way that database systems are integrated with expert systems, the study of possible system architectures is very necessary and has attracted a lot of attention in the past few years.

The design and construction of expert database systems may take many forms according to application requirements. Theoretically, they can be classified into three types: indirect-coupling, loose coupling and tight coupling. In an indirect-coupling system, the linkage between the DBMS and ES is established by using external software written in a high level language, such as Pascal and C. This is very similar to writing application programs which call the conventional databases. Obviously, it is very inconvenient for users since every application requiring to access the database requires its own interface program.

A loose coupling EDS is one where the expert and database systems maintain their original functionalities and communicate through a well-defined interface. From this structure, we know that the transformation between the two systems has to be considered by users who use either the database or the expert system as the major system. The resulting system usually produces a user interface which embeds one language of a system into another.

To provide a uniform interface and a better performance overall, an architecture of tight coupling EDSs is chosen in which one of the integrated systems is used as the front-end and the other system is embedded into it. Under this structure, users interact with a single interface which can be the language of either of the integrated systems or a completely new one defined for the resulting system. The communication interface between the integrated two systems is invisible to the user.
2.2.1 Loose Coupling EDSs

A loose coupling structure is a feasible approach to integrate database systems and AI technology [51] [93] in some circumstances. Under this structure, a DBMS usually acts as a data resource providing factual data and an expert system uses them to perform reasoning. A typical architecture for a loose coupling EDS is depicted in Figure 2.1. Here, the expert system is used as the major system to communicate with users, and the database is used for persistent storage of facts and rules. The transformation manager is required between the two systems so that the forms of facts and rules in the expert system can be translated into the forms of data in the database system, and vice versa.

Internally, the knowledge base of the expert system is assumed to have both rules and facts. In most cases, the transformation between the DBMS database and the expert system only affects the fact component of the knowledge base, since rules usually have much smaller size and much more complex structure compared with facts. Moreover, the transformation of rules between the two systems can greatly decrease the system performance [72]. Facts are fetched from the database into the knowledge base on an as-needed basis, with no clear policy on when it is required. In reverse, revised facts copied back from the expert system into the persistent store of the DBMS face the same kind of situation.

Loose coupling architectures require users to consider when and how knowledge can be transferred between the two systems. For example, the user must issue commands to explicitly retrieve necessary facts from databases before or when activating rules. Generally, query commands used to access facts in the database are embedded inside rule specifications. They can be executed whenever rules' conditions are satisfied and rules are fired. Since loose coupling systems leave the responsibility of the interaction between the two systems to users, a weakness of this type of structures is that the uses of database systems are not transparent. The user needs to take care of two systems and to use two different types of languages as well.

Most of the reported work with loose coupling architectures are implemented by integration of a logic programming language, often Prolog, with a relational DBMS, often
interfaced with SQL [24]. This is a very obvious alternative because both of the systems are defined by first order logic and can thus be transformed easily. However, they also have difficulties when coupled together. This is because Prolog follows a tuple-oriented algorithm, while database systems are set-oriented. Moreover, the semantic contents and functionalities provided by these two systems are significantly different. For this reason various strategies have been developed for realizing and optimizing transformations of knowledge between databases and Prolog [88] [91]. For instance, some EDSs are designed by enhancing Prolog with predicates which allow it to operate and access data directly on relational databases. Others introduce intermediate languages or loading mechanisms between Prolog and SQL. All necessary processes between the two systems are done in this intermediate level, such as query optimization or putting restrictions on possible transformations. Rules are written with SQL commands whenever accesses of the database are required. Systems developed in terms of the latter consideration include PROSQL, Quintus/UNIFY, etc.

An EDS which is built differently from these systems is the KEE/Connection system [1] [3]. It permits the transformation of the frame-based information into a fully normalized relational database structure. Data transformation of KEE/Connection works as follows. Whenever a KEE-based application issues commands containing retrieval of
Chapter 2. Background of the Research

data from databases, KEE/Connection's transformation module is used which generates SQL statements to realize the appropriate data transfer according to the mapping formulate. The SQL statements are then sent to the DBMS and the retrieval data is returned to KEE/Connection. The KEE/Connection then translates the data from the relational format back to frame structures. With this architecture, facts can be maintained in one or more relational databases and accessed by a number of KEE-based applications.

Analysis of loose coupling architectures shows the result that they have fairly simple and easily implemented structure. However, this results in that in certain circumstances, loose coupling systems produce poor performance due to the lack of considerations of mismatch of the two paradigms, and the overall concern of optimization of queries. In certain extent, the performance of loose coupling EDSs can be improved by adding expertise into transformation managers [72]. Performance related issues, such as complexity of the queries, frequency of queries and access patterns have to be addressed in the expertise.

The loosely coupled expert systems with object-oriented database systems use quite different methodology from the integration mentioned above, since the latter one supports more flexible and powerful representation for both data and knowledge. By connecting these two systems, several alternative structures can be considered, specially when expert systems also have high-level programming language interfaces, such as PASCAL, C or C++. Due to the object-oriented representation forms, both expert systems and DBMSs can be used at the front-end and in either cases database operations can be embedded into rule specifications. When an object-oriented data model is used as the major representation form, rules can be defined either inside object types to describe the behavior of objects (used as demons in frame-based systems), or inside general rule classes which store all rules to make inference for certain applications [31] [101]. The latter provides an image that rules can access all the objects specified by the object model as well as control or inference on them. The structure of these types of systems simulates frame-based systems by which object-models are used as the major knowledge representation, while rules are the main vehicles to operate on objects just like procedures. The difference between the integrated systems and frame-based
systems is that the integrated systems make knowledge persistent and provide facilities to manage them in the database as well.

The architecture of integration of expert systems with object-oriented databases is very similar to those with relational ones. However, the expert system can be used at either the front-end or back-end but in both cases object-oriented database systems are utilized for storage of persistent knowledge. Of course, the transformation policy is much simpler compared with the previous ones.

Unlike relational database systems, object-oriented databases support storage of complex and varied lengths of data without losing their efficiency. Individual rules can be simply stored in their plain forms without decomposition or in intermediate data structures as used in AI systems. The choices really depend on the systems to be integrated and application requirements. Sets of rules are classified into different classes in accordance with their application needs. The retrieval of facts and rules from underlying database systems is quite simple and is done by retrieving objects from certain classes which store them.

2.2.2 Tight Coupling EDSs

As pointed out in [97], the loose coupling architecture has several severe disadvantages. Due to the separation of facts in main memory and facts from the database, it may happen that during the inference period, the facts stored in the database have been modified, but their copies in cache for reasoning still remain the same, or vice versa. This causes inconsistency of facts and produces wrong results for either the expert system or database users. Most loose coupling expert database systems do not provide persistent rule bases. Hence, any changes to rules would not be preserved unless manual procedures to save them are utilized. Moreover, the rule bases which are shared and modified may not be noticed by other concurrent users. Because of these disadvantages and those addressed previously, an architecture called tight coupling is proposed. Using this approach, both the database and rule base are managed under the same regime.

In general, a tightly coupled expert database system is a corporate environment where an expert system and DBMS are integrated into one framework which supports
cohesive representation of semantics of both data and knowledge. From Figure 2.2 one can see that a uniform language is provided for the integrated environment. This language allows users to define applications with combined knowledge from both the AI and database areas. There are no explicit commands to be issued to retrieve or store data or knowledge from the database. Users simply use the language to capture features from two paradigms.

An intermediate processor is provided to handle the transformation between the underlying two systems and the interface. Operations are analyzed at this stage and some of them may generate commands which access the database. The precise procedure for transformation depends on the policy adopted by the designer of the system.

Under the tight coupling architecture, both fact and rule base are persistently stored in the underlying DBMS, and thus are shared by multiple users. No inconsistency occurs when updates of either facts or rules are imposed during inferences.

As pointed out in [79], several alternative structures exist for merging between the expert system and the DBMS under tight coupling scheme. The first option is proposed
to enhance the functionalities of the DBMS by adding knowledge representation and inference facility into the data model. Usually these systems are built on top of DBMS and use rules to implement constraints and triggers. Systems like POSTGRES are very good examples to illustrate this type of structures. POSTGRES was implemented by adding rules into the relational database system Ingres and enhanced by providing control of integrity constraints and a trigger mechanism. In the rule system of POSTGRES, rules are triggered by events, which can be database operations such as retrieve, replace, delete, append, new and old [98]. The actions of rules are POSTQUEL commands and they can be used to reject, derive or update data in the database. Two control flows of reasoning are supported: forward chaining and backward chaining.

The second type of tight coupling uses an expert system shell or logical programming language as a main component and enhances it by DBMS functionalities. Under this structure, facts and rules are in fact manipulated on a persistent basis and optimization of queries of knowledge is achieved. For example, LDL developed at MCC is a typical system to illustrate this idea [103] [102]. As designed to enrich logic programming paradigm with relational database primitives and constructs, LDL supports data structures including atoms, complex objects, lists and sets of objects. It uses bottom-up evaluation technique and retrieves data from the database a set at a time instead of a tuple at a time. It adopts extensive semantic analysis and optimization for queries based on a whole rule set, thus achieving the notion of tightly coupled EDSs.

The third type of tight coupling architecture is a unified system built from scratch. For instance, the Knowledge Data Model KDM [84] is designed using this approach to support conceptual level representation. It is rooted on the functional data model DAPLEX [94] and proposed to capture the data and knowledge semantics from one enterprise. Basic abstractions from semantic or object-oriented paradigm, such as classification, generalization, aggregation and membership are incorporated into the data model. Knowledge represented as constraints and heuristics is captured by a type definition. The behavior of objects is not supported at this stage.

The last type of tight coupling architecture treats the expert system and the DBMS independently and merges them as separate subsystems into a single one. Apparently, seldom existing systems adopt this structure. Analysis of this architecture will be
Chapter 2. Background of the Research

<table>
<thead>
<tr>
<th>Features</th>
<th>KEE/Connection</th>
<th>POSTGRES</th>
<th>KDM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Loose Coupling</td>
<td>Tight Coupling</td>
<td>Tight Coupling</td>
</tr>
<tr>
<td>Implementation</td>
<td>Transformation between KEE and Oracle</td>
<td>Built on Ingres</td>
<td>Built from Scratch</td>
</tr>
<tr>
<td>Persistent KB</td>
<td>Facts</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Representation</td>
<td>Frame-Based</td>
<td>Relational</td>
<td>Functional</td>
</tr>
<tr>
<td>Enhancement</td>
<td>KEE</td>
<td>Ingres</td>
<td></td>
</tr>
<tr>
<td>Knowledge</td>
<td>Demons and Lisp Rules</td>
<td>Triggers</td>
<td>Heuristics</td>
</tr>
<tr>
<td>Programming Language</td>
<td>Lisp</td>
<td>None</td>
<td>KDL</td>
</tr>
</tbody>
</table>

Table 2.1: Summary of Features of Previous Systems

discussed later due to that the system concerned in this thesis is mainly designed under this structure.

2.2.3 Summary of Previous Systems

A summary of features of previous expert database systems is given in Table 2.1. It compares three famous systems: KEE/Connection, POSTGRES and KDM by considering features such as architectures used, implementation methods, the knowledge bases provided, the enhanced systems, representation based and other abilities. The attempt is to show the design philosophy of different expert database systems.

2.3 Issues in Knowledge-Based Data Modeling and Languages

Despite the fact that knowledge-based systems and databases have become mature over the past two decades and despite the fact that more advanced technologies have been developed, we are still faced with problems of providing sufficient and more advanced tools for complex applications. Also, different computing technologies are developed in isolation and the capabilities of integrated technologies have not been fully investigated. Currently, there is an increasing awareness of this fact and it seems that the field of integrated systems, especially knowledge-oriented database system, is
headed for a revolutionary transition of database technology.

As mentioned above, research has been carried out on the integration of databases with AI systems to enrich data models by adding inference engines into DBMSs. Also, the incorporation of features from programming languages into database systems became popular as object-oriented programming languages appeared. A number of database systems have been designed for this purpose, such as KDM and STARBURST. A few years ago, a knowledge-based database language called KDL was developed by Potter which is based on the Knowledge Data Model (KDM) [84]. KDM is an extension of semantic data models and provides an object-oriented view of both data and knowledge. KDM models primitives which come from three areas: semantic data models, object-oriented programming languages and AI knowledge representation formalisms. In detail, the constructors defined in KDM include generalization, classification, aggregation, membership, temporal, constraints and heuristics. Constraints and heuristics are the two primitives which are presented as knowledge of objects and both the integrity of data and specification of rules are allowed to be defined within object types. The major contribution of KDM is to find an appropriate way to represent data and knowledge in a uniform manner.

STARBURST is another good example of combining knowledge and data within one data model [59]. STARBURST has been developed to capture features of the next generation of database systems which are influenced by modern programming languages and expert system techniques. Characterized by modeling application-specific semantics and behavior and user-defined production rules, the new language is facilitated by unstructured and structured complex objects, user-defined types, functions, and rules. The rules defined in STARBURST provide an alert trigger system under which users can define triggers at high level abstractions and with different semantics. Unlike KDM, STARBURST is built as an extension of relational database systems.

Even though the integrations produce several new types of database systems which stress different aspects of complex applications, the capabilities of them are not fully developed. In many situations, they are not sufficient enough to provide advantages for richer data models and functionalities obtained from each paradigm. The user interfaces and underlying connections are implemented by seeing the traces of integration of the
two technologies [8]. Indeed, some fundamental issues are not well proposed or solved in the design of knowledge-based database systems, which include, for example,

• **complex architectures**: The loose coupling and tight coupling provide the basic architectures for integration of expert and database systems. However, they are not sufficient to meet some specific requirements of increasingly complex applications. More powerful architectures must be developed to accommodate specific functionality in database systems [32] [60] [61]. For example, in some cases, a system is required to describe what it can do and how it performs [39]. Similarly to the way that meta-rules control the firing of rules in AI systems [6] [29] [28] [45], meta-data or even meta-knowledge can be specified at the meta-level in databases to reflect what the system has in the conceptual schema level or object level [48] [60] [77]. The meta-knowledge represented is subjected to the requirements of applications. For instance, meta-knowledge can be used to monitor the systems, control the behavior of databases and provide information for implementations [18] [25] [33] [36] [43]. The relationships between object level, type level and meta-level of the systems are explored in order to support more sophisticated integration architectures.

• **the synthesis of knowledge and data representation and management**: The issue here is in what forms that knowledge can be incorporated into one data model to enrich its capabilities [15] [100]. Further, more semantic abstractions are investigated and the language constructors corresponding to the abstractions need to be expressed [73] [82]. Finally, the appropriate features of unified languages have to be studied. Languages which incorporate features of programming languages, database operations and knowledge representation need to be developed.

• **different perspectives of real world, like the concept of view used in relational systems**: Relational database systems support derivation of user-defined views of databases according to the schema definitions in databases. Object-oriented database systems do not support view concepts since they are newly developed systems and the structures of objects are quite complicated. Suitable but sufficient views of objects, rather than selections of part of objects in classes, are
difficult to construct. With the combined knowledge in a database system, the knowledge-based database systems should be able to generate multiple perspectives of the user's view automatically according to the user's descriptions and underlying data [74]. With regard to knowledge and data stored in a database, hypotheses can be proved and information about the real world can be gained by reasoning.

- **active database systems**: Most database systems are passive in the sense that they do not react to the events or errors occurring in databases. The new types of database systems can be characterized by their provision of automatic object behavior. Features like inter-object message passing, actively maintaining object semantics and reflections on updates of databases will be captured by active database systems. More importantly, the behavior of object activity itself is part of its semantics and the execution of it will help users to make decisions on both objects and applications.

This thesis only examines some of the issues addressed above. Basically, it emphasizes the semantic enrichment of data models by utilizing knowledge representation from the AI paradigm. In addition, some fundamental issues of object orientation are reconsidered and improvements are proposed. The work presented in this thesis is concerned about designing a knowledge-oriented data model and its language by means of integration of expert systems and DBMSs.

### 2.4 Early Experimental Prototypes

Two prototypes have been developed by the author to study the behavior of expert database systems. They use the relational database system Oracle, the expert system shell Clips and the object-oriented database system Ontos, respectively. Here, Clips is designed with an interface to an advanced programming language, such as C and Pascal. Ontos is implemented by C++ and any C++ definitions are compatible with it.

The first prototype of the research attempted proposed to integrate a relational database with an expert system. This was achieved by loosely coupled Oracle and
Clips. Clips was used as a front-end and queries from SQL embedded into production rules. Rules, facts and database data stored in the Oracle database in order to make the knowledge persistent. We call this prototype Clips+Oracle.

The second prototype was loosely coupled Clips and Ontos. In this case, the knowledge-based database system is enhanced by Ontos which allows more efficient storage of knowledge into the database and more powerful representation forms. In addition, rules are included in object type definitions to help make general inferences on object states and behavior. We call this prototype Clips+Ontos.

These two prototypes were experimental systems implemented using similar methods to those of previous systems. They were aimed gaining some first-hand experimental results to support the third prototype mainly discussed in this thesis. The purposes for the experiments were

1. comparing the benefits and shortcomings of two different methods for integrating similar pairs of systems.

2. gaining insight into the nature of the integration problem in particular identifying the most necessary characteristics of the systems being integrated.

2.4.1 Architectures Used

Loosely Coupled Clips with Oracle

Integration of Clips and Oracle was the first experiment of our research. In the combined system, Clips is used to process data and make inferences on existing facts. Oracle is simply a storage facility and SQL is the language which sends data to Clips and stores them back from reasoning. Communication between the two systems is not very complicated and a C language interface between Clips and Oracle is provided so as to implement the transformation channel between the two systems.

Clips and Oracle have different mechanisms to process data or facts. Clips fires rules by considering one fact each time. Rules are activated when that fact makes the rule's conditions satisfied. On the other hand, Oracle is a set-oriented system which retrieves a set of tuples once a command is issued. The execution of one SQL command cannot be stopped unless all the tuples in a relation are processed. This implies that
facts must be pre-prepared before a certain set of rules is fired. This problem is solved by loading or saving sets of rules and facts before one application task is executed. SQL commands can also be specified in a rule declaration. Consistency between the two systems is considered by users when they define the rules.

Once a large number of rules are defined, rules have to be grouped into different small tasks in terms of a classification of the subtasks provided by the user. Only those rules related to a single task or a subtask are kept in memory at any moment. Facts and rules are cleared after one task or one subtask has been completed. The loosely coupled system requires users to consider which and when data and rules are retrieved from an Oracle database into memory and when and which of them need to be cleaned up. Commands for clean up, save or load tasks are implemented as rules or actions of rules. They are executed when certain conditions are satisfied.

Loosely Coupled Clips with Ontos

The architecture of loosely coupled Clips and Ontos is very close to that of tight coupling, where Ontos is considered as the major enhancement. Due to the flexibility and capability of Ontos, two interfaces can be provided at the same time and allow users to communicate with the resulting system either from Clips side or from Ontos side. When users interact Ontos through Clips, commands of database operations are issued through firing of rules. These commands are implemented by the Clips external function interface and embedded into rule specifications. Rules can be invoked by Ontos through Clips’ C interface to perform operations on Ontos objects. In addition, knowledge from Clips can be organized into different sets in terms of the tasks that they execute. When the set class Rules and the set class Facts are defined, rules and facts corresponding to certain tasks are stored as one set of objects and a unique identifier related to the task is assigned. The structure can be extended by defining new classes of either Rules or Facts. Unlike the integration of Clips with Oracle, there is no explicit transformation occurring between Clips and Ontos due to that both of them are compatible with C language and Ontos also adopts one object processing policy for each operation.
2.4.2 Data Models and Languages Supported

These experimental prototypes aim at providing a set of database and expert system functionalities in their own systems. Thus, data models or languages are enriched from their original ones. Briefly, the integrated systems have been studied to overcome the basic limitations of either the original data or knowledge representation or languages. The enhancements thus are provided which are based on the following considerations:

1. The integration of abstraction mechanisms of data models into expert systems can support more efficient and powerful storage and queries of data and knowledge.

2. It helps to deal with various aspects of data, including semantic and operational perspectives. This result is strongly influenced by AI knowledge representation formalisms [17] [34].

3. It allows the development of more expressive languages for database operations and inferences on them.

Although the experimental prototypes have been developed to address the same application domain, they differ from each other in the way that the database operations are supported. In the following, the major characteristics of the languages and representational forms are summarized and compared according to some general principles.

Static Features

The static features of data models are concerned with the structural properties of data. They consist of some basic concepts like entities, attributes, relationships and integrity constraints. Entities here are used to represent both real world objects and abstract concepts. Not all the database systems have direct mappings from entities to database formulation. According to the definition, tuples and tables are referred to as entities in relational database systems, while types and objects are also regarded as entities in the object-oriented paradigm. Attributes are states of objects and essential structural properties of objects are described by them. Similarly, relational databases describe object attributes by the fields declared in the table definitions. The values of certain objects may be obtained by joining possible tables which are linked
Chapter 2. Background of the Research

by foreign keys. Representing relationships or abstract data types is another feature of object-oriented database systems. Abstract mechanisms, such as generalization, classification, aggregation, association and other semantic constructors [7] [8], are not fully supported by every database system. Thus, they may not be provided by the integrated ones. The static feature also concerns integrity constraints. To enforce the correct information on databases, integrity constraints are defined and evaluated on the states of objects and inter-relationships among objects.

In addition, the static features also include the representation of relations among different objects types. When several objects are related to each other, an update of one object will affect the states of other objects linked by relations. Knowledge can be captured into the descriptions of objects to automatically react on database events and object operations. Consistency and relations among objects are achieved through the definition of rules inside object types.

Dynamic Features

Dynamic features of database objects refer to their behavior and reactions to database events. Usually they are pieces of functions which are defined to describe certain actions of objects. In conventional database systems, only the structure of an application is described in the schemas. Programs or operations on database objects are separated from them. In object-oriented databases, an attempt is made to capture both attributes and behavior of objects in their types. Database systems can store both methods and object states with their schema definitions.

The comparison of the functionalities of data models is based on the features discussed above. The integrated systems may support the features either explicitly or implicitly in terms of whether they allow to specify them directly or not. A summary of the functionality of the integrated systems is given in Table 2.2.

2.4.3 Analysis of Transformation Alternatives

It is well known that relational database systems have a complete different structure for representing data in databases, compared with object-oriented ones. The transformation of knowledge from expert systems into databases thus results in the adoption
Chapter 2. Background of the Research

<table>
<thead>
<tr>
<th>Features</th>
<th>Clips+Oracle</th>
<th>Clips+Ontos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entities</td>
<td>Tables and Tuples</td>
<td>Classes and Objects</td>
</tr>
<tr>
<td>Attributes</td>
<td>Fields</td>
<td>Attributes</td>
</tr>
<tr>
<td>Relationships</td>
<td>Abstract Data Types</td>
<td></td>
</tr>
<tr>
<td>Methods</td>
<td>Member Functions</td>
<td></td>
</tr>
<tr>
<td>Rules</td>
<td>Rules</td>
<td></td>
</tr>
<tr>
<td>Constraints</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-Orientation</td>
<td>Object-Oriented</td>
<td></td>
</tr>
<tr>
<td>Database Operations</td>
<td>SQL</td>
<td>Ontos</td>
</tr>
<tr>
<td>Database Queries</td>
<td>SQL</td>
<td>Ontos</td>
</tr>
<tr>
<td>Programming Language</td>
<td>Implicit</td>
<td>Explicit</td>
</tr>
</tbody>
</table>

Table 2.2: Comparison and Summary of Features of Prototypical Systems

of totally different methods. This section discusses how knowledge is transformed when the integration of Clips and Oracle, and the integration of Clips and Ontos are concerned. Knowledge in Clips is basically divided into facts and rules. The details of mappings are not indicated, and only structural perspectives which influence the methods of integrations are considered. In the following we discuss the approaches of uploading and downloading for both facts and rules in the integrated systems.

Uploading and Downloading of Facts

Clips has a good rule-based programming style and provides several types of forms to represent facts. Facts in Clips can be declared by `deffacts, deftemplate` or other constructors. They basically have the structure that each field describes a certain attribute of facts, and fact names are given with respect to their meanings in applications. Clips facts are free-formed which means that facts are not required to have the same lengths and the same forms of organization. Fields are permitted to be missing when facts are declared.

Clips with Oracle

Facts in Clips are translated into relational tables according to two transformation rules:

1. a fact-list defined by either `deffacts` or `deftemplate` corresponds to one table.

The fact-list name is the table name and each field name of that relational table is assigned by an intermediate processor by using the one defined in deftemplate
or by generating it in terms of the longest facts in fact-list. The first field of the relational table is added by the processor to capture the keys of the table. They define an unique fact name for each individual fact.

2. when facts have various lengths inside a relational table, the longest facts are chosen to define the fields of the table and for those facts with undefined fields, zero is used to fill the fields with numeric types and NULL with scalar types. Apparently, this may cause the redundancy of data which is not the major issue concerned in this thesis.

3. A global table is defined which gives the general information about the transformation of facts. Information of fact tables related to an application and the types of facts are stored, e.g., the application involved and relationships to other facts and rules.

The extended Clips also provides a new function called persistent to query the Oracle database. This function is defined in Clips rule conditions with a set of arguments that correspond to the From and Where statements in SQL. The database operations should be taken in the rule actions which can be the insert, delete or select commands in SQL. Data in the database can directly be stored and accessed by rules in Clips. The SQL-like commands are simulated which in fact communicate with Oracle through the Pro*C interface.

Clips with Ontos

Transformation of facts between Clips and Ontos is much simpler than between Clips and Oracle. First, Ontos supports variable-length storage of data without losing efficiency. Data are classified into different classes according the abstract relationships among data. Second, Clips facts can be simply structured in Ontos. They can be stored in a generic class FactSet with type Set to specify the information about the fact-list name, which application is related, and the reference to a fact-list. A fact-list can be one object of a set which contains a list of facts. Individual facts are stored into class Facts which is defined according to the fact-lists. The general specification of them can only be as simple as having one field which stores facts as their text. More complicated specifications can be provided if the user requires. An alternative way is
that they can define the attributes in terms of fact fields. Facts stored in the generic class **FactSet** are only grouped by the object-identifiers rather than real contents of facts.

**Uploading and Downloading of Rules**

Rules stored in databases have more complex forms than facts because they do not follow any regular expressions. Clips defines rules by the constructor **defrule** which specifies the rule name, conditions and actions. The number of conditions and actions are varied and there is no restriction upon them.

Expert database systems should allow rules to be stored into the database which are treated as data first and later as queries. Therefore, rules are the text which are organized by the consideration of efficiency of querying the database. The solution for our prototypes are always to separate storage of rules with facts and different strategies are supported for the transformation of them.

**Clips with Oracle**

Rules in Oracle are constructed in a flat form [91]. Basically, two types of tables are provided: one is called table **Rules** and the other is called **Patterns**. The table **Rules** has fields for the rule identifier, rule name and the number of conditions and actions that are defined in each rule. Rule identifiers are distinguished from rule names in case that different applications may define rules with the same name. That is rule identifier provides an unique name for every rule in the database. Rule identifiers are generated by the processor when they are saved in Clips.

The table **Patterns** has the information of rule conditions and actions. Each condition and action are considered as one pattern and they are not decomposed in order to preserve the meaning and completeness of rules. The fields of the table **Patterns** are pattern identifier, rule name, signal of condition or action described for the current tuple, the order of conditions or actions and the contents of them. Note that each application can provide its own **Rules** and **Patterns** tables with the same structure defined above.

**Clips with Ontos**

Rules are stored in a similar way as those facts when Clips is integrated with Ontos.
Rules are classified by applications and regarded as first class objects. Instance rules are created when they are transferred from Clips into Ontos. A class named Rules is provided which stores each individual rule as one object. Global information is specified by another class called RuleSet and it refers to a set of rules as one instance. Instances attached to applications are retrieved when a task is performed, and all rules stored as these instances are loaded back to Clips. Clips runs those rules and any updates on either facts or rules need to be saved back to Ontos.

2.4.4 Discussion

The two prototypes are implemented as in a fashion analogous to previous loose coupling systems, with the intention achieving a common type of integrated architecture and gaining the insight into the integration problem. Among many technical difficulties, three issues are identified which must be resolved in the implementation of a realistic integrated system:

1. When the barrier of interfacing the two systems is too high, problems are usually solved by use of simplified transformation methods or by taking too much effort to develop new systems. It seems likely that an appropriate choice of the systems can remove some of the difficulties and obtain even better results. For example, the integration of Ontos and Clips makes the transformation very simple since Clips has a good interface with C.

2. We need to support more powerful representation rather than simply integrating two types of paradigms. According to the implementation, a simply integrated system cannot be used in complex and large applications as they are expected. Also, with the support of integrated system, a powerful interface is relatively easy to build.

3. As pointed out by [79], more advanced architectures are required for designing next generation of database systems. This concerns about how to choose existing systems, reduce communication burden and optimize the queries of knowledge in the database. We stress that object-oriented database systems and tight coupling architecture are more suitable for the research aimed by this thesis.
2.5 Introduction to the Knowledge-Oriented Data Model

It is claimed that a primary requirement for the next generation of the database systems is to provide more powerful data models to represent real-world semantics and behavior, which are normally associated with data in specific applications. The new functionality included in these models is characterized by richer types, functions, semantics constraints, knowledge and relationships between different components and objects into their data models. In the following, a data model called KOM (Knowledge-Oriented Model) is introduced. Its design principles and overview are given in this section.

2.5.1 Design Considerations

Some principles were followed in the design of the knowledge-oriented data model. The first consideration was to offer a suitable conceptual model which addresses the semantics of data of an enterprise completely, and a uniform language which represents the enterprise. An *enterprise* here means a piece of the real-world that is represented by a database[81]. It is not a static concept and contains the semantics and behavior that the real-world objects possess and perform. In the context of a database, this includes entities, relationships among entities, constraints to describe static and dynamic qualities of an enterprise and behavior of the objects of which the enterprise is composed. The critical aspect of KOM is to provide a uniform representation at the conceptual level by which various semantics and behavior of data are described and merged into one. It is different from previous work in the reduction of the distinction between data and knowledge, and semantics and behavior. It also simplifies the communications with the database by unifying DDL and DML into a single language.

The second concern of the KOM language is to integrate knowledge or rules with data specification. Rules are embedded into a type definition to support the description of different aspects of objects, especially the dynamic behavior of objects. The integrated rules can enhance object-oriented data models in several ways. Firstly, rules represent constraints to reflect updates on databases where integrity of objects is maintained. KOM rules expand the functionality by actively maintaining the semantic
dependencies among object’s states, constraints and operations. Unlike where they are placed in other systems, constraints in KOM lie within the object schemas and have a clean, declarative semantics which uses the same syntax as the main language. Secondly, rules can also be used to reason on data stored in a database and to derive new facts to provide more information about applications for users. KOM utilizes this feature to recover from failure by automatically taking actions. For example, once constraints are violated, KOM rules can be automatically invoked to modify other objects which are inter-dependent to recover the consistency. Moreover, rules can react on the result of sending or receiving messages. This behavior is necessary because operations can be decomposed into some essential ones and the dependency among them or the composite operations can be accomplished by rule firing. This obviously eliminates the overlap which usually happens when writing procedure codes. The final contribution of rules in KOM is to allow general programming statements to be defined in the actions of rules. This greatly enhances the capabilities of rules which are not permitted by other systems [56]. After all, since rules are embedded into type definitions, the inheritance of knowledge must be correctly assured.

One weakness of current object models is their lack of support for representing relationships and their semantics. Even though several prototype systems have been proposed, they limit themselves to describing the relationships between objects in tuples. In this case, each tuple corresponds to one relationship object and usually no properties can be defined. In addition, only binary relationships are represented and a mapping of 1:1, 1:m, m:1, or m:n must be declared when a relationship type is defined. These severely restrict current relationship models due to the fact that defining relationships in object-oriented paradigm depends on the theory developed for relational data models.

One design goal of the KOM relationship model was to provide a common theory for defining binary or n-ary relationships. This theory supports relationships as first-class objects that possess all the semantics held by ordinary objects as well as some extra ones. The instances of a relationship type are not merely tuples, but rather sets of tuples. The classification of these tuples into instances of a relationship type depends on application requirements. Thus, the semantics defined in a relationship type is
varied from one tuple to many tuples which are composed of one relationship instance. New semantics and some essential constructors have been introduced and can be used simply as if they operated on ordinary objects.

The last consideration in designing the KOM language was that it should support the concept of transactions in its language. The reason is that the incorporation of rules into data models requires the system to distinguish the points at which to check constraints or fire rules. The concept of transactions used for concurrency control in DBMSs needs to be modified so that it can be used at a conceptual level. Explicit transaction statements and implicitly generated ones are provided by the language. Execution semantics are given for single transactions, nested transactions and dynamically derived transactions. We will introduce these concepts through Chapter 2 and Chapter 3.

2.5.2 The Knowledge-Oriented Model

The motivation behind the development of knowledge-oriented data models is simply to capture even richer semantics and more meaningful knowledge of an application. The efforts made here are to allow the database designers and users to represent their applications in a manner such that their perceptions of the real world are much closer modeled by the database tools. KOM achieves this goal by providing an extended set of modeling constructs of object-oriented data models, a unified language for expressing and handling these construct and capabilities for data and knowledge management.

KOM supports the basic abstraction primitives required by object-oriented and semantic data models, including generalization, classification, aggregation and association. Generalization is an approach by which differences among similar objects are ignored and higher level types are formed. The types of objects hold basic properties for them and are organized in a is-a hierarchy. In the is-a type hierarchy, the types of descendants are more specific than those of ancestors, that is they have more properties than their ancestors.

The concept of classification refers to that objects are considered as instances of types. In KOM, an object can only have one type. The abstraction of aggregation serves as a high-level type which collects a set of objects from one type as an instance.
The grouped objects with an aggregation type can also have their own properties. It corresponds to a *part-of* relationship in semantic data models to refer that objects are components of an instance of an aggregation type.

*Association* abstracts relationships among objects as a high level type. This high level type is constructed from an ordered list of other types. The instances of a relationship type are sets of tuples and each element of a tuple corresponds to an object which is an instance of the type defined by the same field. The aim of this abstraction is to group related objects together and to represent the possible semantics and behavior of inter-object relationships.

KOM is based on two basic concepts: object and type. Types are the major vehicle for representing the semantics and behavior of objects. Basically, they are comprised of four components, namely attributes, constraints, methods and triggers. For example, attributes provide a description of object states, and constraints are defined to guarantee the integrity and consistency of objects. The definition and enforcement of constraints depend on the attribute values held by the instances of the type. Further, methods are procedural functions which operate on an object’s attributes and interface with outside applications. Constraints should be maintained all the time whenever objects are modified by their methods. Whenever attributes are updated, constraints are violated, or methods are invoked, triggers may be fired to take any necessary actions. Objects belonging to a type must possess all the qualifications which that type has.

At the type level, we distinguish ordinary object types from relationship types. A type defined for ordinary objects is called an *ordinary object type* or *ordinary type*, otherwise it is called a *relationship type*.
Chapter 3

The Knowledge-Oriented Model — KOM

3.1 Introduction

In this chapter, we introduce the Knowledge-Oriented Model (KOM) and give a detailed description of KOM types. A type in KOM represents a concept that delineates the structural and behavioral aspects of objects. In order to support the complete description of objects, possible semantics of constraints as well as the implied semantics of relationships of static and dynamic aspects of objects are also defined in the type definition. In fact KOM can be considered as an extension of object-oriented data models where the notions of data definition and manipulation follow functional data models like DAPLEX [94] and KDM [84]. A distinction is made between the concepts of class and type [9] [78]. A type is a specification of a set of qualities that may apply to objects, whereas a class is the set of all objects which possess those qualities. KOM supports type definitions with the description of object’s states, constraints, behavior and dynamic rules.

Through the whole chapter, an example in Figure 3.1 is used to demonstrate various definitions related to an object type. This example provides definitions of type Lecturer and Person and their semantics as specified by the attributes, constraints, methods and triggers. These concepts are explained in the following sections in this chapter.

This chapter is organized as follows. Section 3.2 describes briefly about the components which comprise a type specification in KOM. In Section 3.3 and Section 3.4,
type Person isa Object begin
  attributes
  Name: string;
  DateofBirth: Date;
  methods
  Person(name: string, birthday: Date);
  Age(birthday: Date)->number;
end;

type Lecturer isa Person begin
  attributes
  Employment: string;
  Salary: number;
  Position: string;
  constraints
  empCns: Employment = "employed" or Employment = "unemployed";
  salaryCns:
    Salary >= 30000 and Salary <= 76000
    and (Position = "Professor" implies foreach Lct: Lecturer
      where Lct.Position() <> "Professor"
      enforce Lct.Salary() < Salary)
    and (Position <> "Professor" implies foreach Lct: Lecturer
      where Lct.Position() = "Professor"
      enforce Lct.Salary() > Salary)
  except
    Position = "Volunteer" or Position = "Emeritus Staff";
  methods
  Lecturer(name: string, age: number, employment: string,
    salary: number, position: string);
  newSalary(salary: number)->void;
  newPosition(position: string)->void;
  triggers
  changePosition: if update(Salary) and (not salaryCns())
    after-action
    position: string;
    read(string, "new Position", position);
    newPosition(position);
    endaction; endrule;
  changeEmployment: if message(newPosition)
    after-action
    emp: string;
    read(string, "new State of Employment", emp);
    Employment(emp);
    endaction; endrule;
end;

Figure 3.1: Definitions of Type Person and Lecturer
constraints and triggers, which capture both static and dynamic semantics of objects, are introduced. Section 3.5 discusses inheritance issues in terms of KOM types.

3.2 Type Specification

Typically, an object-base is a collection of typed objects of varying granularity where each object belongs to a single type. A type is a description of the structural and behavioral properties and semantics of a group of objects which exhibit the same characteristics. The objects are called instances of the type and the collection of objects of a given type is named as a class of those objects.

A type in KOM denotes a set of descriptions of attributes, constraints, methods and triggers. Attributes encode the internal states of each of its objects, while the integrity of object states is maintained by their constraints. Methods implement the behavior of objects and provide an external interface to access an object’s states. Triggers are defined for changes of states or updates related to objects. Actions may be taken if changes are detected. The actions initiated by triggers help to maintain the correct semantics of objects and to derive new data increasingly by inference.

Constraints and triggers are totally different concepts in KOM. Constraints describe the allowable states for attributes which are implemented passively. Relationships and restrictions among object’s states are enforced. The specification of a constraint is intrinsic rather than extrinsic. In fact, constraints provide a mechanism through which the inter-relationships and intra-relationships are achieved. Unlike constraints, triggers are more concerned with the active aspects of objects. Object behavior, such as inter-object message passing, is accomplished through rule firing. The important point here is that triggers are used to implement the methods in a way that semantic links between object behavior or states are explicitly represented, while normal methods are simply declared and possible manipulations of objects bury the semantics in the program code.

A KOM type definition has the syntax defined in Table 3.1. The terms in angle brackets are non-terminal symbols and the terms with bold characters are terminal symbols. A optional phrase is enclosed by square brackets. The star notion is used to indicate the optional inclusion of a phrase zero or more times. Thus, a type is defined
Chapter 3. The Knowledge-Oriented Model — KOM

### Type Specification

A type specification is defined by a list of optional attributes, constraints, methods and triggers and each of the lists is identified by their keywords according to the syntax. It is very important to note that constraints and triggers also belong to the schema specification for an object type. They cannot be updated at run-time.

#### 3.2.1 Object States, Attributes and Representation

Attributes of a type describe possible states of objects of that type. In general, an attribute is specified by its type and a symbol which names that attribute. The symbol is a string which is composed of characters and numbers, and the attribute type can be any data types which are allowed inside the KOM system. KOM permits the following data types to be used:

- **number.** Both integers and real numbers can be declared as type `number`;

- **string.** Objects of `string` type are strings of characters.

- **boolean.** The `boolean` type always relates to logical operations, which can only return two values: `true` or `false`;

- **user-defined object types.** The user-defined object types here stand for any abstract data types which are declared as descendants of type `Object`;
lists. The aggregate type in KOM is indicated by keyword list-of. All the members of one aggregate type should have the same type. For example, list-of string indicates that an object is a collection of string objects.

We may now define the attributes more formally. An attribute of an object is defined on a domain and a name that links the object to that special domain. The domain of an attribute gives the type and possible range of values. A default value can be specified for any attribute. Its use is that a constructor can assign an attribute implicitly if no value of that attribute is given. These values can be changed during the operations at run-time. The syntax of attributes can be defined as Table 3.2.

An example of attribute specification can be seen in Figure 3.2. A person in the Person type is described by his name, age and parents, i.e., father and mother. The three attributes provide information about the structural aspects of objects of Person. When the attributes apply to each individual person, the states of that person or the attribute values are obtained. The Parents attribute of a person returns a reference to a set which contains both the father and mother of that person.

### 3.2.2 Object Behavior, Methods and Polymorphism

Object behavior is the term used to denote the collection of all methods, which abstractly describe what an object is capable of doing. Each method defines and models a particular behavior of the object. When a method is invoked by a particular object, it creates a message.
Type Person isa Object
begin
  attributes
    Name: string;
    Age: number;
    Parents: list-of Person;
  end;

Figure 3.2: Attributes of Type Person

Methods specified in a type definition encapsulate the behavior of objects of that type. They define the interfaces to those objects, and can be used to enforce the hiding of any internal structures and states held by the objects. Consequently, methods are the only available way to communicate with an object and access its states. KOM supports the above concepts as well as overloading functions and polymorphism. In this case, the specific method to be invoked will be dynamically determined at run-time.

KOM classifies methods into different categories which are used in terms of the object behavior to be described. Each category of method has its unique features which are described as follows:

- constructors: Constructors are invoked to create a new instance of an object for a type. A constructor method must be defined in the declaration of each type and share the name with that type;
• **get methods**: Get methods are defined for each attribute of objects to fetch the attribute values of objects;

• **set methods**: Set methods are used to assign values to attributes of objects;

• **general methods**: General methods are defined by users to implement other behavior of objects. They can be any functions required by applications.

A KOM method is defined by the category of the method, the method name, the possible parameters and the returned type (See Table 3.3). The return type of a method is **void** if no value is returned from the method. Methods of objects are only declared in the type definitions. The programs to implement them are written separately from the type specifications.

### 3.2.3 Object Semantics, Rules and Knowledge

As remarked by [19] [23] [57] [96], a great deal of the semantics that defines objects are represented by knowledge or rules in their type definitions. KOM supports two kinds of rules which either perform pure consistency checking or take actions with respect to various semantics of objects, such as enforcing constraints, deriving new information and maintaining dependencies among objects’ attributes and behavior. In fact, these rules provide the designer with a facility for defining implicit restrictions, specifications and additional inferred information regarding the nature of the objects.

In the first case, rules are constraints and they are expressed by a list of conditions. The explanation of this kind of rules is that when the conditions are satisfied, the current object states are acceptable and the result is true. KOM constraints allow not only the local constraints to be expressed, but global constraints as well. The local constraints here refer to those which are imposed on the objects of the same type. If constraints involve in more than one types of objects, the representation and implementation of global constraints are required. This is still an unexplored topic in object-oriented database systems. KOM classifies constraints into several categories and to certain extent, global constraints are expressed and enforced for objects.

Constraints are evaluated whenever any updates occur to objects in the database. If the constraints result in true, the operations become valid and the database is updated.
If constraints, on the other hand, are violated, the operations should be abandoned and there is no update to the database.

The second type of rules implements the trigger mechanism in KOM. They are production rules and are used for background reasoning and to represent "forced" or conceptual dependencies of objects. Like constraints, rules are used to reason within a context of one single object. When this is applied, they are typically used to add information or update a database that has a permanent aspect, or reaction to the states represented by constraints. For example, a rule could be used to add a new value to an object's state in order to keep a constraint true, even though some operation caused a temporary contradiction. In other words, triggers are very useful for dynamically maintaining correct semantics of objects.

3.3 Constraints: Ensuring Semantic Integrity of Objects

3.3.1 Introduction

Research on constraint representation in database or knowledge-based database systems has mainly focused on developing new types of semantic constraints inside type definitions [58] [68] [75]. As a result, systems like KL-ONE and Taxis are designed to capture certain types of constraints into the schemas and a fixed set of constraint constructs are provided explicitly [17] [71]. Basically, current database systems support constraints with numeric and limited symbolic operations and comparisons, as well as some kinds of time assertions and intra-class constraints [75] [107]. In certain extension, they also provide more expressive constraints, such as inter-relational or inter-object consistency involved in different object classes. However, stronger ties are required to maintain the semantics of database objects and their mutual consistency [68] due to the requirements from complex applications. For example, the head of the department has the right above lecturers on making decisions for students. This right should be reflected in both the student type and the lecturer type as some kind of constraints. The requirements are mainly concerned with specifying constraints for related objects which in general are buried in the external application programs.

From other aspects, constraints supported by most database systems are also rather
limited. First of all, they only provide static constraints which are only checked during object creation[85] or at the points where explicit requests for constraint checking are issued. No constraint is enforced subject to the arbitrary changes of objects.

Secondly, constraints are passive rather than active in database systems. This means that constraints can only be evaluated at the points where objects are modified regardless of the effect on other objects. The major weakness of present constraint evaluation is that it can not correctly enforce global constraints specified in some types that are semantically involved in other types of objects. Consider the example about students taking courses. When a particular course is deleted from the course class, there is no way to automatically detect the occurrence of the error in the objects of Student type, even though the dropped course is indicated. This results from the fact that the operation is taken in the type Course and the error happens in the type Student. In current database systems, there is no approach provided either specifying this type of constraints or preventing the above case from happening.

To solve these problems, two types of constraints are mainly proposed in KOM, which are called passive and active constraints. The passive constraints are similar to constraints in most systems, and are only imposed when explicit updates occur to objects in a type. They are only enforced by those objects which issue the updates to occur.

Active constraints, on the other hand, are more powerful than passive ones in the sense that they dynamically reflect the modification of objects. When updates occur, the system checks constraints for all the objects which are affected by these updates. This includes objects which are not directly involved in the updates. Any inconsistency among objects can thus be detected, guaranteeing the integrity of objects in the whole database.

In the following, we focus on introducing the syntax and semantics of both passive and active constraints. A constraint in a type definition can be either passive or active. The basic syntax of a constraint is shown in Table 3.4.
Chapter 3. The Knowledge-Oriented Model — KOM

3.3.2 Passive Constraints

Constraints are used to impose restrictions on the allowable object states or attribute values. In a type definition, constraints can be specified as a list of individual constraints, which can be distinguished by their names. Each individual constraint consists of a set of consistency rules and these rules provide a complete statement of certain aspects of object semantics. For example, a salary constraint `salaryCns` in Figure 3.1 expresses a lecturer should hold salary between $30,000 and $76,000 and all professors' salary should be greater than lecturers'. In this case, the constraint is an independent unit and can be processed or checked individually. This means that a function which contains a constraint can be called if the evaluation of that constraint is required. In general, a constraint in KOM can be represented by a name, followed by conditions which are linked by logical operators and semantic constructs. Between each individual constraints, the logical operator `and` is implied, which says that an object is valid on the condition that all the constraints are satisfied.

Except that the three logical operators: `and`, `or` and `not`, are provided, KOM supports another logical operator called `implies`, which links two conditions in a form `A implies B`. Logically, the statement equals to `not A or B` or `if A then B`. Here, `A` and `B` are lists of conditions. It states that if `A` is satisfied, then `B` must also be satisfied. For example, an object must have human features only because it refers to a human being. If the object is a bird, it should have bird features. This is a very useful logical operator, which not only simplifies the expression of conditions, but conceptually gives more powerful descriptions of constraints.

The KOM language provides some basic constructs to help specifying constraints. The syntax in Table 3.5 shows possible conditions which can be defined in KOM. In fact, these conditions can be distinguished into the following categories and all the notions with angle-brackets are non-terminals specified in Table 3.5.
Chapter 3. The Knowledge-Oriented Model — KOM

1. Basic restrictions. These explicitly declare the range and extension of attribute values, such as the possible constant values or the bounds that the attributes must satisfy. For example, a person’s age must be greater or equal to zero and less than 150.

2. Joint-object restrictions. Those constrain attribute values by means of logical operations definitions. The explicit manner in which the joint-object restrictions are specified, because the implicit relation between the attributes of type for-condition and the attributes of type for-declaration are specified.

Table 3.5: Syntax of Passive Constraints

| <passive-constraint>    | ::= | <constraint-name> : <condition-list>  
|                        |     | [except <condition-list>] ;           |
| <constraint-name>      | ::= | <identifier>                       |
| <condition>            | ::= | <expression> <comparison-operator>  
|                        |     | <single-expression>                  |
|                        |     | | foreach <for-condition>             |
|                        |     | | forsome <for-condition>             |
| <for-condition>        | ::= | <for-declaration>                   
|                        |     | [where <condition-list>]              
|                        |     | enforce <condition-list>              |
| <for-declaration>      | ::= | <variable> : <type-name>            
|                        |     | [in <set-name>]                      |
| <condition-list>       | ::= | <condition>                         
|                        |     | | not <condition>                     
|                        |     | | ( <condition> )                     
|                        |     | | <condition> <logical-operator>      
|                        |     | <condition-list>                     |
| <expression>           | ::= | <single-expression>                 
|                        |     | | <single-expression> <arithmetic-operator> 
|                        |     | <expression>                         |
| <single-expression>    | ::= | <value>                              
|                        |     | | <variable>                          |
| <logical-operator>     | ::= | or                                   
|                        |     | | and                                 
|                        |     | | implies                             |
| <arithmetic-operator>  | ::= | + | - | * | /                          
| <comparison-operator>  | ::= | < | > | = | > | = | <= | < > 

Table 3.5: Syntax of Passive Constraints
1. **Basic restrictions.** These explicitly declare the range and extension of attribute values, such as the possible constant values or the bounds that the attributes must satisfy. For example, a person's age must be greater or equal to zero and less than 150.

2. **Intra-object restrictions.** These constrain attribute values by means of logical and comparison operators within one object. These differ from the above definition because the implicit relationships among attributes of one object are specified. Thus, the change of some attribute values may require the change of others. For example, a man is retired if he is older than 65. Here, Age and Employment are the attributes of type Person and they are semantically relevant.

3. **Inter-object restrictions.** These express constraints among objects from several classes. This kind of constraints often relates attributes of the current type to attributes of other types. Attributes and methods in the current type are utilized to build the indirect reference to reach the targets which are required to restrict. Consider two object types: Course to describe the courses taken by students, and Student to provide general status information about students. A constraint may state that a student whose Major is "Computer Science" must take course CS13. Once his or her major is changed, the constraint is not required to hold anymore.

4. **Set-oriented restrictions:** These are defined for testing constraints for classes of objects, using existential or universal quantifiers. The universal quantifier is described by a foreach statement and the existential qualifier by a forsome statement. The foreach statement says that in a class or a set all the objects which satisfy the conditions in the where clause must also satisfy the conditions in the enforce clause. Similarly, the forsome statement says that if any object in a given class or a set satisfies the where conditions, at least one such object also satisfies the enforce conditions. An example of set-oriented restrictions is that all the professors in class Lecturer must earn more salary than ordinary lecturers do (See Figure 3.1);
5. **Exceptions**: Exceptions become relevant only after constraints are violated. The basic contribution of exceptions is to validate the new constraints for the objects in terms of the violated constraints. Unlike constraints, exceptions are evaluated only when the constraints related are violated. For example, in Figure 3.1, volunteers and emeritus professors do not require to satisfy the salary constraints. When the attribute values of a lecturer violate the salary constraint, we need to check whether the lecturer is a volunteer or an emeritus professor. Constraint `salaryCns` is still satisfied when the exception is true. The aims for adopting exceptions with constraints are that they provide clear semantics for exception cases and different an approach for evaluation of constraints.

The enforcement of constraints becomes complex when the inter-object dependencies take into account. The reason is that constraints have to be represented in all places where objects are restricted and may be re-evaluated if one of the objects is updated. The strategy of constraint validation involves in both the representation and implementation levels. In the following the representation problems are discussed; the implementation issues will be addressed in Chapter 5.

### 3.3.3 Active Constraints

Constraints in some object-oriented database systems, such as LAURE and ALICE [21][22][106], are declared independently from their schemas, and thus checked every time that any component of a schema is updated. As a result, they become global and are enforced for all objects in the database context. Compared with this, constraints specified inside type definitions have limitations that they can only react to updates of the current object’s states. For example, a constraint named as `prjSalary` in Figure 3.3.3 specifies that a lecturer who manages a project with fund over $300000 can earn higher salary than other lecturers. Apparently, this constraint links objects belonging to type `Lecturer` and type `Project`, and is affected by the update of attribute `Fund` defined in type `Project`. So far no systems can explicitly represent this type of constraints and enforce them correctly. An error may be hidden when the project’s fund is reduced but the lecturer still earns its old salary.
Active constraints are thus proposed for specifying constraints of which objects are causally related. The causally relationships among objects refer to that the modification of some of objects may affect the existence or states of other objects. Often, the two parties of causally related objects belong to different types. The objects which need to be constrained are called affected objects. The objects which cause constraints of other objects to be re-evaluated are dominant objects. In general, global constraints are specified in the type which defines the affected objects. They are checked when states of dominate objects are updated. If the two parties of objects are affected each other, constraints can be specified in either of the types or in both of them.

Active constraints enforce global constraints by representing the causal relationships in constraints and allowing them to be propagated to relevant objects. The syntax of active constraints is shown in Table 3.6. The affectedby clause lists the objects’ attributes and their types which affect the current object’s constraints. Once these attributes are modified, constraints of the current object need to be re-evaluated. A keyword affectedby implies that this constraint must be guaranteed in all cases.

When a dominate object propagates updates to the affected objects, it usually requires to iterate the whole classes of objects to find those which are affected by the current object. If references to those objects can be obtained from the current object, it can make the constraint evaluation much more efficient. The syntax for this purpose is provided by the foundby clause. The attribute name after keyword foundby gives the reference of the affected object.

An example of specification of a global constraint is shown in Figure 3.4. This constraint is defined in type Lecturer and affected by attribute Major defined in type Student. It says that a lecturer can supervise a student if they are in the same department. Once the student major is changed, the lecturer can not be his or her advisor anymore. The particular lecturer affected can be found directly through the attribute

prjSalary: forsome JP: Project in Projects
where JP.Fund>300000
enforce Salary >30000 and Salary<84000;

Figure 3.3: Example of Casual Related Constraints
Chapter 3. The Knowledge-Oriented Model — KOM

\[
\text{<active-constraint>} ::= \text{<constraint-name> affectedby <propagation-list>}: \\
\text{<condition-list> ;}
\]

\[
\text{<propagation-list>} ::= \text{<attribute-name> of <type-name> [foundby <attribute-name>] , <propagation-list>}]*
\]

Table 3.6: Syntax of Active Constraints

\[
\text{stdAdvisor affectedby Major of Student foundby Advisor:}
\]
\[
\text{forsome S: Student where S=Supervised enforce S.Major=Department;}
\]

Figure 3.4: Example of Active Constraints

value of \text{Advisor} of this student. If no action is taken to modify the value of attribute \text{Supervision} in type \text{Lecturer}, the evaluation of this constraint will automatically issue error message.

The enforcement of active constraints requires dynamic propagation of constraint checking to those objects affected by the current one. Since the update of the current object may result in constraint violation in other objects, the related objects' states also need to be modified in order to satisfy the constraints. Actions may be taken through rule firing before the system checks the constraints for objects (details of automatically maintenance object's semantics and enforcing constraints are introduced in the following section). As a result, the affected objects may cause other objects to be affected again. This process can be repeated until all relevant objects stored in the database are updated and satisfy their constraints.

Note that active and passive constraints need to be distinguished because of their different uses. The advantage of passive constraints is that they are local constraints and very efficient compared with active constraints. However, an active constraint can be accomplished by a list of passive constraints with slightly different semantics. Passive constraints which are equivalent to an active constraints, therefore, should be specified in all types which involve in that active constraint. Obviously, no matter
which components of the active constraints are modified, the corresponding passive constraints will be checked. From the representation point of view, this is awkward and does not provide a clear way to express the semantics of the constraints. Besides, the major difference between passive and active constraints is that all the objects joined in the passive constraints are equally important, whereas, in the type which defines the active constraints, the semantics of objects is presented but those objects making the updates determine the sequence of changes.

### 3.3.4 Constraint Evaluation

In general, constraints are evaluated in the order in which they are defined. When the order of specification is changed, the evaluation results become different. This is due to the fact that constraint violations may cause rules fired to be fired (see the following section). Since rules are commonly used to recover from constraint violation, we decided that the execution order would be to fire rules immediately after each constraint is checked.

The evaluation of active constraints may introduce non-determinism problems because they reflect the changes of other objects' states. Using the example in Figure 3.4, the update of a student major in type `Student` may result in the violation of the constraint defined in type `Lecturer`. If more than one active constraint is defined on the student major, the order in which they are enforced becomes an issue. The inadequate solution adopted in KOM is to put a time-tag for each object. Constraints are checked first for those objects created earlier. But they are all checked after the current object. No good solutions have been found at this stage.

### 3.4 Trigger Mechanism: Dynamic Maintenance of Object Semantics

#### 3.4.1 Motivation

Trigger mechanisms in database systems require rules to be activated by certain events. Usually, they restrict the conditions and the triggered actions to be single
operations in order to avoid triggering rules inside rules. Even though some systems tried to use the concept of transactions to describe the execution of rules, the technique is limited to considering one rule as one transaction [46]. No nested transactions are supported and the detailed semantics of rules is rarely explored except at the highest level. For example, when a rule action causes some events which may fire other rules, the existing execution models don’t investigate how these rules are to be fired, e.g. at which point that rules are exactly fired, but only point out existing problems and possible solutions [46] [56]. Thus, no precise execution model for the combination of nested rules and nested transactions is proposed. This greatly reduces the capability of full application of AI and database systems. Nevertheless, the ability to fire rules asynchronously and the combination of rules with transactions are considered as desirable extensions for database systems [56] [99].

Traditional trigger mechanisms are mainly used to achieve consistency checking. This is by far not sufficient, and only remains one possibility to assure the integrity of data. Triggers defined to repair inconsistency are usually initiated in response to update operations. Constraints are different in the sense that they are attached to object states rather than operations, and thus belong to object definitions. Whenever an object’s states are updated, constraints of objects are enforced. In addition to maintaining consistency, the trigger mechanism should capture more dynamic features and react to an object’s general behavior. However, with a few exceptions [56], existing trigger mechanisms do not regard message arrival as an event worthy of initiating rule firing.

Timing is another important semantic issue in designing trigger mechanisms. The existing systems, such as POSTGRES [98] and CPLEX [46], proposed that timing should be implemented by considering firing rules immediately or deferring them to the end of the same transaction or different transactions. This helps to gain more complex execution models for triggers.

Some suggestions have been made that rules should be defined outside object schemas [55] [56], so as to treat them as first-class objects, and to insert, delete and modify them independently [63]. However, we believe that rules are a part of the semantics of objects like constraints. They should be defined and stored with type
definitions. Like any other parts of schemas, they cannot be modified. In other words, rules are components of object schemas which describe dynamic aspects of objects.

KOM is designed as an active database, in the sense that it allows users to specify actions to be taken automatically, without user intervention, when certain conditions rise. Its trigger mechanism is proposed to overcome the above shortcomings and adopts the concepts of nested transactions and nested rules, associated with timing and events. Rules are very useful when inter-dependencies among an object's states and behavior, or among a group of objects are concerned. There might be a number of components or objects involved in one dependency, e.g., causing the update of one of the component or object necessitates the update of others. Also, trigger mechanism is a very efficient and semantically clear way to prevent constraint violations, a use that is unreasonably limited unless a full range of conditions and actions is permitted in the trigger specification.

In the past, rule bodies could in theory utilize program control structures as well as database operations, but was rarely realized in real systems [56]. Rules in KOM have very flexible forms and can include any number of conditions and actions. In particular, the actions of rules can be defined as an arbitrary list of programming language statements. This needs the co-existence of concepts of nested transactions and nested rules. Introduction of these concepts requires a major revision of the detailed semantics for the dynamic behavior of rule firing.

### 3.4.2 Syntax and Semantics of Rules

In KOM, rules are expressed by a set of rules. Each rule is specified by keyword `rule`, a rule-name and optional priority, followed by a list of conditions and actions. The priority of rules can range from 1 to 10 and must be integer. The greater the number is, the higher priority a rule has.

The conditions of a rule contain both events and some restrictions. An event is a run-time circumstance under which rule firing is considered. Restrictions are conditions which must hold before rule actions are taken. Actions are sequences of programming statements which will be executed when rules are fired. The syntax of trigger declaration can be seen in Table 3.7. In the following, detailed definitions and semantics of
Chapter 3. The Knowledge-Oriented Model — KOM

### Definition of Events

Generally, rules are triggered by events. This means that they result in the conditions of rules to be evaluated and actions to be taken. Two categories of event occurrences are considered in KOM: update operations and message catching. Update operations refer to changes of object states, i.e., changes of attribute values. An update is made whenever an attribute value is assigned, no matter whether the new value is the same as the old one or not. Message catching happens due to the invocation of methods for objects. When a message is received by an object, an event occurrence is established for that object.

Timing is very important issue related to KOM events. There are certain events that are established immediately upon the occurrence of the event and others which are delayed to the end of operations. As a result, KOM events can be classified into

---

```
<triggers> ::= [<declaration-statement>]* <rule> [<rule>] *
<rule> ::= rule <rule-name> [<priority>]
          if <trigger-condition-list>
          <trigger-action> * endrule;
<rule-name> ::= <identifier>
<trigger-condition> ::= <event>
<trigger-condition-list> ::= <trigger-condition>
                           | <condition-list>
<event> ::= update[ (<attribute-name> ) ]
          | message( <method-name> [ ( <parameter-list> ) ] )
<priority> ::= priority <decimal-number>
<trigger-action> ::= beforeaction <statement-list> endaction;
                    | afteraction [ deferred ]
                    <statement-list> endaction;
```

Table 3.7: Syntax of Triggers

rules are introduced.
Chapter 3. The Knowledge-Oriented Model — KOM

four types:

• **pre-update**: A pre-update event is established when an update operation is reached, but before it is executed;

• **post-update**: A post-update event is established just after the update operation is executed, but before the next operation is reached;

• **pre-message**: A pre-message event is established when the control goes to the method, but before it is really invoked;

• **post-message**: A post-message event is established when the control returns from the invocation of a method, but before next statement is executed.

Events are evaluated as boolean functions in KOM. The event functions return `true` only at the point where they are established. Except that, the functions return `false`. The event functions in the trigger syntax are expressed by keywords `update` or `message`. For convenience, events are also classified as **pre-events** and **post-events**. Obviously, pre-events refer to pre-update and pre-message events and post-events to post-update and post-message events.

**Rule Conditions**

KOM rule conditions can be arbitrary boolean expressions. As well as the syntactic elements usually permitted in KOM expressions, a special type of boolean functions is available, which are the event functions introduced in the previous subsection. More than one event functions are allowed in a rule’s conditions.

When an `update` function is used, its parameter can be either an attribute name or empty. The former tests if the current event is an update of the named attribute; the latter if it is any update event. Examples of this type of events can be seen in Figure 3.1. In the `changePosition` rule in type `Lecturer`, condition `update(salary)` tests if the `salary` is being updated.

The `message` function requires both the method name and the parameters and their types to be specified in order to identify an unique method. The parameter names can also be used in the rule conditions, as well as in the rule body. Take
rule changeEmployment in Figure 3.1 as an example. When a new position is assigned through method newPosition, the message is caught and new employment status needs to be confirmed.

Constraint functions explained in Section 4.4.4. can also be called in rule conditions, so that rule actions can be taken according to the evaluation of constraints to recover from failure before the system issues error messages. Therefore, constraints can not only be checked as a whole, but individually as well by calling the system-provided functions. For example, to check the salary constraint salaryCns in Figure 3.1, the method salaryCns() is called in rule changePosition's conditions.

Rule Actions

An action taken by a KOM rule is described by a list of statements which are those allowed to implement methods and functions (see Section 4.2). An action can be defined as either before-action or after-action, which correspond to pre-event and post-event, respectively. The before-action is taken before a method execution or an update are performed. In this case, the object which causes the event to occur still holds old values, i.e. values before updates or executions, and operations inside the actions are based on the old values rather than the new values. The update operations or method executions are performed after the actions are accomplished. There are cases in applications which require before-actions to be considered. For example, when an object is removed from a database, other objects which depend on this object should be removed first. Before-actions are useful to specify all semantically predefined operations attached to such events.

Alternatively, the after-actions refer to actions that are done after the events complete. The values used in an after-action are the new values after the update operation or the method execution. It is very often that users need to know both the old value and new value at the same time so that comparison can be made on the update. Previous systems usually provide two possible values for each attribute [38] [98]. One is the current value of the object attribute. Another is the old value of the attribute held before the most recent update. A weakness of this approach is that the database system must store both the new and old values and thus the storage requirements are doubled.
triggers
    oldSalary: number;
    .......
rule updateSalary
    if update(Salary)
    then beforeaction
        .......
        oldSalary:=Salary;
        .......
        endaction;
    afteraction
        .......
        if (Salary<oldSalary)
        then begin
            write("Warning: The salary is being decreased.");
            .......
            end;
        .......
        endaction;
    endrule;

Figure 3.5: Local Variables in Rule Definitions
tions should be delayed until there are no more transactions for this object. We name this type of rules as \textit{deferred} rules by using keyword \texttt{deferred} in front of rule actions. Obviously, deferred rules are always defined as having \textit{after-actions}. Rules activated inside a transaction belong to that transaction, thus are fired before constraint checking and the transaction commitment, as described in Section 4.3. When no transaction is explicitly defined, deferred rules are fired immediately after the events as ordinary post-event rules. Ordinary post-event rules are still fired inside the transaction at the place where they are activated.

Note that rules with both \textit{before-actions} and \textit{after-actions} can be considered as two rules, each of which is defined by either \textit{before-actions} or \textit{after-actions}, separately.

\section*{Rule Semantics}

Finally, we define the semantics of rules. Rule semantics includes two aspects. The first aspect is when rule conditions are evaluated. When an event begins, only rules with \textit{pre-events} have their conditions evaluated. After the event finishes, the conditions of rules with \textit{post-events} are evaluated.

The second aspect of rule semantics is about the order of evaluation of rule conditions. Basically, rules are evaluated according to the priorities that they have. Different from ordinary rule-based systems which use priority to decide which rule to be fired first, priority in KOM is used to determine the order of testing rules' conditions if more than one rules may be triggered by the same event. The higher priority a rule has, the sooner it is tested. Testing rule conditions follows the order listed below.

1. rules with the highest priority;
2. rules defined in the current type;
3. rules inherited from a supertype;
4. if rules have the same priority according to above definitions, then they follow the order of their specification.

Triggers are activated and fired in a total ordering. In fact, testing of trigger conditions and execution of trigger actions follow the same ordering. Once triggers are
Chapter 3. The Knowledge-Oriented Model — KOM

f() -> void
begin
    .......
    attribute-name:=value;
    or
    return-value:=method-name();
    or
    =>
    invoke return-value:=method-name();
    start transaction for this object;
    fire pre-event rules;
    update attribute-name:=value;
    or
    fire post-event rules;
    fire deferred-action rules;
    check constraints for this object;
end transaction;

Figure 3.6: An Event and its Transaction

activated, they are all fired, unless a specific deferred request is made. Deferred rules are fired by following the deferred order. First deferred rule is fired first.

3.4.3 Run-time Semantics

In order to provide precise and correct execution semantics for rules, run-time transactions are created whenever rules are fired. Rules co-exist with transactions in KOM. The reason is that rules are partially designed for recovering from constraint violation, and constraints can only be checked for objects before a transaction is committed, but after all possible update operations.

A transaction is created dynamically due to an event occurrence. It starts from the evaluation of conditions of the first pre-event rule and ends with the completion of actions of the last post-event rule. Constraints for the object which causes this event are evaluated before this transaction is committed. Figure 3.6 shows the correspondence between an event and its transaction.

A rule may itself raise events to initiate further actions. In this case, nested rules arise. Both rules and transactions thus become nested. The result of execution of the current rule can only be obtained after the sequence of rules activated by the current rule is fired. The control structure of nested rules is more complex than general forward or backward chaining in AI systems.

A very important issue relevant to nested rules is how to handle the termination of
rule firing. Consider a group of people and some of them are partners (see Figure 3.7). When some person catches disease, all his or her partners are needed to be found. When person A is recognized, through the rule firing person B and C are also retrieved. However, person A does not need to be identified again when person B and C are found. Otherwise, it will cause infinitive loop of rule firing.

A common solution to this problem is given by testing time-out at run-time, combined with action analysis to detect cycles [56]. KOM solves this problem by defining specific status for rules. Rules in KOM are identified as inactive and active. A rule is called active for a particular object if the condition is being evaluated, or the action is being executed, or the action has been deferred awaiting execution. Once a rule becomes active, its conditions can not be evaluated again, thus is impossible to be fired again. A rule can become inactive only after its conditions are not satisfied or it finishes its firing. Through the example in Figure 3.7, we can see that this approach can easily avoid firing person A’s rule by making it active. The advantage of this approach is that it uses the semantics of objects to prevent cycles of rule firing.

3.5 inheritance

3.5.1 Introduction

Inheritance is one of the key concepts supported by object-oriented systems [27] [49] [65]. It is introduced as a means of enhancing both productivity through reuse of descriptions and understandability through the classification techniques of speci-
fication and generalization. In the most general case, inheritance mechanisms may allow users to define new types by freely inheriting properties from existing object types and overwriting unnecessary properties [26] [90]. However, in practice this freedom of inheritance may result in violations of some basic criteria required by object-oriented systems and cause confusion about the semantics of objects [64]. For instance, freely overwriting properties of an existing type to gain a new type may lose the specialization/generalization relationship among the two types. Hence, there is tradeoff between reusing descriptions and overwriting existing properties.

In addition to the idea of generalization/specialization, a concept called conformity has been developed which is used to formulate the inheritance mechanism in some object-oriented database systems, such as Emerald [14]. Informally, it is said that a type $t$ conforms to a type $t'$ if $t$ provides all method declarations specified by $t'$, and for each individual method of $t'$, there is corresponding method in $t$. In particular, the method in $t'$ has the same number of arguments as that in $t$ and each of the types of the arguments conforms to the one in $t$ and the return type of the method in $t$ conforms to the corresponding one in $t'$ [86]. With conformity, subtyping and supertyping hierarchy can be dynamically built and changed. The *is-a* relationship among types is constructed according to the type definitions at that moment. In other words, the *is-a* relationship can be changed if a new type is declared and this type is not required to be a subtype of any other existing types. In fact, it can be some types' supertype if other types can conform to it. Apparently, this approach provides a more flexible and dynamically way of constructing type hierarchy.

KOM combines the concepts of subtype/supertype relationship and conformity in the design of its inheritance mechanism. By subtyping and supertyping, KOM allows types to be organized by their semantics and by conformity KOM restricts the properties inherited by a subtype from those in its supertypes to satisfy the conformity definition. In fact, conformity in KOM does not help the system to construct the inheritance mechanism, but to define and explain the semantics of subtyping relationships in terms of KOM types, especially for method inheritance and dynamic binding. In the following, the inheritance mechanism of KOM is introduced according to these two aspects.
Chapter 3. The Knowledge-Oriented Model — KOM

The basic inheritance system supported by KOM is the subtype/supertype hierarchy. A type $t$ is defined to be a *direct subtype* of type $t'$ if type $t$ is declared to have an *isa* relationship with type $t'$ in the type definition of $t$. We require the relation *direct subtype* to be acyclic. The relation *subtype* is the reflexive transitive closure of the relation *direct subtype*. The relation $t \geq t'$ means that $t$ is a subtype of $t'$. It is easy to see that $\geq$ is a partial order. (Note that, by definition, each type is a subtype of itself). Since the *isa* part of a type definition is compulsory, the set of all KOM types forms the set of nodes of a directed tree with root *Object*, as shown in Figure 3.8. The edges of the tree show the *direct subtype* relation. If $t \geq t'$, then the *distance* between $t$ and $t'$ in the directed tree is denoted by $\Delta(t', t)$. According to the rule of subtyping inheritance, instances of a subtype must be instances of its supertypes. This type of inheritance has an advantage that objects of a subtype can be referred to wherever instances of supertypes are specified. In other words, a subtype can reliably substitute for its supertypes in a system description.

When the substitution conditions are very restrictive, subtyping inheritance can be regarded as a limited refinement of supertypes and loses much of its advantage. Therefore, the KOM system only allows the subtyping inheritance used as a basic design principle:

**Design Principle:** If a type $t$ is a subtype of $t'$, all instances of type $t$ are also instances of type $t'$. 

![Figure 3.8: KOM Type Hierarchy](image)
Some essential extensions adopted from conformity have been made so that properties of supertypes can be overwritten at the subtype levels. Before defining the rules for inheritance and conformity, we classify the properties of a type into two categories: inherited and new defined properties. The inherited properties are those defined at supertypes’ level but inherited or maybe overwritten at the current level. The new defined properties are new definitions which are only declared in the current type.

The properties of a type $t$ can be described as a four component tuple: $t = (a; c; m; r;)$, which stands for attributes, constraints, methods and triggers. Each of them can be represented by a set of definitions, such as a set of attributes $a = \{a_i : t_i \mid 1 \leq i \leq n\}$, with $a_i$ being a name and $t_i$ a type. Since the concepts of attributes, constraints, methods and triggers provide more flexible definitions and richer semantics, inheritance of these properties may lead to an unexpected change in the semantics of operations. In other words, the inheritance mechanism, while powerful, can be dangerous if misused. In the following, the issues of inheritance are examined more closely, and more precise definitions of inheritance are given separately for attributes, constraints, methods and triggers.

The inheritance of KOM types obeys the rules of Attribute Inheritance, Constraint Inheritance, Method Inheritance and Trigger Inheritance introduced below. Types defined against these rules either produce the wrong results or are rejected by the KOM system. Inheritance rules differ from inference rules in the sense that they are embedded in the implementation of the KOM system, but the latter can be specified at user’s level and enforced at run-time. In other words, the inference engine in KOM only manages user-specified rules rather than conformity rules.

In KOM, we do not consider multiple inheritance because the way to merge the inherited properties from multiple supertypes is not clear and differs from case to case. The precise semantics of multiple inheritance will be investigated in the future.

### 3.5.2 Inheritance of Attributes

Suppose we have two object types $t$ and $t'$, and $t \geq t'$. The attributes of $t$ are those attributes declared in the type definition for $t'$, together with those attributes of $t'$ which have different names. If an attribute $a$ declared in the type definition of $t$
has the same name as an attribute $a'$ of $t'$, then $a'$ is not inherited from $t'$ by $t$ but rather overridden. In keeping with our design philosophy that instances of type $t$ are also instances of type $t'$, we impose the restriction that $a$ has a type which is a subtype of the type of $a'$. The inheritance of attributes follows two rules defined below:

Suppose the attributes of $t'$ are \{ $a_i' : t_i' | 1 \leq i \leq n'$ \}, the attributes declared in the type definition for $t$ are \{ $a_j : t_j | 1 \leq j \leq n$ \}, and $t \geq t'$.

**Attribute Conformity:** We impose the conformity of inherited attributes as:

If $a_j = a_i'$ then $t_j > t_i'$ ($1 \leq i \leq n'$, $1 \leq j \leq n$).

**Attribute Inheritance:** The attributes of $t$ are

\{ $a_j : t_j | 1 \leq j \leq n$ \} $\cup$ \{ $a_i' : t_i' | 1 \leq i \leq m, a_i' \neq a_j$ for $1 \leq j \leq n$ \}.

For example, type PostGraduate and SpecializedCourse can be defined as subtypes of type Student and Course, respectively. When an attribute in type Student is declared as takeCourse of type Course, it can be overridden in subtype PostGraduate as takeCourse of type SpecializedCourse. The new attribute takeCourse defined in type PostGraduate conforms to the old one in type Student by using the same attribute name. However, because the new type is a subtype of the old type, the new attribute is meaningful wherever the old was.

### 3.5.3 Inheritance of Constraints

The case of inheriting constraints is slightly more complex. The reason is that the same attribute value can be constrained in both the parent types and the descendant types. For instance, when a constraint is defined for type Person's attribute Age, it may specify that a person's age should range from 0 to 100. After a subtype Employee of Person is defined, the constraint may be restricted to the range of 20 to 65. Of course, the latter constraint is more specific than the former one and the correct semantics of the Employee is also preserved. In order to guarantee that the inherited constraints in a subtype are more specific than those in supertypes, a constraint declared in a subtype does not override one of the same name in a supertype but instead is joined to it with a logical and.
We now formulate a precise definition of constraint inheritance. The declaration of constraints in a type definition $t'$ consist of lists of constraints, i.e., $c' = (c'_i : p'_i | 1 \leq i \leq n')$, where $c'_i$ is a constraint name and $p'_i$ predicates for $1 \leq i \leq n'$. This can also be applied to type $t$, $c = (c_j : p_j | 1 \leq j \leq n)$. The inheritance of constraints follows the rule defined below:

**Constraint Inheritance:** Suppose the constraints of $t'$ are 

$$c'_i : p'_i | 1 \leq i \leq n'$$

then the constraints declared in the type definition for $t$ are 

$$c_j : p_j | 1 \leq j \leq n$$

If $t \geq t'$, then the constraints for $t$ are 

$$c'_i : p'_i | 1 \leq i \leq n, \quad c'_i \neq c_j \text{ for } 1 \leq j \leq n \cup (c'_i : p'_i \land p_j | 1 \leq i \leq m, 1 \leq j \leq n, \quad c'_i = c_j \cup (c_j : p_j | 1 \leq j \leq n, \quad c_j \neq c'_i \text{ for } 1 \leq i \leq n').$$

In terms of this inheritance rule, constraints are checked according to the following orders:

1. Constraints defined in the supertype but not redeclared in the current type are checked according to the order that they are declared in the supertype.

2. Constraints defined in both the supertype and the current type are checked according to the order that they are declared in the current type, regardless of the order that they are defined in the supertype.

3. Constraints only defined in the current type are checked according to the order that they are declared in the current type.

In contrast to the situation with constraints, it is not necessary to impose a separate conformity rule. This is because the use of the logical conjunction in the inheritance rule makes conformity automatic.

### 3.5.4 Inheritance of Methods

The inheritance of methods allows users to define either new methods for a subtype or override existing ones defined in its supertypes. To override a method, one needs to rewrite the declaration of a method but keep the implementation body of it, or to keep
the declaration of the method but re-implement the body of it. In the former case, the definitions of methods require to follow the rule of conformity. In the latter case, dynamic binding of methods is achieved when a method is called. The real program of the method to execute is determined at run-time according the reference referred to the method. The issue of dynamic binding will be discussed in Section 3.5.5.

Here, the concept of conformity is defined for restricting the inherited methods from one type to another. Informally, a method \( m \) in a type \( t \) conforms a method \( m' \) in a type \( t' \), if \( t \geq t' \), and

1. \( m \) and \( m' \) have the same method name and the same number of arguments;
2. each argument of \( m \) conforms to the corresponding one of \( m' \); and
3. the return type of \( m' \) conforms to that of \( m \).

Basically, the inheritance of methods follows two rules defined below:

Suppose types \( t \geq t' \) and the methods of \( t' \) are \( \{ m_i' : x_i' \rightarrow y_i' : b_i' \mid 1 \leq i \leq n' \} \) and the methods declared in the type definition for \( t \) are \( \{ m_j : x_j \rightarrow y_j : b_j \mid 1 \leq j \leq n \} \), where \( m_i' \) and \( m_j \) are method names, \( x_i' \) and \( x_j \) are argument lists of method \( m_i' \) and \( m_j \), \( y_i' \) and \( y_j \) are the return types of method \( m_i' \) and \( m_j \), and \( b_i' \) and \( b_j \) are the program bodies of method \( m_i' \) and \( m_j \), respectively, for each \( 1 \leq i \leq n' \) and \( 1 \leq j \leq n \).

**Method Conformity:** We impose the conformity of methods as

for \( 1 \leq i \leq n' \), \( 1 \leq j \leq n \),

1) \( m_i' = m_j \),

2) when we define the argument lists as

\[
x_i' = \{x_{ik}' : t_{ik}' \mid 1 \leq k \leq m' \} \quad \text{and} \quad x_j = \{x_{jl} : t_{jl} \mid 1 \leq l \leq m \},
\]

they satisfy the following conditions:

\[
x_{ik}' = x_{jl} \quad \text{and} \quad t_{ik}' \leq t_{jl} \quad \text{and} \quad m' = m;
\]

then \( y_i' \geq y_j \).
Chapter 3. The Knowledge-Oriented Model — KOM

**Method Inheritance:** In addition to the above definition, we define the overwriting of method bodies as if \( m_j \) conforms to \( m'_j \), then \( b'_i = b_j \) or \( b'_i \neq b_j \) for \( 1 \leq i \leq n', 1 \leq j \leq n \).

The methods of \( t \) are \( \{ m_j(x_j) \rightarrow y_j : b_j | 1 \leq j \leq n \} \cup \{ m'_i(x'_i) \rightarrow y'_i : b'_i | 1 \leq i \leq n', m'_i \neq m_j \land x'_i \neq x_j \text{ for } 1 \leq j \leq n \} \).

For example, the method `display()` of type `Lecturer` has different implementation body but the same declaration as the one defined in type `Person`. Also, the method `getAllCourses()`->`Course` in type `Student` can be conformed in the subtype `PostGraduate` with the return type `SpecializedCourse`.

### 3.5.5 Inheritance of Triggers

Compared with constraint inheritance, inheritance of triggers is quite simple. It is very possible for users to define new conditions or new actions on the old rules which are inherited from supertypes of the current type. Similar to method inheritance, the inheritance of triggers requires that an overridden rule in a subtype keeps the same rule name of the one in its supertypes but with redefined rule body. For example, when type `Professor` is defined as a subtype of type `Lecturer` in Figure 3.1, the inherited rule `changePosition` needs to be modified by re-input the salary rather than giving a new position, since the objects of type `Professor` are required to have position `Professor`. In fact, the rule `changePosition` has new actions which read the salary again.

No conformity rule is required for trigger inheritance. The precise definition of trigger inheritance is:

**Trigger Inheritance:** Suppose types \( t \geq t' \) and the triggers of \( t' \) are \( \{ t'_i : r'_i | 1 \leq i \leq n' \} \) and the triggers declared in the type definition for \( t \) are \( \{ t_j : r_j | 1 \leq j \leq n \} \).

The triggers of \( t \) are

\[ \{ t_j : r_j | 1 \leq j \leq n \} \cup \{ t'_i : r'_i | 1 \leq i \leq n', t'_i \neq t_j \text{ for } 1 \leq j \leq n \} \]

The point to remember is that both message receiving or constraint testing in the trigger conditions depend on run-time binding. More detail of dynamic binding can be
Chapter 3. *The Knowledge-Oriented Model — KOM*

3.5.6 Dynamic Binding

Since at some extent we allow the four components of a type definition to be overridden at subtype levels, dynamic binding at run-time is needed when certain properties attached to variables can not be determined at compile time. For example, when attribute `takeCourses` of type `Student` is redefined in subtype `PostGraduate`, the statement in Figure 3.9 should return courses defined by type `SpecializedCourse` rather than type `Course`.

Dynamic binding usually happens when a variable is defined as having the type of a supertype but binds with its subtypes at run-time. Because some of the properties defined in supertypes are overridden in subtypes, the correct properties which should be referred by the variables must be from the subtypes rather than the supertypes. In fact, the KOM system binds a variable not according to what the pointer reference is but rather according to the type of the object.

Dynamic binding of methods is more complex due to many methods that can be matched at run-time. The methods which can match the same message are regarded as *compatible* with that message. Usually, these methods are those defined in the same type and having the same name but with different argument types or different program bodies. For example, the two methods in Figure 3.10 are compatible by defining the same method name but different argument lists. The precise definition of compatible methods is...
Chapter 3. The Knowledge-Oriented Model — KOM

Definition A declaration of a method can be described as \( m(x_1 : t_1, x_2 : t_2, \ldots, x_n : t_n) \), where \( x_i \) is a parameter and \( t_i \) is the type of \( x_i \) for \( 1 \leq i \leq n \). Any number of these methods can be visible for the same name. When a call is issued at run-time: \( m(x' : t'_1, x'_2 : t'_2, \ldots, x'_n : t'_n) \), the declaration and call are compatible if \( t'_i \geq t_i \) for \( 1 \leq i \leq n \).

Since the arguments of the methods \( f(x,y) \) and \( f(x',y') \) in Figure 3.10 are subtypes and supertypes, a message with arguments of the supertype makes the two procedures matched. The KOM system chooses the first procedure to invoke according to the least distance between methods and the message. This motivates the next definition:

Definition The distance between a declaration and a call is defined by a vector \((\triangle(t_1, t'_1), \triangle(t_2, t'_2), \ldots, \triangle(t_n, t'_n))\).

We can now give the rule for dynamic binding of calls to declarations.

Definition 3. A function call binds to the visible compatible declaration to which its distance is least in lexicographic order.

Rules for dynamic binding also apply to methods used in constraints and triggers and free functions. In addition, methods called outside type definitions follow dynamic binding through checking of navigation paths. That is, only methods at least with the same name and navigation path as the call can possibly be compatible.

3.6 Comparison

In this section, we compare the features of KOM with some related systems introduced in [11] [30] [41]. First of all, ODE is very similar to the way that KOM is defined. For example, both KOM and ODE support constraints and triggers and data (or attributes), methods, constraints and triggers as basic components of an object type. Constraints in both systems are used to maintain the consistency of the database and triggers to monitor the events happening in the database and react by executing some operations. However, there are some major differences between the two systems in terms of how they specify and implement the constraints and triggers.
Chapter 3. The Knowledge-Oriented Model — KOM

Two Methods are defined as follow:

\[
f(x : t, y : t') -> \text{void}
\]

\[
\text{begin}
\]

\[
\ldots\ldots
\]

\[
\text{end;}
\]

\[
f(x' : t', y' : t) -> \text{void;}
\]

\[
\text{begin}
\]

\[
\ldots\ldots
\]

\[
\text{end;}
\]

Here \( t \) is a subtype of \( t' \). A message is sent as:

\[
g() -> \text{void}
\]

\[
\text{begin}
\]

\[
a : t;
\]

\[
\ldots\ldots
\]

\[
f(a, a);
\]

\[
\ldots\ldots
\]

\[
\text{end;}
\]

Then the first method is executed.

Figure 3.10: The Order of Dynamic Binding
First, only passive constraints are supported in ODE. When a constraint is involved in more than one object, they have to be specified at all places where they need to be ensured. Otherwise, consistency is not guaranteed. In KOM, this is achieved by active constraints.

When an ODE constraint is defined as an inter-object constraint, they are automatically deferred till the end of the transaction. Only intra-object constraints are checked immediately after the attributes are updated. In the first case, even if the constraint is violated at the beginning of the transaction, the execution still continues to the end. However, in KOM users are allowed to define conceptual level transactions (see Section 4.2). Conceptual level transactions ensure that violation of inter-object constraints are found and recovered immediately after a sequence of updates. Triggers or rescue statements (See Section 4.2) can be executed just before the transaction aborts or commits. Nested transactions are also supported.

The constraint checking in ODE is only done when public functions of an object are called. Several shortcomings of this approach are: constraints may be checked even when the object’s states are not updated in the public function, or inconsistency may potentially exist when private functions change the object’s values or the values which violate constraints are passed to public variables and public variables are not restricted and checked. KOM checks its constraints whenever the relevant objects are updated unless transactions are explicitly defined. A disadvantage of this approach is that it makes the constraint evaluation very expensive, but it guarantees the consistency once they are defined.

A feature of ODE triggers is that they are not part of the transaction that fires the triggers and also do not abort the transaction even if the trigger’s actions fail. This is not supported in KOM since triggers can be used to detect constraint violation and to fix it. Naturally, they are part of the transaction. Extension could be made to triggers which are defined for message catching. Triggers and transactions can belong to separate transactions when they are not defined for constraint checking.

ODE only implicitly support one type of event, which is the update of object states. This makes the uses of triggers rather limited. KOM certainly is more powerful in the sense that they support three major events: updates of objects, message catching
and violation constraints. In the actions of triggers, the operations can be written by arbitrary statements provided by the language. Besides, the conditions of triggers can be composite one and they can link different kinds of events and some other conditions by logical operators.

Triggers must be explicitly activated in a method body of ODE. When they are fired, the ordering of execution is not explicitly defined. In KOM, triggers only depend on the event occurrences. When an event occurs, triggers are fired no matter where the execution is. The semantics and execution order of triggers are defined precisely.

A system discussed in [41] also introduced a model for triggers and transactions. Triggers defined in the paper basically reflect the execution of methods. Once a method is executed, it may trigger some actions and these actions are only executed when users want them to be. Thus, in this model, events are related to methods and conditions of trigger firing are defined by time. The time can be the beginning of a method execution or after the completion of the execution. The difference between this system and KOM is that it allows the events specified for different objects. In other words, the execution of other objects' methods may cause the trigger defined in the current object fired. However, KOM only allows triggers to react on the execution of the current object's methods.

Like ODE, the system in [41] does not support composite conditions or as many events as KOM does. Neither of the systems provide an approach to eliminate or reduce the chance of infinite looping of trigger firing. In KOM, an approach is proposed to avoid firing rules when they are active. This approach is easy to implement.

In summary, all three systems are designed on the basis of active databases, object-orientation and nested transactions. They all support triggers, but in a slightly different way. Constraints are not supported in the second system, which makes the structure of trigger firing simpler and easier to implement. Further comparison on the transaction issue is discussed in Section 4.2.5.
Chapter 4

General Functionality of the KOM Language

Until now, we have considered that types are a set of declarative statements through which the semantics of objects are represented. In the following, we introduce the operations and functions which are supported by KOM. These operations and functions can be used to write programs or build real applications.

In Section 4.2, general programming statements and some database operations are described. Section 4.3 defines the concepts of transactions and rescues are defined. In Section 4.4, KOM functions and methods are constructed by using the statements provided in Section 4.2. After this, a query language is proposed in Section 4.5 which operates on objects of certain classes. Iterator statements are defined which retrieve objects from classes according to the conditions specified by the user.

4.1 Programming Statements

4.1.1 General Statements

KOM adopts features from object-oriented programming languages and allows general programming statements and operations to be performed [44] [65]. These include statements for input and output, assignment, if-then-else, while, for-iterator, compound, break, return, undo, abort, transaction and rescue. The syntax of the statements is listed in Table 4.1.

The read and write statements get or put data from or to the standard input/output device. The <string> in the syntax is a phrase which describes the data to be read or
Chapter 4. General Functionality of the KOM Language

Table 4.1: Syntax of General Programming Statements

| <statement> | ::=  | <single-statement> |
|            |     | | <compound> |
| <single-statement> | ::=  | <input>  | | <output>  | | <assignment> |
|          |     | | | | | if-then-else>  | | <while>  | | <for-iterator> |
|          |     | | | | | | <return>  | | <break>  | | <undo>  | | <abort> |
|          |     | | | | | | <transaction>  | | <rescue> |
| <statement-list> | ::=  | <single-statement> [ <single-statement> ]* |
| <compound> | ::=  | begin <statement-list> end; |
| <input> | ::=  | read(<type>, <string>, <variable>); |
| <output> | ::=  | write( <string> , <expression> ); |
| <assignment> | ::=  | <identifier> := <expression> ; |
| <if-then-else> | ::=  | if <condition> then <statement> |
|          |     | | [ else <statement> ] |
| <while> | ::=  | while <condition> do <statement> |
| <for-iterator> | ::=  | foreach <for-declaration> |
|          |     | | where <condition> <statement> |
|          |     | | | forsome <for-declaration> |
|          |     | | | where <condition> <statement> |
| <return> | ::=  | return [ <expression> ] ; |
| <break> | ::=  | break; |
| <undo> | ::=  | undo; |
| <abort> | ::=  | abort; |
| <variable> | ::=  | <identifier>[.<identifier>]* |
|          |     | | this[.<identifier>]* |

An indirect reference is one that a variable is defined to refer to a component object. Usually, it starts from a known object and ends with the target. The components in between are attributes or variables from two or more objects which have some implicit associations according to their type definitions. The "dot notation" is adapted to chain the components of an indirect reference together. The last element of the path must be a variable.
Chapter 4. General Functionality of the KOM Language

written. Assignment statements have a PASCAL-like syntax; the value of <expression> is assigned to the <variable>. The “if-then-else” statement specifies that if the conditions are satisfied, execute the then statement, otherwise execute the else statement. The while statement states that statements will be executed when conditions in the while clause are satisfied. It continues until the conditions become false.

The “for-iterator” statement specifies that for each or one of the objects in a class or a set, if the conditions are satisfied, then execute the statements. We will give the detail in Section 4.5.

Compound statements consist of a list of general statements defined above bracketed by begin and end. They act as single statements which can be used in anywhere where a programming statement above is defined. The return statement terminates a function execution and returns control to the next statement where the function is invoked. If a return value is specified, its type must be the same as the return type defined for that function. The break statement is usually used inside loop statements and cause control to return the higher level statements.

Transaction statements are defined to delay the checking of constraints for specified objects. Inside a transaction body, a list of KOM statements can be written to perform the updates. If the commitment of the transaction may cause the constraints to be violated, a rescue statement can also be defined to validate the conflict before the commitment. In fact, the rescue statements are defined to help transactions and functions to recover from programming or data errors before the system makes any decisions. One of the recovery actions is abort to abort the execution and return control back to the system. Another possible action is undo which abandons all the updates performed by the current transaction. Decisions can be made in a rescue statement either to stop the execution or commit the transaction to continue the execution. These statements will be explained in later sections.

An indirect reference is used when a variable is defined to refer to a composite object. Usually, it starts from a known object and ends to the target. The components in between are attributes or methods from two or more objects which have some logical associations according to their type definitions. The “dot notation” is adopted to chain the components of an indirect reference together. The last element of the path must
have the same type as the variable to be assigned. Note that the method name can be simplified without specifying the parameter list and the type of a method refers to its return type.

The keyword this is provided to refer the current object itself whenever it is required. In general, the attribute or the method name of the current object can be used directly inside a type specification.

### 4.1.2 Database Operations

Two database operations are provided in KOM to allow users to communicate with the database (See Table 4.2). The open database command is used to open the user's database with the name given by `<database-name>`. The close command closes the database currently opened. At a time, only one database can be opened.

In addition, three primitives are provided by which users can create, find and delete an object. The syntax of these primitives is defined in Table 4.3. When an instance is required to create, the create primitive calls a constructor for that object from its type and passes the attribute values through the parameter list to the instance. The constructor returns a reference of the object to a variable. The find primitive returns a reference to the object with a given type and name, or null if there is no such object. Primitive delete simply removes the specified object from the database. If the object that is being deleted participates in relationships in other objects, those relationships are required to re-assign so as to maintain the correct semantics.

**Table 4.2: Syntax of Database Operations**

```plaintext
<db-operation> ::= openKOMdb( <database-name> ) ;
                  | closeKOMdb() ;

<database-name> ::= <identifier>
```

A useful feature of TAIPE is that it designs transactions explicitly as a separate
Chapter 4. General Functionality of the KOM Language

4.2 Transactions

4.2.1 Motivation

Transactions are a very important concept which is introduced by database systems to tackle concurrent processes of database operations. This concept is featured by its atomicity and causes a database operation once performed, it only allows that a single execution of the program either occurs in its entirety or does not occur at all upon certain database items [104]. Items are units of data and must be locked before they are accessed. A transaction is committed after it is finished or aborted if it fails. The usefulness of transactions is the way that they forbid other operations taking place on the same data during the execution.

Some of database systems use the concept of transactions to support their data model designs at conceptual levels. For example, POSTGRES as well as some other systems which provide trigger mechanisms explain the semantics of rules by means of transactions. Since trigger mechanisms in most cases are utilized to maintain the integrity of data, constraints can only be checked after actions have been taken by rule firing. The concept of transactions here helps to restrict the places where rules can be fired or constraints can be checked. As a result, POSTGRES proposes its future rule activation policies by immediately upon the occurrence in the same or different transactions or deferred to the end of the same or different transactions [98]. In other systems such as CPLEX [46], transactions are described as a set of modifications accompanied with some triggers. Here, transactions are used to define the points to evaluate conditions or to take actions upon triggers.

A useful feature of TAXIS is that it designs transactions explicitly at a conceptual
Chapter 4. General Functionality of the KOM Language

level [71]. A transaction in T AXIS corresponds to a method with pre-requisites. When the pre-requisites are satisfied, operations defined in a transaction are performed and results are generated. Since the constraints of data are specified inside transactions rather than attached to data themselves, the T AXIS users need to define handlers for constraint violation and exception cases. In this case, constraint checking and action taking are directly associated with a transaction, and thus no complicated issues are arisen such as nested transactions or precise points of constraint evaluation.

Constraints defined in KOM are more complex than those allowed in T AXIS. Updates performed from any methods can result in constraints to be evaluated, specially in many cases that updates can not be accomplished in one single operation. Suppose we wish to perform the two operations $x := x - 1; y := y + 1$ and there is a constraint $x + y = 10$. Clearly we should delay the constraint checking until after both operations are completed, rather than doing it after each. Therefore, the two statements $y := y + 1$ and $x := x - 1$ constitute one atomic meaningful operation and we define them as one transaction. The major motivation for introducing transactions into the KOM language is to guarantee that constraints are evaluated at the correct points. A transaction can not be interrupted during the execution. Constraints are only regarded as violated if they are not satisfied after transactions are completed.

Transaction statements can be used in KOM to achieve the concept of atomicity as they are considered at the implementation levels. In the following, we elaborate the concept of transactions and introduce their syntax in KOM. More complex situations, such as nested or dynamically defined transactions, are concerned. The semantics of both simple and complex transactions is explained, followed by examples.

4.2.2 Transaction Syntax

As specified in Table 4.4, the syntax of a transaction consists of a list of statements enclosed by transaction brackets: transaction and endtransaction. The variable list in the header names the objects for which constraint checking should be delayed until the end of the transaction. The statement list inside the transaction can be any statements allowed in KOM, including transactions. Therefore, nesting of transactions is permitted, through either static declaration of one transaction inside another.
Chapter 4. General Functionality of the KOM Language

4.1.5 Execution Semantics of Transactions

Lecturer::newSalary(salary: int)->void
begin

   transaction this do
   Salary(salary);
   if (salaryCns())
   then begin
      /* 1. find the professors whose salaries are lower than
         the current lecturer's salary;
      2. update the professors' salaries. */
      end;
   end;
endtransaction;

Figure 4.1: Salary Transaction

or dynamically generation due to transactions executed by methods invoked during
transactions or firing triggers.

An example of the transaction defined for the Lecturer type in Figure 3.1 is shown
in Figure 4.1. The function newSalary(salary) assigns a new salary to a lecturer.
From Figure 3.1 we know that the rule changePosition is fired to ask the user to pro-
vide new position for the lecturer if the salary constraint is violated. If the position is
correct, the user may want to increase the professor's salary to keep the constraint sat-
sified. A transaction thus is defined in the method which accomplishes the assignment
of the lecturer's salary and checking of the constraints. If the salaryCns constraint is
violated, the professors will be found and their salaries will be updated. After this, the
transaction is committed to allow the system to evaluate the constraints for the current
object. The keyword this refers to the object which invokes the current method.

When the constraints are violated either during the execution of the transaction or
while checking the named objects at the end of the transaction, control can be passed to recovery operations defined for this purpose. We name the recovery operations as rescue. Rescues defined in KOM are mainly used to resolve constraint violation rather than program error or system failure. More detail can be found in the next section.

### 4.2.3 Execution Semantics of Transactions

The correctness criterion for the transaction model is concerned with the order in which transactions are executed, and when and which objects can be checked for constraints after transactions are completed. The first issue is mainly related with triggers and has been discussed in Section 3.4.3. In this section, we stress the second issue and consider mainly about constraint evaluation.

During the execution of a KOM program, a transaction is called active if the control has passed the transaction keyword but not yet passed the endtransaction keyword. Due to dynamic and static nesting, any number of transactions can be active at any given moment.

When a transaction becomes active, the objects referenced by the variable list after the transaction keyword are said to be protected by that transaction, and they remain so while the transaction is active. Objects may well be protected by more than one transaction at the same time.

While an object is protected, the checking of constraints that normally accompanies attribute updating is not performed. When a transaction becomes inactive, any object which thereby becomes unprotected has its constraints checked. More precisely, an object has its constraints checked when a transaction becomes inactive if it was protected by the transaction but not by any other active transaction. For the purpose of these semantics, a rescue statement (see Section 4.3.5) behaves the same as a transaction statement which names the object for which a failure has occurred.

Two methods are defined in Figure 4.2 to illustrate the semantics of transactions. Each of the methods contains a transaction to perform some operations. Inside method A, a transaction is defined within which method B is called. Method B defines another transaction. Since the variables x and u in the two transactions refer to the same object, the constraints for the object referenced by x can not be evaluated when the
transaction in method B is finished. They can only be checked after the transaction in method A is accomplished. The constraints for the object referenced by \( v \) are checked when the control passes the keyword `endtransaction` in the second transaction and the constraints for the objects referred by \( x, y, z \) are evaluated at the end of the first transaction. The control flow of the execution is also shown in Figure 4.2.

The implementation of this scheme is particularly easy. Each object has an associated counter whose value is the number of active transactions by which it is currently protected. This counter can be readily incremented or decremented as transactions become active or inactive. Objects with a positive counter value are protected against normal constraint checking, but their constraints are checked as soon as the value is decremented to zero.

### 4.2.4 Failure Handling

In KOM, a *failure* occurs when constraints are violated due to the updates of the object attribute values. KOM provides rescue statements to permit an attempted
recovery from failure inside transactions, methods or functions. This is similar to the rescue clauses provided by EIFFEL [66] to retry operations once they fail. In the following, we only discuss the case of rescue statements in transactions. Rescue statements can be applied to methods and functions in the same way as they can with transactions.

There are two possible ways to deal with constraint violation in KOM. When there is no rescue statement defined in a transaction, errors cannot be recovered and the transaction has to be aborted. In this case, the execution is stopped and the control returns back to the system. Another approach is to define a rescue statement for the transaction. When a failure occurs inside a transaction, the system automatically passes control to the rescue statement defined in that transaction. Some actions can there be taken to correct the errors. The syntax of a rescue statement is given in Table 4.5.

The statements between the keywords `rescue` and `endrescue` can be any programming statements except other rescue statements. Once the execution passes to rescue statements, those objects whose constraints are violated in the transaction become protected. Constraints can only be checked for them after all the recovery operations are executed. A transaction or function may have at most one rescue statement. If there is no rescue statement defined, the default of `rescue endrescue` is assumed.

In order to efficiently correct failures, constraints can be tested inside rescue statements through constraint methods. This has an advantage that errors can be fixed directly by finding which constraints are violated and which attributes are relevant to the constraints. Several methods can be called to test constraints which are inherited from the root type `Object`, or generated by the system for a type. More detailed information about constraint methods can be found in Section 4.4.4.

There are two general operations which are supported in rescue statements. One is the operation `undo` which undoes all updates since the beginning of the current trans-
Lecturer::newSalary(salary: int)->void
begin
    ......
transaction this do
    ......
Salary(salary);
rescue
    ......
if (not salaryCns())
    then begin
read(number, "Input New Salary Again:", salary);
Salary(salary);
end;
endrescue;
endtransaction;
......
rescue
    if (Violation() 
        and (not empCns()) 
        and Position = "Lecturer")
then begin
    /* 1. find the professors whose salaries are 
        lower than the current lecturer's salary;
        2. update the professors' salaries. */
foreach x: Lecturer
    where x.Position = "Professor"
begin
if x.Salary < Salary
    then begin
read(number, "Input Professor's Salary:", sy);
Salary(sy);
end;
end;
end;
Figure 4.3: Rescue Statement in Method newSalary()
Chapter 4. General Functionality of the KOM Language

action, and then continues execution at the next operation after the current transaction statement. Another is the operation abort which simply stops execution and returns control back to the system. Syntactically, these two operations can be defined anywhere in the KOM programs, but they are more meaningful if they are used inside rescue statements, especially for the undo operation. They are useful when the rescue statements can not mend the errors and compulsory actions should be taken.

If failure still exists when a rescue statement for a transaction finishes, control passes to the rescue statement of the innermost surrounding transaction or function for which the rescue statement is not already active. If failure still exists when a rescue statement for a function finishes, the abort action is taken.

An example in Figure 4.3 is given to illustrate how rescue statements function in KOM. There are two rescue statements in the method newSalary(). The first one re-assigns the salary when the salary constraint salaryCns() is violated. If the constraint is still violated after this rescue, the control will go from the first rescue to the second one to test the empCns() constraint. Actions may be taken in the second rescue if this constraint is violated.

4.2.5 Comparison

Similar work is introduced in a paper called “A Model for Active Object Oriented Database” [11]. It also defines its transaction model based on the concepts of active databases, OODB’s and nested transactions. In this section, comparison between the two systems is made and we find that KOM provides almost all features discussed in the paper as well as some extra ones in terms of KOM type definitions. In order to refer to the two systems easily, we name the model in [11] as AODM.

Both KOM and AODM are defined as object-oriented database systems and thus the general characteristics of OODB’s are supported. Both systems agree on that the interaction with databases is through methods and methods are triggering events. The actions of triggers may also be regarded as the execution of methods. Therefore, methods are naturally used in defining transactions.

In AODM, the database system is responsible for operations such as define, abort or rollback transactions. In KOM, users have their own choice by declaring them
explicitly. One benefit of explicit defining transactions is that users can commit a sequence of operations on multiple objects as one atomic operation and constraints can only be checked when the transaction reaches the end. These allows more sophisticated constraints to be specified and enforced. In addition to this, users are allowed to define abort or rollback operations by needs of rescue statements. A sequence of recovery operations can thus be executed before the KOM system really aborts the transaction or rolls back to a previous one. If the operations really solve the occurring problems, then transactions can still be committed and the execution will continue. Transactions abort only when rescue also failed.

The actions of triggers in AODM are defined through time intervals of method executions. In other words, triggered actions occur before some methods are executed or after they are accomplished. In AODM, the methods are not necessarily defined in the current object, but may belong to any object in the database. However, KOM specifies the time intervals of method executions only for those methods defined in the objects. We believe that this can preserve the encapsulation of objects better. When the actions of triggers in an object need to be triggered by methods defined in other objects, triggers can be defined in those objects and in the action body, the actions for the current object can be implicitly invoked. If the events are composite ones, only the operations which can trigger other events are specified. The actions are also taken inside the current object.

The implementation of AODM is quite similar to that of KOM. In AODM, each object is attached with a method call method processor. This method handles the active part of an object and records all executable actions triggered by the execution of the method. Before executed, those actions trigged by the current method are inserted into a queue and those in the queue are executed if they are required to do so before the beginning of the method. This is followed by the execution of the current method. Once it is done, active actions triggered by the after-method events are inserted, followed by the methods which must be executed after the current method is finished. KOM processes triggers by passing the current rule set to an expert system shell before and after the current method execution. Rules are fired in the way that the expert system shell supports and the semantics and ordering of rule firing are specified in the language.
In both systems, the nested transaction structure is created by invoking methods inside method executions or triggering actions inside triggers. It is executed and maintained by the facilities defined above.

4.3 Functions and Methods

4.3.1 KOM Programs

KOM programs are implemented by a sequence of functions written by the KOM language. Generally, two types of functions are supported which are member functions and free functions. Member function are methods which are used to implement the behavior of objects and they must be specified in a type definition. Free functions are those defined outside object types to perform general operations for specific applications. They can be invoked in the whole range of an application program. To interpret our semantic description correctly, note that methods are functions, but functions are not necessarily methods. The precise syntax of functions and methods is shown in Table 4.6.

A free function in KOM comprises of the function declaration and function body. A function declaration gives the specification of the parameter list and return type of the function. The return type of the function can be any data type allowed in KOM and it is void if no value is returned. It only differs from a method declaration in the sense that the type that the method belongs to must be given at the beginning of the method declaration followed by two colons. A function body in KOM starts with begin statement and finishes with end. Between the begin and end statements, two sections are optional: declarations and operations. The declaration section declares variables which are used inside the function. After that, general operations written by programming statements are described.

Each application program has to provide one main function to indicate the place where execution begins. It must have the name main and returns no value. Generally, types are declared before they are used; otherwise the order is unimportant. Table 4.7 shows how a KOM program can be defined.
Chapter 4. General Functionality of the KOM Language

4.3.2 Construction of Objects

A constructor is a method which creates a new instance of a type and its attribute values. The syntax of constructors is shown in Table 4.8. When a constructor is declared, it must possess the same name as the type and has no return type to be specified. After the parameter list, an object identifier must be given which can be any one of the parameters that users want to pass as an object identifier. No two instances are allowed to have the same name. After creation of an object, one can always find a reference which actually points to that object by using the object name.
A constructor for type Lecturer in Figure 3.1, for instance, can be defined as in Figure 4.4. The constructor is no prefixed type and followed by the type name Lecturer and constructor name Lecturer. After the parameters passed for object states, an object name is assigned by the lecturer name after the colon. The statements inside the constructor body assign values to the object states passed in the assignments name, age, employment, salary and position.

Note that a type can have several constructors but must be with different parameter specifications. A real constructor chosen for an object is determined at run-time by matching the parameter types and specifications.

4.3.3 Methods for Accessing Object States

Only the *get* and *set* methods in a type definition can access the states of objects of that type. In KOM, they are underlined methods and implemented by the system whenever a type is declared. Therefore, each attribute in a type associated with two methods. One is used to get the value of the attribute and another to set the value for the attribute. Usually, a *get* method has a the return type which is the same as one of the attribute but with an empty parameter list. It is used to retrieve the value of the attribute. In opposite, a *set* method is defined with return type `void` but with one parameter of the type of the attribute. It is used to assign a value to the attribute.

When a type of an attribute is a subtype or type of `Set`, it usually has two kinds of assignments. One is to assign the reference of a set directly to the attribute. Another
is to insert members for the set. If the attribute is not assigned yet or required to be modified, then assign the attribute too. The syntax of get and set methods for an attribute with an aggregate type is similar to those defined above.

4.3.4 Constraint Methods

A list of constraint methods is also generated by the KOM system according to the constraints specified in a type definition. Generally, the methods are member functions of the type defined those constraints. Any instances of that type can test the constraints by calling these methods.

A standard constraint method is implemented in the KOM types: Constraints(). It returns true when constraints are satisfied, otherwise it returns false. If the user wants to know the evaluation result of a particular constraint, e.g., the salaryCns, a function named with the same name of the constraint, e.g., salaryCns(), can be called. Individual constraint methods are also generated by KOM systems. Constraint methods have boolean return type and empty parameter list. Since they belong to a type, constraint methods can be called by any instances of that type.

4.4 Query Statements — Part I

Designed as other object-oriented database systems [10] [53], query statements in KOM concern about the retrieval and possible manipulation of objects in the database. Usually, they are achieved by iterations over a set or class of objects. When the required conditions are satisfied, operations are applied on the objects retrieved. This continues until all the objects of the set or class are searched.

At the schema-level, type definitions can be queried as ordinary objects. This can be done by querying Type. Usually, information from schema-level can not be modified and the query only returns a set of values representing the properties associated with the types, included its supertypes and subtypes hierarchy and relationships involved.

When complex objects are queried, the statements can be embedded into each other through the composite links of instance variables.
Chapter 4. General Functionality of the KOM Language

Example 1:

```plaintext
default lect: Lecturer
  where lect.Position="Professor"
  and lect.Department="Computer Science"
begin
  write("The professor is", lect.Name);
end;
```

Example 2:

```plaintext
std:=find Student("John");
write("The Student Subject:", std.Subject());

forsome crs: Course in std.takeCourses
  where crs.Mark="high distinction"
begin
  write("The Course Name:", crs.Name());
end;
```

Figure 4.5: Example of Query Statements

4.4.1 Single Query Statements

Two basic query constructs are provided here. The `foreach` construct directs that a given statement should be done for each individual object in a given class which satisfies given conditions. The `forsome` construct directs that a given statement should be executed for one member of a class which satisfies the conditions. If more than one member of the class satisfies the conditions, the one is chosen non-deterministically. The syntax for basic iteration constructs can be seen in the Table 4.1.

Examples of the query statements are shown in Figure 4.5. The first example finds all the professors in the department of computer science from the class Lecturer. The second one uses the `find` statement to get the reference of a student named John and prints out his subject. The query statement returns a course name which the student John obtained the grade “high distinction”. In fact, the `in` clause in the query statement implements a query for composite objects. Since the attribute `takeCourses` in type Student has type Set, the `forsome` statement iterates over each instance of class Course and tests if it is a member of the set. If it is, check conditions in the `where` statement. The query completes when the course has a grade “high distinction”.
foreach std: Student
begin
  forsome crs: Course in std.takeCourses()
  where crs.Mark="high distinction"
  begin
    write("The student is ", std.Name());
    write(" major ", std.Major());
    write(" degree ", std.Degree());
  end;
end;

Figure 4.6: Example of Embedded Query Statements

Otherwise, continue the next course in the class Course.

4.4.2 Embedded Queries

The query constructs allow iteration over multiple classes through multiple query statements. For example, a user may wish to point out the name, major and degree of each student who has obtained at least one "high distinction". This example is shown in Figure 4.6.

The example above requires that two query statements are specified. The first one iterates over the students of the Student class and the second one filters out the students whose courses include one with "high distinction". The two queries are linked through a set function std.Course() and the pointers returned are restricted to the set of courses that the current student takes. Several types of objects are associated together through the conditions and operations of query statements. The join operations identified in relational data models are implicitly achieved. Query statements can be nested to arbitrary levels.
Chapter 5

Relationships: Description of Associations among Objects

5.1 Overview

This chapter identifies the importance of modeling associations and inter-relationship constraints among objects in the design of object-oriented database systems. Relationships are very useful concepts for building larger applications which contain interactions among many types of objects. Traditionally, the representation and behavior of relationships are buried in the specification of attributes or the implementation of methods of objects. Thus, the semantic significance of relationships is hidden from users. The specification of relationships delegated to programmers rather than the designers of the object-oriented systems leads to some drawbacks, such as:

- The information about relationships is divided among different classes of objects, which are linked by the attributes with abstract data types. Hence, the semantics of the relationships is not clear and the access of relationships is awkward. The control of relationships is not localized inside one object.

- To implement the relationships, the user has to consider the relevant attributes and to manage the consistency of the relationships. The interactions among objects are defined among object attributes and methods. This makes the modifications of relationships rather complex and the maintenance of consistency of relationships difficult.
Chapter 5. Relationships: Description of Associations among Objects

The explicit declaration of relationships can help to abstract the semantics and interactions of relationships in a very natural way and support users' operations on them [37] [87]. The whole information relevant to the same type of relationships can be gathered into one definition, and the fundamental principles of the object-oriented paradigm can apply to the relationship types, i.e., object identity, encapsulation and inheritance. In fact, the representation of relationships as an abstract construct increases the expressive power of the object-oriented database system. From certain viewpoint, this abstract construct represents an inherent constraint among objects of different types. This constraint is not something to be hidden, but rather to be specified abstractly.

Most of the existing object-oriented database systems bury the descriptions of relationships in the specification of attributes or methods in the object types [35] [42] [54] [76]. Even for some systems supporting the representation of relationships as an independent construct, the semantics expressed and the functionality supported are very limited. First, these systems only support binary relationships even though they declare that the models provided can be extended into n-ary relationship models by combining several binary relationships [12] [21]. This is not necessary to be true because the n-ary relationships built on n binary relationships are not semantically equal to n-ary relationships in mathematical terms. On the other hand, the latter uses could be very impractical in real applications.

Second, the existing relationship models are normally quite simple. They only support the linkage of objects from different classes, and provide basic access and manipulation of elements of relationships [87] [92]. The full functionality from the object-oriented paradigm is not captured. For example, they do not support the full specification of relationship types as ordinary object types. Encapsulations and inheritance are not considered.

Third, the semantics of relationships captured by previous models is restricted to inter-constraints and only certain types of dependencies are enforced. For example, the model described by [37] mainly deals with the slave-master relationships. In an individual relationship, the slave is a dependent object and the master is an influential object. The possible behavior between these two types of objects is described by a relationship class called link. Another model proposed by [67] was motivated by
cating inter-constraints into the object definitions. Objects are linked by constraints existing between them and the relationships are represented by a set of equations to accommodate different types of behavior.

Finally, the existing models classify the binary relationships into several categories: one to one, one to many or many to one, and many to many relationships [87] [92]. No unified representation is studied to contain all mappings. From our observation, we believe that such unified representation should be stressed by the designers of relationship models, and generic models thus require to be proposed.

From an object’s perspective, we find that it is useful to divide the representation of relationships into what we called internal relationships and external relationships. Internal relationships are the relationships which relate composite objects to its components. These relationships convey an object its attributes, constraints, methods and rules. As such, the internal relationships of an object constitute an integral part of an object’s definition. Chapter 2 was concerned about this type of relationships. On the other hand, external relationships are the relationships that hold between sets of objects, rather than their components, to express various types of associations and constraints. Unlike internal relationships, external relationships do not contribute to or affect the definition of an object. They are independent concepts and have their own properties and semantics.

The work presented in this chapter is motivated by the need for more support of representation and manipulation of external relationships, especially for expressing n-ary relationships and accommodating knowledge into them. External relationships in KOM are proposed as individual types. They explicitly model relationships among objects and certain behavior of them. Like ordinary object types, relationship types can have their own properties and the embedded knowledge in the forms of constraints and triggers.

This chapter is organized as follow. In Section 5.2, a formal definition of relationships is given with a description of their semantics. An example is given to demonstrate how to define and use a relationship in KOM. Section 5.3 provides the specification of relationship types and the definitions of slice types, attributes, constraints, methods and triggers. In Section 5.4, we introduce the generic relationship type and its proper-
ties which are defined in KOM. Section 5.5 describes the general rules for inheritance of relationship types. Finally, a query language for relationships is supported which allows users to retrieve various types of information from relationships in Section 5.6.

5.2 Definitions and Semantics of Relationships

This section provides the basic formal theory for defining KOM relationship model based on n-ary relationships. Compared with other models, the relationship model here has many advantages and several of them are very important, such as

- The model supports very generic forms to specify relationships in an object-oriented data model. No distinction has been made between binary and n-ary relationships. In addition, the model captures different types of mappings in one specification. There is no need for users to concern and classify the relationships into different types. To define a relationship type, users only require to consider the participants of the relationships and the semantics that the relationship type presents.

- Relationship types play the same role as ordinary types in KOM. Hence, the features of object-oriented paradigm can be fully obtained by relationship types. Also, the specific features of relationship types are developed which are based on the definitions and semantics that we introduce below.

The term relationship covers many forms of connection among objects. A relationship in the database sense can be defined recursively as a semantic association between one type of object and another. The number of types of objects involved in a relationship can be arbitrary. In general, a relationship type associates a list of types which are called participant types. Informally, an instance of a relationship type is a collection of tuples, each of which is a list of one object from each participant type, together with the attendant properties.

Definition An n-ary relationship type specifies an ordered list of types $T_1, T_2, \ldots, T_n$, called the participant types, and a set of properties in the
form of attributes, constraints, methods and triggers. Let \( X_1, X_2, \ldots, X_n \) be the classes of objects defined by the types \( T_1, T_2, \ldots, T_n \), respectively. The direct product of \( X_1, X_2, \ldots, X_n \) is the set
\[
X_1 \times X_2 \times \cdots \times X_n = \{(x_1, x_2, \ldots, x_n) \mid x_1 \in X_1, x_2 \in X_2, \ldots, x_n \in X_n\}.
\]
Each element of the direct product is called an n-tuple, or simply a tuple. An instance of the relationship type \((T_1, T_2, \ldots, T_n)\) has a value consisting of a set of tuples \((x_1, x_2, \ldots, x_n)\), where each \( x_i \) has type \( T_i \). In other words, the value is a subset of \( X_1 \times X_2 \times \cdots \times X_n \). An instance of an n-ary relationship type is called an n-ary relationship. The word “2-ary” is usually replaced by the more common word “binary”. Here, we distinguish the relationship type and the relationship instance by calling them a “relationship type” or a “relationship”, respectively.

In order to provide reasonable functionality for relationship types, it is necessary to permit attributes and constraints to be attached not only to instances of the types but also to the individual tuples comprising them. KOM introduces an innovative generalization of this idea, which we call slices. A slice is a subset of a relationship instance (i.e., it is a set of tuples) defined by fixing some of the elements in tuples and permitting the others to vary freely. A vector describing which entries are fixed and which are not is called a pattern.

**Definition** Let \( R \) be a relationship type defined on classes \( X_1, X_2, \ldots, X_n \). A pattern consists of a vector \( P = (u_1, u_2, \ldots, u_n) \), where each \( u_i \) is either an instance of \( X_i \) or the special symbol “*”. The slice of \( R \) specified by the pattern \( P \) is the set of all tuples in \( R \) which match \( P \), where “*” matches anything. Formally, the slice is
\[
\{(x_1, x_2, \ldots, x_n) \in R \mid x_i = u_i \text{ or } u_i = "*" \text{ for } 1 \leq i \leq n\}.
\]
Note that \( R \) itself is a slice, corresponding to the pattern \((*, *, \ldots, *)\), and \( \{(x_1, x_2, \ldots, x_n)\} \) is a slice for each tuple \((x_1, x_2, \ldots, x_n)\), corresponding to pattern \((x_1, x_2, \ldots, x_n)\).
Chapter 5. Relationships: Description of Associations among Objects

Figure 5.1: Relationship of Students Taking Courses on Some Majors

An example shown in Figure 5.1 illustrates the definition of relationship types and slices. The type defined is a ternary relationship type which describes the association among students, courses and majors. Each point in the three dimension space corresponds to a tuple which comprises one student, one course and one major, e.g., ("John", "Information Systems", "Computer Science"). If we want to know all the courses that student "John" takes as part of major "Computer Science", the slice ("John", *, "Computer Science") is obtained, which is a line in the picture. Properties of this slice can be defined, such as the mark and the set of assignments required by each course. Also, if we want to know about the courses and the majors that the student "John" possesses, the slice ("John", *, *) is considered, which is a plane in the picture. This slice can have properties like the degree that "John" can get. Therefore, the object model should support the description of semantics of both relationships and the slices derived from them.
Chapter 5. Relationships: Description of Associations among Objects

5.3 Modeling Relationships

Relationships are declared as types in KOM by using the keyword `relationship`. In general, a relationship type can be defined as an n-ary relationship and inherited from another relationship type or the system-defined abstract type `Relationship`. Here, type `Relationship` is the root of all relationship types, which contains the basic attributes and methods required by any relationship types. It has zero participants and allows its offspring to define as many participants as required.

The participants of a relationship type must have unique names, but can be with the same types. In addition, a relationship type has its own properties. These properties can be expressed using an extension of the syntax for ordinary object type definitions, namely slices, attributes, constraints, methods and triggers. The syntax of a relationship type definition in KOM is defined in Table 5.1. The slices section in a relationship type defines the slice types used in the specification and operations of that type. In the following, we will use an example to explain each part of relationship type definitions in detail.

The example that we are going to use is the binary relationship type `take` defined in Figure 5.2. It describes associations of “Students Taking Courses” between two types
of objects: Student and Course. A key point in defining relationships in KOM is how to classify relationship instances. In fact, different semantics for the same definition can be obtained by different classifications of relationships. For example, if each tuple of a student and a course corresponds to an instance, the semantics of slices is ignored. If we define the instances according to each student taking several courses, the semantics of a student with a set of courses is described. On the contrary, the semantics of a course with a group of students can be expressed if the instances corresponding to each course taken by a class of students. However, a more generic description for this relationship, i.e., a group of students taking a set of courses or a set of courses taken by a group of students can be defined by the same type. The classifications of instances can be made according to the students' subject, or the department or faculty that they are enrolled. It has advantages that the type contains all the semantics described by three considerations mentioned above. To explore the power of the KOM relationship model, we use the final case as our classification of relationship instances.

5.3.1 Slice Types

The first part of a relationship type definition defines all possible slice types which may be used inside a type definition. This part starts with the keyword slices and followed by a list of slice type definitions. A slice type defines a type which is composed of n components, and each component must be either the same as the corresponding participant, or *. In KOM, two slice types are not allowed to have same pattern if they have the same slice type names. The syntax of slice types is shown in Table 5.2.

Unlike other types in KOM, slice types are dependent concepts which are relied on the relationship type defining them. In fact, slice types must be defined inside relationship types, and only attributes and variables inside method bodies can be specified by them.

In each instance of a relationship type, a variable with a slice type refers to a set of slices which are defined by that type. For example, in Figure 5.2, the slice type allcourses specifies variables which refer to one student taking courses for all the students contained in the current instance. In other words, the number of slices defined by slice type allcourses is equal to the number of students defined by an
relationship take(std: Student, crs: Course) isa Relationship
begin
    slices
        allcourses: (std, *);
        tuple: (std, crs);
        class: (*, crs);
    attributes
        facultyname: string;
        grade of tuple: number;
        nopassed of allcourses: number;
        enrollees of class: set-of Student;
        university: string;
    constraints
        Undergraduate:
            foreach s: Student in std
            enforce std.Status = "undergraduate";
        NoStudentsofClass:
            foreach x: class
            x.cardinality()<30;
        StudentGrade:
            foreach x: tuple
            x.grade<=10 and x.grade>=0;
        MarkCns:
            foreach x: allcourses
            enforce (x.nopassed>0 implies forsome y:Course in y.crs
            enforce (x.std, y).grade>5);
    methods
        take(facultyname: string);
        GetGrade(student: Student, course: Course)->number;
        AssignGrade(student: Student, course: Course)->void;
        NoPassed(student: Student)->number;
        Classmates(coursename: Course)->set-of Student;
        University(univ: string)->void;
    triggers
        monitor:
        if message(AssignGrade)
        after-action
            name:string;
            std:Student;
            read(string, "Student Name:", name);
            std:=find Student(name);
            call NoPassed(std);
        endaction; endrule;
end;

Figure 5.2: Relationship Type take
Chapter 5. Relationships: Description of Associations among Objects

5.3.2 Attributes

Like ordinary attributes, relationship attributes are defined by attribute names and their types. The syntax of relationship attributes can be seen in Table 5.3. Since slice types have been introduced, attributes can be defined for the slices too. An attribute for slices is described by the keyword `of` followed by the pattern of slices, i.e., a slice type, and the type of the attribute. Ordinary attributes are defined by attribute names followed by the types of the attributes. Consider the example in Figure 5.2. The attribute `grade` associated with slice type `tuple` has type `number`. Actually, it is equal to the mark that a particular student gets in a particular course. In one instance, the `grade` attribute can have a number of values which depend on how many tuples of courses and students hold for an instance.

Slice typed attributes can be queried, constrained and manipulated like ordinary attributes. When slice typed attributes are used in the program, they are usually manipulated through `foreach` or `forsome` clauses. The iterator clauses are utilized to get the references to slices in an instance of the relationship type.

The participant variables declared with a relationship type can be used in the same way as the attributes. They can be queried, constrained and modified inside a type definition or method bodies. In addition, when attributes are defined with slice types, participant variables can be used to get participants of a particular slice instance. For example, in Figure 5.2, attribute `nopassed` of type `allcourses` can use `nopassed.crs` to refer to the second participant of the slice. When an element of a slice type is declared by a participant variable, the above formulae returns a reference.
to that participant. If the element is declared as *, it returns a reference to a set which hold a set of objects composed that participant. Consider the above example again: variable `nopassed.std` returns only one student, but variable `nopassed.crs` returns a set of courses that a student takes. The participant variables differ from the attribute variables in the sense that they can be accessed directly inside and outside the relationship type definition, but attribute variables can only be accessed directly inside the relationship type definition.

### 5.3.3 Constraints

Constraints supported in relationship types have similar syntax to those defined for ordinary object types, except that restrictions can be put on participants and on slices as well as on entire relationship instances. The participants of relationships can be constrained like ordinary attributes. For example, the relationship `take` only concerns undergraduate students. A constraint `Undergraduate` is thus defined in Figure 5.2 to restrict the variable `std` to only have the status `undergraduate`. However, the semantics of constraints for participants is different from ordinary constraints. Using the above example, the constraint requires all the participants of an instance of relationship type `take` to have a value `undergraduate`.

Constraints on slice typed attributes can be defined by considering them as ordinary attributes. When such constraints are defined, all the slices contained in an instance are restricted. This is equivalent to that the constraints are evaluated for every slice but only defined once. For example, in Figure 5.2, constraint `NoStudentOfClass` limits the number of students for each course.

When a constraint involves more than one slice typed attributes, some operators must be used to identify which slices are restricted. Take constraint `MarkCns` as an
example. For each value of the attribute `nopassed`, at least one `grade` among a set of tuples is equal or greater than 5. Constraints regarding to several slices are usually specified by using `forsome` and `foreach` statements.

5.3.4 Methods

Methods of a relationship type include a constructor for relationship instances and functions which perform operations on them. A constructor of a relationship type simply passes the name of the relationship instance to the relationship type and an instance is created without assignment of any tuples. For example, when the relationship type `take` is defined, an instance of it can be constructed by a group of tuples of students and courses and the way by which they are constructed usually depends on the classification of students. One approach used here is that the students can be classified according to their subjects. For those students enrolling in the faculty of `Engineering`, they can constitute one instance of the relationship type. Simply, the instance can be named as `Engineering` and accessed through this name.

To add tuples to a relationship instance, the instance should be retrieved through its name. After this, the function `insert` is called and the arguments are passed as a tuple to the relationship instance. For example, two arguments ("John", "Information Systems") can be passed to the function `insert` to assign the values for `Std` and `Crs`.

Member functions of a relationship type can be specified in the same way as those in a type definition. The only difference is that they should provide extra codes to identify the slices and participants which are processed. Several operations are supported in relationships, as follows:

1. Operations on Relationships. A relationship instance can be located using operator `find` by providing the relationship type and the instance name. For example, a pointer to an instance of relationship `take` can be obtained by issuing `r1 := find take("Arts")`. This operation returns the instance for all the tuples of students and courses in the faculty of "Arts". After a reference to a relationship instance is found, references to individual sets of participants can be obtained by applying function `Participant(i)` when the `i`-th participant is asked, or using
the participant variable directly. For example, the following statement returns a
set of students for that relationship instance.

\[
\text{stds:=r1.Participant(1); or} \\
\text{stds:=r1.std;}
\]

2. Operations on Slices. Slices can be operated like relationships. A slice can also
be found by operator \text{find} followed by the slice type and the participant names.
For example, if we want to find all the courses taken by the student "John", this
statement should be issued:

\[\text{s1:=find allcourses("John", *)} ;\]

The "*" here matches all the courses which are taken by "John". This operation
returns a reference to a set which contains all the tuples of student John and the
courses that he takes. Note that this operation is relevant to a particular rela-
tionship instance; if we are not in the context of one instance (such as executing
one of its methods), a reference to an instance must be provided, eg:

\[\text{s1:=find r1.allcourses("John", *)} ;\]

Similarly, the slices corresponding to \text{class} and \text{tuple} can also be obtained:

\[
\text{s2:=find class(*, "Information Systems");} \\
\text{s3:=find tuple("John", "Information Systems");}
\]

The references to the participants of slices are also obtained by using function
\text{Participant(i)}. An example of this is to get the reference to the courses that
"John" takes.

\[\text{c1:=s1.Partialicipant(2);}\]
In a relationship type, each attribute is also attached with a pair of set and get methods. For attributes of slice types, the methods are defined for the slices rather than the relationship instances. Thus, whenever an attribute value is assigned or retrieved, the slice is needed to be found first, and then the set or the get methods are invoked.

In Figure 5.3, some examples of method definitions are given. The first two methods set and get the grade for a pair of a student and a course. The third method calculates the number of courses which are passed by a student. This function has three parts: the first one finds the slice of allcourses for the given student and initializes the variable NoPassed. The second part finds all the courses taken by the student. Tuples are identified and the grade of each course taken by the student is obtained. The variable NoPassed is increased when a course is passed. The last part calls the set method to assign the result to the attribute nopassed and returns the value to the calling place.

Transactions can also be used in the manipulation of relationships. Special care must be taken since the updates of attribute values of participant objects can affect the evaluation of relationship constraints. For example, a situation can occur when the first transaction operates on object x, y and the second one on relationship z, while the objects x, y are the participants of the relationship z (See Figure 5.4 (1)). Constraints of relationship z can not be checked until the transaction for objects x, y is finished. A correct way of defining this transaction is shown in Figure 5.4 (2).

5.3.5 Triggers

Like constraints, triggers are extended for relationships to capture the updates of attributes and receiving messages either for relationship instances and instances of slice types. The operations on triggers introduced previously can also apply to the relationship ones and additional semantics for slices has been included. In a relationship type, triggers should react to updates of both relationship attributes and slices attributes. Since one relationship instance can have a set of values for each slice attribute, rules in the relationship are performed for every tuple of the relationship instance. This is achieved by putting foreach clause at the beginning of the rules. For example, the trigger monitor can be fired for every tuple of take when the attribute grade is up-
Chapter 5. Relationships: Description of Associations among Objects

Figure 5.3: Method Specifications

\begin{verbatim}
take::GetGrade(student: Student, course: Course)->number
begin
    x: tuple;
    x:=find tuple(student, course);
    return x.grade();
end;

take::AssignGrade(student: Student, course : Course, mark : number)->void
begin
    x: tuple;
    x:=find tuple(student, course);
    x.grade(mark);
end;

take::NoPassed(student: Student)->number
begin
    x: allcourses;
    y: tuple;
    np: number;
    np:=0;
    x:=find allcourses(student, *);
    foreach z: Course
    begin
        y:=find tuple(student, z);
        if (y.grade>=5) then begin np:=np+1 end;
    end;
    x.nopassed(np);
    return np;
end;
\end{verbatim}
Chapter 5. Relationships: Description of Associations among Objects

transaction x, y do
       .......
transaction z do
       .......
endtransaction;
       .......
endtransaction;

(1)

transaction x, y, z do
       .......
endtransaction;

(2)

Figure 5.4: Transactions on Relationships

dated and is over 5. This rule calls the method nopassed for each student to gain the recent result of the course work. Rules for slices are usually defined by those for set operations.

5.4 Generic Relationship Type

Relationship is the generic relationship type which provides basic properties and operations required by relationships. It serves as a supertype of all relationship types. Information in the Relationship type includes a constructor for relationship instances, displaying the content of a relationship instance, insert, delete or test tuples and slices, etc. The fundamental attributes and methods defined in type Relationship are listed below.

1. The Relationship Identifier. An attribute which is named as RelationName is provided to store the name for each instance.

2. Create Relationship Instances. The create operation generates a new relationship instance with an identifying name provided by users. For example, when relationship type take is defined, the classification of instances can be in terms of faculties. In this case, the RelationName is assigned by the faculty name, and a list of tuples is given which corresponds to those students taking courses in that faculty. Initially, a relationship instance has no tuples.
3. **Delete Relationship Instance.** The delete operation removes the relationship instance and those tuples which comprise the instance. The objects which participate in the relationship instance will still exist without change. On the other hand, the deletion of participant objects affects the relationship instance. It may result in inconsistency if one object in a tuple is removed and the tuple still exists. In KOM, the general reaction to the deletion of participant objects is to remove the tuples which contain this object.

4. **Display Relationship Instance.** The display method displays tuples of a relationship instance on the output device. There are three ways of presenting relationship tuples. One depends on the input order of tuples. The other two display the tuples according to the alphabetical order on either the sides of the participants. All atomic objects are displayable and their values are shown. The abstract data objects and aggregate objects are displayed by their identifier names.

5. **Operations on Tuples.** The operations on tuples of a relationship instance include insert, delete and test. The insert function adds a pair of objects into the current relationship instance and the relationship identifier is not changed since there is no creation during the insert operation. The delete function simply removes a tuple from the relationship instance. The relationship instance is not removed from the database even if the pair is the only tuple of the relationship instance. The operation isTuple tests if the given tuple is a tuple of the given relationship instance. It returns false if the current tuple of objects is not contained by the relationship instance.

6. **Participant Selection Operations.** The function Participant returns the object participating in a specified position in a given relationship instance or slice. The position number must be given when function Participant is used. These have been described in Section 5.3.4.
Chapter 5. Relationships: Description of Associations among Objects

5.5 Inheritance

5.5.1 Introduction

The importance of capturing relationships as an independent logical construct in object-oriented database systems has been pointed out many times in the literature [12] [81][87]. However, although much effort has been spent on the investigation of the expression and semantics of relationships, other key issues, such as the encapsulation and inheritance of relationships, have been little considered. This is partially due to the absence of a satisfactory relationship model. Relationship models developed previously either do not support some essential features of the object-oriented paradigm, or assume that they ought to follow the same rules defined for ordinary objects. From the observations and research carried in earlier sections of this chapter, we can see that relationships possess their own unique features and have more generic and richer semantics, even though theoretically relationship instances are regarded as objects. They still must specify specific rules to describe their semantics and behavior, in terms of fundamental concepts used in the object-oriented paradigm. In this section, we discuss the semantics and essential criteria for defining the inheritance of relationship types. Related issues are also addressed. The inheritance of relationships proposed is based on the relationship model introduced in this chapter.

This section uses several relationship types as examples to illustrate how relationship types inherit properties from others. The first type is take which is defined in Section 5.2. The second is the subtype of take called takeSubject which have three participants: (std: Student, crs: Course, subj: Subject). It describes a particular group of students which require to do experiments while they are taking some courses, like students in the department of chemistry. Hence, takeSubject is distinguished from take by having a third participant with type Department. The first two participants of takeSubject are inherited from take and must have the same order and the types as they have in take, or subtypes of them. In fact, the names of inherited participants can be the same as or different from those used in the supertype. Let’s consider another example. A workon relationship type depicts lecturers working on some projects, thus having two participants: (lec: Lecturer, prj: Project).
Now, a relationship type cooperate is defined to describe the cooperation among three participants: the head of the department, the project involved and the industry organization related. Type cooperate is a subtype of type workon. From the application we know that the head in fact holds type Professor which is a subtype of Lecturer and also the appropriate name for describing the head is head rather than the lecturer lec. In summary, the actual type declaration for cooperate is

\[ \text{relationship cooperate} \]
\[ (\text{head: Professor, prj: Project, org: Organization}) \text{ isa workon} \]

begin
  
  
  .......

end;

Despite its perceived importance, the inheritance of relationships should support more generic type hierarchy and capture more semantics from application domains. This means that not only the properties of a subtype should be more specific than its supertypes, but the participants of the subtype can be different from its supertypes in numbers and names as well. In general, the isa relationship of types described in a type hierarchy does not include the definitions of properties of those types. When a type is declared as a subtype of another, it implies that it’s properties must satisfy the inheritance rules imposed by the database systems. The role of participants in a type definition is completely different. The participants of a relationship type belong to the type declaration rather than the properties of types. Before a relationship type is declared to be a subtype of another relationship type, the semantic restriction imposed on participants of the subtype must be satisfied. For instance, the generic relationship type Relationship has zero participant, while his offspring can have many as they are required from applications. The relationship type take defined in Figure 5.2 has two participants: std and crs. Precise semantics and detailed rules of inheriting participants for relationship types are discussed below. An example of relationship type hierarchy is shown in Figure 5.5.

Before formal inheritance rules are discussed, general descriptions of relationship types are introduced. Suppose we define two relationship types \( r \) and \( r' \), and \( r \geq r' \) (i.e.,
Chapter 5. Relationships: Description of Associations among Objects

5.5. Relationships: Description of Associations among Objects

Figure 5.5: Relationship Type Hierarchy

\[ r \text{ is a subtype of } r' \]

The description of relationship type \( r \) can be \( r(p) = (s; a; c; m; r) \), where \( p \) stands for a list of participants, and \( s, a, c, m \) and \( r \) refer to lists of those properties of relationship types, respectively: slices, attributes, constraints, methods and rules. Each of them can be represented by a set of definitions, such as a set of participants \( p = \{ p_i : t_i | 1 \leq i \leq m \} \) or a set of slice types \( s = \{ s_i : v_i | 1 \leq i \leq n \} \).

Here, \( p_i \) is the name of the participant, \( t_i \) is the type of that participant; \( s_i \) is the name of a slice type and \( v_i \) is the pattern defining that slice type.

The inheritance of relationship types obeys rules defined for both participants and properties of a relationship type. The latter includes inheritance of slices, attributes, constraints, methods and triggers.

5.5.2 Inheritance of Participants

The description of relationship participants in a subtype follows syntax like:

\[
\text{relationship } r(p_1:t_1, \ldots, p_n:t_n) \text{ isa } r' \\
\text{ begin } \ldots \text{ end;}
\]

In order to satisfy the general Design Principle of inheritance which is introduced in Section 3.5, the semantic restrictions on \( r \) requires that each instance of \( r \) is also an
instance of $r'$. Several ways of satisfying this requirement can be considered. The most obvious one is that the subtype has exactly same participants as its supertypes. In this case, both the participant names and their types are the same as they are defined in the supertypes. Refinement of the supertype can be achieved by defining more specific properties in the subtypes. Many relationship models adopt this assumption when the inheritance mechanism is supported. In other words, these models assume that the inheritance of relationship types follow the same rules defined for ordinary object types.

As we have pointed out in the early of this section, the inheritance of relationship types, specially participants, can be extended by capturing richer semantics and thus creating new inheritance rules. The extension of inherited participants can be made at several places. Firstly, the types of participants in a subtype can be declared as subtypes of the original ones. This is not conflict with that instances of the supertypes are the instances of its subtype. Secondly, participant names in a subtype can be renamed in order to provide more precise descriptions for more specific cases. To preserve the semantics of inherited participants, the order of inherited participants in a subtype must be the same as they are declared in the supertype, and thus provides a way of correct resolution of the participants. Finally, the newly defined participants of the subtype go to the right end of the participant list. New participants can have new names and any KOM types. For reasonable modeling of real world applications, it is necessary to permit a relationship to have more participants than its supertype. Type \texttt{workon} and \texttt{cooperate} mentioned previously are very good examples to demonstrate this idea.

We summarize the criterion for inheriting participants from one relationship type to another below:

\textbf{Participant Inheritance}: Suppose the participants of $r'$ are \{\(p'_i : t'_i\) | \(1 \leq i \leq m\}\}, the participants declared in $r$ are \{\(p_i : t_i\) | \(1 \leq i \leq n\}\}, and $r \geq r'$. Then the participants of $r$ satisfies conditions:

1) \(n \geq m\);
2) \(t_i \geq t'_i\) for \(1 \leq i \leq m\).
In general, a relationship type $r$ can be regarded as conforming to a relationship type $r'$ if the participants of $r$ satisfy the inheritance rule defined above. Formally, we have the following definition:

**Relationship Type Conformity:** Suppose the participants of $r'$ are \{p'_i : t'_i | 1 \leq i \leq m\}, the participants declared in $r$ are \{p_i : t_i | 1 \leq i \leq n\}. Relationship type $r$ conforms to $r'$, if the participants of $r$ satisfies conditions:

1) $n \geq m$ ; 

2) $t_i \geq t'_i$ for $1 \leq i \leq m$.

Note that $r$ conforming to $r'$ does not require $r \geq r'$.

The participants defined in a relationship type can be used as variables in the same way as their attributes in the type definition. In detail, they can be used in defining slice types and specifying of constraints and triggers, as well as utilizing as variables in the method bodies. Once the inheritance occurs and either the name or the type of inherited participants are changed, the new redefined participants require the correct replacement of old participants. Since the change of a participant's type to its subtype still guarantees the validation of the inherited operations and definitions, only interfaces are needed which transform the redefined participants into the old ones defined in the supertypes. This is actually a recursive procedure. The inherited participants are transformed to forms in the direct supertype first. For those inherited from upper levels of the current type, the transformation can be continued until no participants need to be converted.

The uses of participants are also different from the attributes in a type definition. This refers to that the participants including inherited ones can be used directly inside and outside the body of a relationship type definition, but the attributes inherited from the supertypes can only be used directly if they are redefined in the subtypes. Attributes inherited from supertypes have to be accessed through *set* or *get* methods. This states that variables declared in method bodies of a type can have the same names as attributes inherited from supertypes without overwriting them. However, variables in the same situation overwrite the participants provided that the participants are
renamed in the current type. For instance, one can not define variables by having the name std or crs in type take if he or she does not want to overwrite them. However, one can use the name like university as a variable name inside this type. The final point is that the renamed participants in a type cannot have the same names as they are defined in the supertypes but with different order. We put this restriction only to simplify the process at compile-time.

5.5.3 Inheritance of Slices

Several unique features of slice types make the inheritance mechanism for relationship types very complex. First of all, slice types are a dependent concept which can only exist after their relationship type is defined. They cannot be declared independently outside the type definition which derives them. Hence, there is no direct specification of supertypes and subtypes among slice types. Second, according to the inheritance rules defined previously, the inherited attributes of a type can have types which are subtypes of the original ones. Since attributes can be defined as having slice types, the definition of subtypes of slice types should be introduced. Third, the renamed participants can make that the inherited slice types have different specifications from the original definitions. Finally, a subtype of slice types may have more participants than its supertypes do and thus the number of elements defining a slice type in the supertype may be different from the one inherited by the subtype. Apparently, this results in inconsistency between the two slice types.

In fact, the above discussions contain two aspects of the problems: one is what we addressed in the inheritance of participants. Another is how to transform the inherited slice types back to the original forms so that the definitions defined in the supertypes can still be valid. In this section, we continue discussion of the first aspect. The second one is considered in the inheritance of properties of relationship types.

New concepts have to be introduced to handle the above situations. Before we define the subtype/supertype relationship among slice types, a useful concept needs to be introduced. A slice type A conforms to slice type B, if A and B have the same number of participants in declarations, and each position either the symbol “*” is used for both A and B, or A has a participant with a subtype of the type of the
participant given for \( B \). Note that here we consider the types of participants in a slice type definition rather than the names of the participants.

Now we define the subtype/supertype relationship between slice types. Suppose we have two slice types \( A \) and \( B \), and the numbers of participants of \( A \) and \( B \) respectively are \( m \) and \( n \) and \( m \leq n \). The slice type \( A \) can be regarded as a subtype of the slice type \( B \) if \( A \) conforms to \( B \) and the last \( n - m \) participants of \( B \) are given as \(*\). From the semantics of the KOM relationship model, \( A \) and \( B \) must have the same name if slice type \( A \) is the subtype of slice type \( B \). For example, the slice type \texttt{tuple}: \((\text{std}, \text{crs})\) is a supertype of the slice type \texttt{tuple}: \((\text{std}, \text{crs}, *)\). Consequently, the inheritance of slice types obeys the following rule:

**Slice Inheritance:** Suppose the participants of \( r' \) are \( \{p'_i : t'_i \mid 1 \leq i \leq m\} \), the participants declared in \( r \) are \( \{p_i : t_i \mid 1 \leq i \leq n\} \), and \( r \geq r' \). If a slice type defined in \( r' \) is \( st' = (u'_1, u'_2, \ldots, u'_{m'}) \), where \( u'_i \) is either \( p'_i \) or \( * \), then the slice type \( st \) inherited from \( st' \) satisfies the conditions below:

1. \( st = (u_1, u_2, \ldots, u_n) \);
2. \( u_i = u'_i \) for \( 1 \leq i \leq m \) if \( u'_i = * \), or \( u_i = p_i \) for \( 1 \leq i \leq m \) if \( u'_i \neq * \);
3. \( u_i = * \) for \( m < i \leq n \);

### 5.5.4 Inheritance of Attributes, Constraints, Methods and Triggers

Basically, the inheritance of attributes, constraints, methods and triggers of relationship types follows the rules defined for ordinary object types (see Section 3.5). However, complications may rise when slices types are taken into account. From rule **Slice Inheritance**, we see that when a slice type is inherited from a supertype, the number of participants will be increased if needed to match the new type definition. This may cause problems since a slice type defined for a tuple in a supertype may become a slice type defined for a set in the subtype. For instance, the slice type \texttt{tuple} in type \texttt{take} refers to a tuple but it refers to a set of tuples when it is inherited into type \texttt{takeSubject}. In fact, the slice type \texttt{tuple} becomes \((\text{std}, \text{crs}, *)\) which returns tuples of a student and a course taken in different departments. It causes semantic inconsistency between the two slice types, even though they actually refer to the same
one. Specially, when definitions and operations involving slice types are inherited by subtypes, semantic conflict can result in the subtype definition incorrect. To fix the problems, we need to introduce a concept called compatible slice types, which helps to provide a correct explanation of the inheritance of relationship types of the KOM model. In this section, the word of operations includes various operations performed in the components of constraints, methods and triggers of a relationship type.

Suppose we have two slice types: A and B. B is considered compatible with A if B is a subtype of A, and the elimination of participants from m + 1 to n of B at its right-hand side gives the same definition as A, where m and n are the number of participants defined in A and B, respectively. The reverse definition is not true, that is the slice type A is not compatible with B. Consider the above example: ignoring the "*" of the second slice type tuple gives exactly the same definition as the first tuple. The concept of compatible slice types provides a reasonable way of making operations inherited from supertypes valid in the subtypes. It allows that variables with a slice type can be replaced by those variables with a compatible type of that slice type.

When attributes, constraints, methods and triggers of a relationship type are inherited by its subtypes, the operations inherited are applied to the variables with the compatible types rather than the inherited types themselves. In other words, the aims for the inherited slice types adopting different forms from what they are really defined in the supertypes only give a good explanation and semantic consistency of the model.

Therefore, a rule is finally generated for describing the semantics and executions of the inheritance of properties of relationship types. We call the rule "variable inheritance" because the additional semantics for the inheritance of relationship types only affect the attributes and variables declared and used in method bodies. Here, we also regard attributes as a kind of variables. This rule combined with inheritance of slices and those defined in Section 3.5 consists of the restrictions for inheritance of properties of relationship types.

**Variable Inheritance:** Suppose a variable \( x' : t' \); is defined in \( r' \). Type \( r' \) is the direct supertype of \( r \). Variable \( x' \) inherited by \( r \) has the form \( x : t \); and \( x = x' \). Here, \( t \) is modified in terms of the slice inheritance rule based on \( t' \). When operations on the inherited attributes, constraints,
methods and triggers of \( r' \) are performed by the variable \( x \), \( x \) is converted back to \( x' \) according the procedure described above. From the definition of compatible slice types, we know that \( x \) and \( x' \) are compatible and thus \( x \) can be substituted by \( x' \).

5.5.5 Dynamic Binding

All the rules defined in Section 3.5.6 for dynamic binding can be applied to relationship types. Unlike those issues mentioned in this section, dynamic binding performed for relationship types does not require additional rules to describe variable matching and method invocation. This is because the binding of variables or methods at run-time can be determined by the relationship types that the variables or methods possess. Once particular relationship types are confirmed, no ambiguity occurs inside the relationship type definitions. In the following, we analyze two cases to support our conclusions.

The first case is about variables defined with slice types, including attributes. In KOM, no two slice types can have the same definition. The only case which can require the types of variables to be determined at run-time is that they have compatible slice types. According to our definition, we can see that these compatible slice types must be defined or used in different relationship types rather than the same one. Therefore, the types of variables bound at run-time can be determined through the binding of the relationship types of those variables.

Several methods can match one message at run-time in the KOM system. The one chosen to execute is determined by the distance between the declaration and the call, where distance is defined in the same ways as in Section 3.5.6. When variables with relationship types take into account, the distance between relationship types should be computed. The distance between relationship types can also be calculated by the edges between the two types in the directed tree of the type hierarchy, no matter how the participants of the types are defined or how many numbers of participants they have. Thus, the formulae for obtaining the distance between a call and a declaration is still valid for methods with relationship type parameters.
Chapter 5. Relationships: Description of Associations among Objects

5.6 Query Statements — Part II

Query operations on KOM relationships correspond to the retrieval of a collection of basic data from relationship classes or types. According to the definitions and semantics of relationships described in previous sections, these operations include querying relationship participants, and instances and tuples which are contained in the instances or slices.

The querying of relationship participants starts from the known relationship type and the operation iterates through all participants of the relationship type. This is operations imposed on relationship schemas and different from those based on querying the classes of objects. Querying of objects of a class returns each object with the same type. However, querying participants of a relationship type returns each participant which may have different types. For example, querying the participants of relationship type take can be written as follow (Figure 5.6).

This example retrieves the name and type name for each participant in type take. If the type of the participant is required, x.type() should be used. Here, Participants is a system-defined type which contains information about participants for each relationship type, such as the names, the type names and the types of participants of the corresponding type. The query on type Participants allows the user to retrieve information of participants defined this type.

The query of relationship instances of a relationship class adopts the same syntax as that for an ordinary class. For example, when we want to retrieve all the instances of type take, the query in Figure 5.7 can be used to print out each faculty for a group students taking a set of courses in that faculty.

Querying of tuples or slices composed of each relationship instance can be achieved

```
foreach x: Participants
where type="take"
begin
    write("The participant is:", x.name(), x.typename());
end;
```

Figure 5.6: Querying Participants of Relationship Type take
Chapter 5. Relationships: Description of Associations among Objects

foreach x: take
begin
  ....
  write("Faculty: ", x.faculty());
end;

Figure 5.7: Querying Instances of Relationship Type take

foreach x: take
  foreach tp: tuple in x
  begin
    write("The tuple is:",
          tp.std.ObjectName(), tp.crs.ObjectName());
  end;

Figure 5.8: Querying Tuples Contained in Each Instance of Relationship Type take

in two ways. One is retrieving slices directly through slice types of certain relationship type. The operation has two steps: the first one gets the reference to each instance of a relationship class. The second step loops on all the slices of the current instance and the elements of the slices can be obtained through participant variables. For example, the tuples of a student taking a course can be retrieved by two embedded query operations as shown in Figure 5.8.

When slice types contain * as elements, the query of slices can be accomplished by adding another level of query statements. Using the above example again, if we use the slice type class instead of tuple, the query made on students is obtained (see Figure 5.9).

foreach x: take
  foreach tp: class in x
    foreach y: Student in tp.std
    begin
      write("The tuple is:",
            tp.std.ObjectName(), tp.crs.ObjectName());
    end;

Figure 5.9: Querying Elements of Instances of Relationship Type take
Chapter 5. Relationships: Description of Associations among Objects

foreach x: take
    foreach y: Student in x.std
    foreach z: Course in x.crs
    begin
        write("The tuple is:", y.ObjectName(), z.ObjectName());
    end;

Figure 5.10: Another Way of Querying Elements of Instances of Relationship Type

Another way to query slices is concerned with retrieving slices without explicitly defining the types. If this occurs, queries have to be executed through iteration on all the participants of the slices. For example, the query in Figure 5.8 can be rewritten as the one in Figure 5.10. Note that the return values of participants at the instance level are sets of objects. However, they are objects with participant types at the slice level if the participants of the corresponding slice types are not *.

5.7 Discussion

One of the major contributions of this research is to use the concept of slices to define the properties of relationships. However, it also introduces problems when variables are defined as slices types and slice types are regarded as ordinary object types. Here, a few things need to be clarified and more research has to be done in the future.

First of all, the semantics of a slice makes it more like a view or a query on a data type, as it is defined on a subset of relationship class according to the given conditions. In fact, this is not true because slices correspond to classes which contain all the tuples belonging to the slice type. Queries or views can be defined on slices.

Secondly, KOM defines slices as slice "types". The reason is that slice types behave like object types in most cases. For example, one can define variables of a slice type and these variables can be referred to in constraints, methods and triggers. In addition, queries can be defined on slice types in the same way as object types.

However, slice types are not object types since they can not fit into the type hierarchy. They are neither subtypes nor supertypes of relationship types. Besides, slice types are local types which only exist inside relationship types. They can not be referred to
outside relationship types as types in the object type hierarchy.

The inheritance rules for slices are required due to the inheritance of relationship properties and dynamic binding of variables defined as slice types. Slices defined in a type have different patterns from those inherited in the subtype according to the syntax. In other words, the subtype may have more participants than its supertype does, so, the inherited slices in the subtype automatically incorporate the extra participants into their patterns. The consistency between slices and the inherited ones is ensured by introducing the concept of compatible slice types. Compatible slice types basically define the same sets of tuples and thus can be replaced by the slice type it is compatible with.

Finally, inference rules may be possibly defined for the inheritance for both relationships and slices and enforced by the system. However, this is not explored in this thesis due to the limited time for research and implementation. More importantly, a weakness of slice types in KOM shows that it does not support a uniform and consistent way of specifying types, specially in the type hierarchy. Confusion may occur when slice types are sometimes used in the same way as object types and other times aren’t. A solution to this problem might be to use the conformity concept to construct the type hierarchy dynamically. In this case, the relationship of a type and its subtype is built at run-time according to the type definitions. In any case, the conformity definitions for relationship types and slice types need to be extended. Further research is required specially when we consider ordinary object types as unary relationships. Therefore, a generic framework for modelling relationships can be established for the specification of KOM. Currently, KOM is a loosely-typed OODB system. It can be enhanced into a strongly-typed OODB system when a consistent type hierarchy is built.
Chapter 6

Implementation of KOM and its Language

6.1 Introduction

In this chapter, we investigate the issues of implementation of the knowledge-based database system and its language. Research problems in this area include how to model data and knowledge, optimize the integration of AI and database systems and establish the run-time architecture. In detail, we confine our interests to the following issues: interface functions, transformation methods, constraint enforcement, trigger execution, transaction and rescue implementation, relationship construction and inheritance mechanism. The object-oriented programming language C++ plays the major role in the implementation of the KOM system and all underlying systems which contribute to the implementation are compatible with it. The implementation opts for a great deal of pre-processing, and the KOM syntax is defined by LR(1) grammars and operated by Yacc++. It generates a parser for the KOM language and a translator which transforms KOM programs into C++ ones. A run-time facility is also adopted to support functionality which cannot be handled at compile time, such as dynamic binding. The whole system has been established on a complete basis and can be used as a prototype for building knowledge-based database systems.

There are two basic systems which are used to implement KOM. One of them is the expert system Clips which is featured by its production rules and flexible interface with the C language. Clips is an expert system tool which was developed by the Software Technology Branch, NASA-Lyndon B. Johnson Space Center. It is designed
to facilitate the development of software to model human knowledge or expertise, and is also fully integrated with programming languages such as C and PASCAL. Clips can be invoked from C programs, to perform its rules and then return control back to the calling program. Likewise, procedural codes can also be defined as external functions and called from Clips' rules. When a rule is fired, the external code is executed to complete its task and return control to Clips when it finishes.

Clips has similar syntax to Lisp and uses several *define* constructors to specify knowledge. The basic constructors of Clips are *defrule* to define rules and *deffacts* to declare a list of facts. Other constructors are also provided by which more complicated facts are managed. For example, Clips adopts the object-oriented paradigm as part of the system. Specifications of objects and message passing are permitted, but unfortunately fail to connect with the rule-based system section. The implementation of KOM only makes use of the two basic constructors: *defrule* and *deffacts*.

Facts have free formed in Clips. This means that facts may have different lengths and be organized in a way which easily represents the content of real world. In Clips, a set of facts is grouped by *deffacts* followed by a name and a list of facts. They are used in reasoning by matching certain conditions of rules. For instance, the information of person John can be expressed as he is at age of 24, rich and employed as a manager (see Figure 6.1).

Clips provides many kinds of conditions and actions for rule specifications. The essential capabilities of rules are that they are allowed to add new facts or remove old facts from the system, in order to keep activating rules. Rule conditions are quite simple and usually consist of a set of matching facts or testing predicates. Rule actions are always functions which perform some tasks and cause new conditions to be satisfied. In general, a Rule has a form like the one in Figure 6.2.

This rule expresses that when the *initial-fact* is inserted or the system is reset,
(defrule rule-name
  (initial-fact)
  =>
  (printout t "Start Inference" crlf))

Figure 6.2: Clips Rules

the rule is activated and fired to print the string “Start Inference”.

The other underlying system used is the object-oriented database system Ontos which is compatible with C++. Ontos is an object database management system with a C++ class library interface. As a database, it provides a persistent storage facility for C++ objects. Basically, it allows objects denoted by C++ program variables to have a lifetime which are longer than the programs that create them. In addition, it allows C++ programs to retrieve persistent objects into their program variables. Ontos includes a variety of class definitions as well as database functionalities. It permits users to define object classes from the root of class hierarchy and persistent object classes must be inherited from the abstract class called Object. Basic database operations, such as create object with unique identifiers, delete objects, put objects into the database or get objects from the database, are supported by class Object. This class has a member function Name() which returns the object name for each reference in the class.

Query abilities for objects, types, aggregate classes are provided by the Ontos database system, too. This can be accomplished through a set of iterators which loop among objects in a class. The references of instances of the class are returned by the iterators and operations on the selected objects can be performed. Information of object specifications can be found from system-defined types such as class Type, Property and Procedure. The obvious advantages of Clips and Ontos are that they are compatible with C++ language and the interface between them can be easily implemented.

This chapter is organized as follow. In Section 6.2, the interface between Clips and Ontos is introduced and basic communication functions of the interface are listed. Section 6.3 addresses some important issues related to the implementation of constraints and triggers. Section 6.4 describes the strategies for realizing transactions and
rescues in KOM. In Section 6.5, relationships are constructed as an extension of the Ontos database system. Generic implementation methods of n-ary relationships under object-oriented paradigms are proposed. Due to its complexity, inheritance becomes one of the most difficult parts to be implemented in the KOM system. Approaches to implement inheritance rules and dynamic binding are discussed in Section 6.6. Finally, Section 6.7 creates a persistent knowledge base for storage and manipulation of both rules and facts used by KOM. Some general types are defined which can be extended by declaring subtypes of them.

6.2 Interface between Clips and Ontos

The interface between Clips and Ontos is implemented through C++ functions. In the C++ programs, Clips is invoked by several commands, such as initializing the environment, loading rules, resetting Clips and running rules. These are the four basic functions which must be called to fire Clips rules from any outside application programs. Also, there are some other functions provided by Clips which allow rules to access data from application programs. These functions are very useful to implement the interface, and the communication between Ontos and Clips is based on them. More information about Clips external functions can be found in [5].

There are some basic functions which are required to construct the interface between Clips and Ontos. Two aspects of operations are considered. One is how to perform database operations by Clips rules and another is how to run Clips rules by the Ontos database system. No matter which side of operations is concerned, some common functionalities must be included. The reason is that operations from both sides must be executed by Clips rules. For example, the rules can execute the database operations or the database operation can cause rule firing no matter where to start the execution.

The communications between Clips and Ontos are supported by the following functions:

- **Invoking Rules**: Several Clips functions are called to start the inference engine through the C++ interface.

  1. `InitializeCLIPS()` — It initializes the Clips system and must be called
prior to any other Clips functions.

2. **Load Constructs()** — It loads a set of constructs into the Clips database, including rules.

3. **Reset CLIPS()** — It resets the Clips environment for reasoning.

4. **Run CLIPS()** — It allows rules stored in the Clips database to be executed.

**Database Operations**: These include the operations performed on the Ontos database before and after the process of objects in a particular database. They are provided by the Ontos database management system.

1. **OOpenDatabase()** — Open the Ontos database. The database must be opened before it is used.

2. **OCloseDatabase()** — Close the Ontos database. The database requires to be closed after its uses.

3. **OTransactionStart()** — Start the Ontos transaction. The transaction has to be started before any operations are performed on the objects in the database.

4. **OTransactionAbort()** — Abort the Ontos transaction. All the update operations are aborted and the database remains unchanged.

5. **OTransactionCommit()** — Commit the Ontos transaction. The updates are accepted if the transaction is committed.

**Data Transformation**: This refers to the transformation of objects between Clips and Ontos. Two commands are provided by the interface and they are the load and save functions.

1. **OLoadInstances(class-name)** — Load the instances of a class from Ontos to Clips;

2. **OSaveInstances(class-name)** — Save the instances of a class from Clips to Ontos;

**Invoking Member Functions**: This interface makes it possible that Clips rules can execute the functions defined in an Ontos object type. Member functions are
invoked in rules by providing parameters such as the reference of the class, class name, method name and the restrictions of returned values or the values needed to be passed to the object. Three functions are defined for this purpose.

1. **OCallInt(parameters)** — Call the member function which returns integer value;

2. **OCallString(parameters)** — Call the member function which returns string value;

3. **OCallVoid(parameters)** — Call the member function which passes the values to attributes without returned values;

The functions defined in the interface are usually called in the action part of rules. An example of this is to load all the instances of class Lecturer into Clips:

```
(defrule load-instance-rule
  (initial-fact)
  =>
  (OLoadInstances Lecturer))
```

Operations on objects of Ontos classes can be written in a way that in the rule condition part, the class name is given and an instance pointer attached to that class is defined. For example, an object reference of the Lecturer class can be obtained in Clips by issuing the rule below. In fact, this rule iterates all the instances of class Lecturer since no restriction is put on the selection of objects.

```
(defrule process-lecturers
  (Lecturer ?lee)
  =>
  (...))
```

The actions taken on particular objects of class Lecturer are specified in the rule action part. The following is a complete rule which illustrates that a lecturer with particular object name is ?obj and satisfies constraint salaryCns requires new position. The conditions of the rule is to find such an object and the actions read the value and assign it as new position for that lecturer.
(defrule changeposition
  (Lecturer ?class_ptr)
  (Object-Name Lecturer ?obj)
  (test (eq (0CallString ?class_ptr Lecturer AName) (str-cat ?obj)))
  (test (eq (0CallString ?class_ptr Lecturer salaryCns) "TRUE"))
=>
  (printout t "Running Clips Rules ...changePosition" crlf)
  (bind ?position (readline) )
  (0CallVoid ?class_ptr Lecturer changePosition))

This is a typical example of rules which is generated by the KOM compiler. In the rule condition, it specified that the object is from the Lecturer class and has a name attached to variable ?obj. Two test statements are required in the rule's conditions that the first one helps to find the object which is the same as given in ?obj. The second test checks the constraint violation for that object. The test result is true when function salaryCns returns "TRUE". The actions of rules read new position and assign it back to the lecturer.

6.3 Transformation from KOM Programs to C++ Functions

The implementation of KOM consists of several steps. The first step is the interface described in the previous section. The interface functions plus Clips and Ontos C++ libraries consist of KOM library. A KOM compiler is generated by Yacc++ and Lex++. Programs with KOM statements are compiled by the KOM compiler and transformed into C++ programs. After this, the C++ programs are compiled by C++ compiler, linked with KOM C++ library, and then a running program is generated.

Basically, KOM types are directly transformed into Ontos class definitions. The attributes in a type correspond to a list of instance variables in the private section of an Ontos class. In addition, two methods are generated for each attribute in the class, by which the value of the attribute can be set or retrieved. The set and get methods are defined in the public section of the class. The rest of properties of the KOM type,
i.e., constraints and methods are translated into a set of methods defined in the public section of the class in order to be accessed by other types of objects. Constraints are converted into a list of logical conditions and they are evaluated through constraint methods. KOM methods are syntactically translated into Ontos member functions. Basically, it is one to one mapping between KOM and C++ statements and definitions, except that the foreach/forsome statements are translated into iteration operations based on Ontos iterator classes. The translation is done at compile-time. Examples of the transformation are listed in Appendix A.

Two classes are defined in Ontos to provide for the storage of rules. The Rule class is used to store each individual rule in KOM. In other words, each rule of KOM is linked with one instance of class Rule and an attribute of class Rule is defined to save the content of rules. The second class defined for rules is RuleSet which classifies rules according to their types. This class has aggregate type Set and is related to class Rule according to the type by which are defined rules. Rules belonging to one KOM type correspond to one instance of class RuleSet. When a type definition is compiled, an instance of class RuleSet is created with the identifier of the type’s name. After this, rule instances are inserted as members of that instance created for class Rule. The content of each rule is stored into the attribute of class Rule. The identifiers of rules in class Rule are their type names combined with rules’ own names. The set instance is retrieved from RuleSet when objects referring to the same type is operated. All rules containing in the set instance are loaded into Clips and waiting for firing.

6.4 Enforcement of Constraints

In KOM, the framework of optimization for constraint checking is based on a proposal presented in [75]. Basically, the enforcement of constraints is accomplished in two steps. The first step is to select all relevant constraints for an update. The second step validates the selected constraints with respect to all relevant objects. In the following, we investigate the appropriate approach to realize and optimize the constraint enforcement of KOM in terms of these two steps.

Manipulation of objects in a database system usually involves several types of oper-
ations, including creation, deletion, retrieval and modification of objects. The creation of an object can only cause the constraints defined for that object to be violated. If some of its attributes hasn't assigned yet, it will not violate the constraints according to the constraint semantics defined for that object. Constraints are re-evaluated when attribute values are assigned later. The deletion of objects is a little bit complex compared with creation operation, since it may result in constraints defined for other objects to be violated. In this case, the two groups of objects constitute one piece of semantic constraints, e.g., active constraints. The deletion of one partner requires re-evaluation of constraints for the other partner. The technique for evaluation of this type of constraints will be discussed later. The retrieval of objects from a database does not cause any evaluation of constraints.

Modification of objects is the major factor to cause constraints to be checked. Since constraint checking can easily become the bottleneck of the system, efficient enforcement must be investigated to avoid accessing and evaluating non-applicable objects. In addition, the system should try to completely avoid iterative checking, whenever possible.

In order to select relevant constraints to check after a database update, the KOM compiler collects a list of constraints which are affected by an attribute of a type. If the attribute is updated, the corresponding constraints are evaluated. The criteria to select relevant constraints are defined as follow. For an attribute, if there is a constraint restricting it this constraint is selected for that attribute. If the constraint is active, then it is also selected for those attributes declared in the constraint which are affected by it. For example, in Figure 3.1, attribute Salary in type Lecturer relates to constraint salaryCns and prjSalary, but constraint prjSalary is also chosen for attribute Fund of type Project.

Once all relevant constraints are collected for each attribute of each type, the next step is to generate efficient methods to optimize the access to objects that are concerned with the constraints. According to the definitions provided previously, the evaluation of constraints in KOM needs to be considered separately into three categories: passive, active and relationship constraints. They have some overlap between them, such as that constraints defined in a relationship type consist of both passive and active
Chapter 6. Implementation of KOM and its Language

constraints, as well as those defined for the participants. If the constraint contains participants, the constraint is active too. Different categories of constraints require different approaches to process the constraints. To simplify the evaluation of passive constraints, the basic, intra- and inter-constraints can be processed directly whenever an update of an attribute is involved with them. In other words, there is no particular optimization adopted for them. Since set-oriented restrictions always cause constraints of a set of objects to be checked, efficient approaches to optimize the evaluation should be provided. KOM incorporates a technique from [75] and utilizes aggregate information of maximum and minimum of a set of values. A set of heuristics is offered which is based on the comparison between the values before and after the update. For example, the constraint prjSalary only needs to be checked when the maximum fund among projects is decreased. Any other cases, such as the fund increased or funds except the maximum one decreased, do not require the evaluation of the constraint. Constructor implies and except are logical operators and do not belong to any categories described above. In fact, they are converted into the basic logical operations at compile-time.

The evaluation of active constraints needs to consider both the current object and those which affect it after updates. For the current object, active constraints are evaluated by using the same techniques as passive constraints. For the updates on the other party of objects, different strategies are adopted according to the specification of the constraints. First, we consider an active constraint with a foundby clause. When both the types of objects and the references to the affected objects are given, the updates of the attributes specified by affectedby will cause the active constraint to be evaluated only for the objects referred by those references. Second, if active constraints are specified only by affectedby clause, run-time optimization is required to determine which objects’ constraints need to be checked. Otherwise, all the objects of the current type have to check their constraints when the attribute of any object of the type specified in the affectedby clause is updated.

The run-time optimization of active constraints is achieved by defining additional attributes to the other type and dynamically assigning objects for that attribute. Take constraint stdAdvisor in Figure 3.4 as an example. An attribute called cns-stdAdvisor is generated for type Student at compile-time. When constraint stdAdvisor is evalu-
ated for an object of type Lecturer, attribute cns-stdAdvisor for those objects will be assigned to the objects of type Student which involve in the current constraint evaluation.

The evaluation of constraints defined in a relationship type is similar to the process of constraints defined in an ordinary type. However, constraints involving relationship participants are another story. When the attribute values of participants are updated, it may be that the participants can not attend the relationship instances anymore because constraints are violated. Thus, constraints defined for participants are regarded as active constraints. Participants are those objects which will affect constraints to be evaluated for the relationship instances.

In general, each constraint of a type corresponds to one method of an evaluation. The method is defined by having the constraint name and contains the information of attributes causing it to be checked and the logical operation to do the evaluation. The return type of a constraint method is Boolean. Constraint methods can be called independently by using their names.

### 6.5 Trigger Mechanism

Triggers are fired on event occurrences. Once an event occurs, conditions of rules activated by that event are tested. If the conditions are satisfied, the rules are fired and actions are taken. Since triggers have multiple functionalities and their semantics is quite complex, we divide our description of the implementation of triggers in KOM into the following issues: detecting event occurrence, storing both old and new values, realizing deferred rules, firing rules according to priorities, and creating run-time transactions.

Detecting event occurrences happens at either the beginning or the end of a method. Note that this is also valid for update operations since they are eventually achieved through set methods. In detail, each method in KOM is associated with three conditions, separately: (class name ?class-ptr), (Object-Name class name ?object-ptr) and (Before Action) or (After Action). These conditions are asserted at both the beginning of and the end of each method of a type. Examples of the generated rules
can be seen in Appendix A.2.5 and A.3.5.

Deferred rules in KOM need to be fired at the end of the transaction in which they are activated. In order to achieve this, both the transaction and the rule need to be remembered by the system. The system automatically assigns an internal name for each transaction. When a transaction is started, the name of it should be kept in a variable for later uses. After this, any deferred rules which are activated inside this transaction are kept in a queue. When the current transaction is completed, all the rules stored in the queue are retrieved and fired according to the order in which they are stored. In detail, each deferred rule is assigned an extra condition of the transaction name. At the end of each transaction, the fact called (Transaction-Name name ) is inserted. After this, Clips is called. Rules are fired if their conditions are satisfied. Here, name is the internal transaction name assigned by the system.

Run-time rule firing creates run-time transactions. Run-time transactions are used to protect constraints from being checked during rule firing. When an event occurs, a run-time transaction is begun. It is ended when rule firing is completed and control is returned back to the place following the event. If an event is an update operation, the transaction is protected for variables which modify object values. If the event is message catching, the transaction is started without protecting any objects. In this case, the transaction is defined for uniformly generating run-time rule model rather than real execution of transactions. In KOM, each fired rule corresponds to one transaction which starts with the first condition checking and ends after the last action is completed. Run-time transactions are different from statically defined ones in the sense that they depend on whether the conditions of rules are satisfied or not. If not, creations of run-time transactions fail.

6.6 Implementation of Transactions and Rescues

The implementation of transactions is fairly easy. First, a transaction in KOM is defined for a list of objects which are protected from constraint checking during the execution of statements inside that transaction. When a transaction is completed, objects which have no protection from any other transaction have their constraints
checked. If constraints are violated, control passes to the rescue statement of the innermost surrounding transaction or function for which the rescue statement is not already active. Execution continues until all the rescue statements are finished or, finally, the abort action is taken.

Second, rescue statements are implemented as functions which can only be called by the transactions or methods which define them. If no rescue statements are defined for that method, a default rescue statement is used. It is the exit(1); function in C++.

An internal attribute called counter is created for each object which keeps count of the number of transactions active for that object. The default value for the counter is zero. When a transaction begins, the counters for the objects referred to the variables in the variable list in the transaction statement are incremented. If the counter for an object is increased from 0 to 1, a copy of this object is created in case it needs to be restored by the undo operation. When the transaction is completed, the counters for the protected objects are decremented and if they become zero, the constraints for these objects are checked. In the event of an undo operation being required, the values in the copy are restored to the original. In any case, the copy can then be deleted.

6.7 Construction of Relationships

There are many ways of structuring relationships in object-oriented database systems. A quite reasonable approach adopted for systems built from scratch is to support independent logical constructors for describing relationships. Otherwise, extension of existing models must be made and relationship types are declared as subtypes of a generic relationship type which is usually implemented by several complex data structures. As a consequence, relationship types can be very difficult to implement if more than two participants, i.e., n-ary relationships, are considered. Even for binary relationships, limitations are imposed and basic features of object types such as constraints and methods are hardly supported.

A reasonable easier way of constructing relationships has been found for the implementation of KOM relationships. The basic idea is that when a relationship type is
Chapter 6. Implementation of KOM and its Language

compiled, the compiler generates a set of Ontos classes. In this case, each slice type which can be derived from the relationship type corresponds to one Ontos class and the attributes defined for that slice are the attributes of the class. Their relationship type is generated as an Ontos class, too. In this class, some essential attributes and methods are defined which include the attributes, constraints, methods and triggers defined for this relationship, as well as those generated for building the links between the classes of relationships and slices. Classes for slices are dependent classes of the relationship one and can not be accessed directly by users.

In the following, we use the examples shown in Section A.3 in Appendix A to explain how the approach works. The relationship type is transformed into an Ontos class which has the same name of that relationship type, such as class take. A set of attributes defined in the private section of class take are those with types of Set and the members of attributes are the instances of corresponding slice types. In addition, attributes defined for the relationship type rather than for slice types are also included in this section. At the same time, a set of variables is created in the public section of the class, which point to a set for each type of the slice in order that the instances of that slice type can be accessed directly. Variables are declared to refer to the sets of participants of the relationship type as well and the participants of one relationship instance can be retrieved by iterating the set referenced by the variable. Finally, a method called insert is generated which takes the input of tuples and assigns the tuples to the attributes and variables mentioned above. For example, in Appendix A.3., when a tuple (John, CSOS) is inserted into a relationship instance, it will be inserted as a member of pattern0 to pattern2 and John will be inserted as a member of set_std and CSOS as a member of set_crs. All these are done by function insert. In other words, the constructor of take creates the instances of relationships and the insert function is used to insert tuples into slices for each instance of the relationship. Assignment and retrieval of attributes for slice types require to know the participants of the relationship first. Once the slice is found, a method is called and the value of the attribute is assigned. Examples of assigning and retrieving attribute values of relationships and the generated C++ codes are shown in Figure 6.3 and Figure 6.4, respectively.
assignAttributes() -> void
begin
  local name: string;
  local link: take;
  local std1: Student;
  local crs1: Course;
  local mark: number;

  read(string, "Faculty Name: ", name);
  link:=find take(name);
  read(string, "Student Name: ", name);
  std1:=find Student(name);
  read(string, "Course Name: ", name);
  crs1:=find Course(name);
  read(number, "Mark: ", mark);
  link.AssignGrade(std1, crs1, mark);
end;

retrieveAttributes() -> void
begin
  local name:string;
  local link: take;
  local std1: Student;
  local no: number;

  read(string, "Faculty Name: ", name);
  link:=find take(name);
  read(string, "Student Name: ", name);
  std1:=find Student(name);
  no:=link.NoPassed(std1);
  write("The number of courses passed: ", no);
end;

Figure 6.3: Assign and Retrieve Values of Relationship Attributes
void assignAttributes()
{
    char* name; take* link; Student* std1; Course* crs1; int mark;
    cout<<"Faculty Name:"<<endl;
    name=new char[READ_VAR_LEN];
    cin>>name;
    link=(take*) OC_lookup(name);
    cout<<"Student Name:"<<endl;
    name=new char[READ_VAR_LEN];
    cin>>name;
    std1=(Student*) OC_lookup(name);
    cout<<"Course Name:"<<endl;
    name=new char[READ_VAR_LEN];
    cin>>name;
    crs1=(Course*) OC_lookup(name);
    cout<<"Mark:"<<endl;
    cin>>mark;
    link->AssignGrade(std1,crs1,mark);
}

void retrieveAttributes()
{
    char* name; take* link; Student* std1; int no;
    cout<<"Faculty Name:"<<endl;
    name=new char[READ_VAR_LEN];
    cin>>name;
    link=(take*) OC_lookup(name);
    cout<<"Student Name1:"<<endl;
    name=new char[READ_VAR_LEN];
    cin>>name;
    std1=(Student*) OC_lookup(name);
    no=link->NoPassed(std1);
    cout<<"The number of courses passed:"<<(no)<<endl;
}

Figure 6.4: Generated C++ Codes for the assignAttribute() and retrieveAttribute() Methods
Chapter 6. Implementation of KOM and its Language

The second part of the implementation deals with the slice operations. Each slice has one corresponding class which holds the attributes defined for that slice. In the public section of the class, functions to access and to assign the attributes are defined. In addition to this, variables which contain the participants in the slices are declared. The operations including creating slice instances and inserting objects to participants are done by the insert function declared in the relationship class. Finally, an insert function is also defined for the slice classes which insert tuples into slice instances. Operations and Queries can be defined on the slices, participants of relationships or attributes of relationships directly. Examples of them can be seen in Figure 6.5, 6.6 and 6.7.

Note that the generic relationship type Relationship is not implemented at the current stage. Some essential operations of relationships, such as deletions of instances, are directly adopted from functions provided by the Object class in Ontos.

6.8 Inheritance Mechanism

The inheritance of KOM is implemented by the assistance of C++ inheritance mechanism. In C++, inheritance is achieved by support of derived classes and virtual functions. A derived class is defined by declaring it as a subtype of an existing type. In a class definition, inherited methods can be overwritten by declaring them as virtual functions. The implementation bodies of these methods can be rewritten according to the requirements of objects of the subclass. Using the same approach, an attribute can be redefined in a subclass by declaring it as having a subtype of its original one. Basically, the inheritance rules supported by C++ are consistent with those provided by KOM. This greatly simplifies the procedure of implementing inheritance mechanism of KOM.

Two major issues need to be considered in the implementation of KOM inheritance mechanism, which are the inheritance of constraints and triggers. The enforcement of constraints is realized by a set of methods which implement each individual constraint separately. This helps the implementation of overwritten constraints by using virtual functions. If constraints are redefined in the subtypes, the methods are declared as
insertSlices() -> void
begin
  local name: string;
  local link: take;
  local std1: Student;
  local crs1: Course;
  read(string, "Faculty Name:", name);
  link:=find take(name);
  read(string, "Student Name1:", name);
  std1:=find Student(name);
  read(string, "Course Name2:", name);
  crs1:=find Course(name);
  link.insert(std1, crs1);
end;

queryRelationships() -> void
begin
  local name: string;
  local mark: number;
  foreach x: take
  begin
    name=x.Name();
    write("Faculty Name:", name);
    foreach y: tuple in x
      begin
        name=y.std.Name();
        write("Student Name:", name);
        name=y.crs.Name();
        write("Course Name2:", name);
        mark=y.grade();
        write("Course Grade:", mark);
      end;
    end;
  end;
end;

Figure 6.5: Insert Slices and Query of Slices and Relationship Instances
void insertSlices()
{
    char* name;
    take* link;
    Student* std1;
    Course* crs1;
    cout<<"Faculty Name:"<<"\n";
    name=new char[READ_VAR_LEN];
    cin>>name;
    link=(take*) OC_lookup(name);
    cout<<"Student Name1:"<<"\n";
    name=new char[READ_VAR_LEN];
    cin>>name;
    std1=(Student*) OC_lookup(name);
    cout<<"Course Name2:"<<"\n";
    name=new char[READ_VAR_LEN];
    cin>>name;
    crs1=(Course*) OC_lookup(name);
    link->insert(std1,crs1);
}
void queryRelationships()
{
    char* name;
    int mark;

    // begin Instance Iterator
    Type *tp1;
    tp1=(Type*) OC_lookup("take");
    InstanceRefIterator anInstance1(tp1);
    take* x;

    while (anInstance1.moreData())
    {
        x=(take*)(Entity*)anInstance1();
        name=x->Name();
        cout<<"Faculty Name:"<<(name)<<"\n";

        // begin Instance Iterator
        SetIterator anInstance1(x->pattern0);
        tuple* y;

        while (anInstance1.moreData())
        {
            y=(tuple*)(Entity*)anInstance1();
            name=((Student *)y->std.Binding(y))->Name();
            cout<<"Student Name:"<<(name)<<"\n";
            name=y->crs->Name();
            cout<<"Course Name:"<<(name)<<"\n";
            mark=y->grade();
            cout<<"Course Grade:"<<(mark)<<"\n";
        }

        // begin Instance Iterator
        SetIterator anInstance2(x->pattern1);
        allcourses* z;

        while (anInstance2.moreData())
        {
            z=(allcourses*)(Entity*)anInstance2();
            mark=z->nopassed();
            cout<<"Number of Courses Passed:"<<(mark)<<"\n";
        }
    }
}

Figure 6.7: Generated C++ Codes for the queryRelationships() Method
virtual functions and overwritten by calling the old ones in their bodies. This can be achieved at run-time through the retrieval of the properties of supertypes. The inheritance of rules is achieved by either adding rules from supertypes or completely overwritten them in the subtypes.

The inheritance of relationships are complicated due to the consideration of slices. First, the class generated for the subtype of a relationship is a subclass of the one for the supertype. Since tuples for the subtype may have more participants than the supertype does, the insert function has to be a virtual method which can be overwritten in the subclass. Second, the slice classes derived for the subtype are subclasses of those slice classes defined for the supertype, but the insert function and variables of participants need to be overwritten. If the slice types have no supertypes, they are regarded as subtypes of Object. At this stage, the generic types for slice types, participants and relationship types are not implemented.
Chapter 7

Conclusion

7.1 Summary

The aim of the research described in this thesis is to enhance functionality of databases based on object-oriented technology. A primary contribution of the work is the uniformity: standard features of OODB’s, relationships and knowledge of triggers and constraints are integrated into one uniform representation. On top of this, possible extensions and enhancement are achieved. The major contributions of the research are:

- Constraints are declared in object type definitions and restricted relationships among objects are allowed to be specified. The difference between KOM and other systems is that it supports not only intra-constraints for the current object, but also constraints for related objects. Most of the systems support constraint checking only at certain places such as where objects of the constraints are defined. If the constraints are related to other objects and those objects are not updated, inconsistency may occur and the system cannot detect the potential errors. KOM solves this problem by introducing active constraints. The system keeps records of the related objects so that the objects’ constraints can be checked no matter which party is updated. The order of ordinary (passive) constraint checking is precisely defined, to eliminate one possible source of non-determinism. However, we do not propose a good solution for the nondeterminism due to active constraint checking order, but leave it as a research problem.
Chapter 7. Conclusion

- Triggers are the other feature provided by object types. In KOM, triggers are fired as rules and only react to events. Three types of events are supported: updates of object's states, message catching and constraint violation. The third one may be regarded as a composite condition of the first two events: updates of objects and method invocation (constraint methods). Logical operations can be defined for trigger conditions and they can be events or any other conditions, even selections of methods through their parameters. Actions of triggers can be taken before events or after events upon user's request. If transactions are defined for multiple object updates, then the action can be deferred until just before the transaction ends. Unlike most active databases, local variables can be defined at the beginning of the trigger section and shared by all triggers in both their conditions and actions. Besides, the actions of triggers can be implemented by any programming statements provided by the language. This greatly increases the flexibility and power of triggers.

- Transactions are supported at the conceptual level. This permits objects to be temporarily protected from constraint checking when a critical sequence of updates is required. Strategies on controlling and implementing nested transactions are given. The execution semantics for both rules and transactions are introduced.

- A major achievement of KOM is the development of a theory to represent and implement n-ary relationships. The semantics of relationships is elaborated and the concepts of slices and slice types are proposed. In the KOM context, a relationship instance is not a simple tuple but a set of tuples which can have their own properties according to the classification of tuples. New constructors are introduced and the links between relationship instances and the slices are established. Properties of slices are specified in the relationship type which they belong to.
7.2 Future Research

In the short term, KOM can be refined by solving the problems related to active constraints and slices. The specification and implementation of active constraints should be improved. A more meaningful approach for the constraint checking orders is required. Once this is achieved, a model for transactions needs to be proposed. In this model, places where constraints are checked, methods are executed and rules are fired should be precisely specified. The consequences of transaction operations, such as commit, abort or roll back must be defined.

Issues related to slices include how to define slice types as real object types, where they can fit into type hierarchy and what the extra features or semantics of slices are (if there are any). We leave these issues open for further research.

In the long term, the data model and its language provided in this thesis can be refined and expanded to accommodate other functionalities. First of all, a complete object model should be established, which will describe all the objects as relationships. In this case, if only one participant is defined for the relationship type, then it is exactly the same as ordinary object types. Under this scheme, only objects exist, which can be slices, relationship instances or ordinary objects. New concept or rules must be introduced to define the type hierarchy. Once the theory is established, no confusion or inconsistency will exist.

The second consideration is to implement KOM with a reflective architecture. Reflection is the process of reasoning about and acting upon one’s self. It is different from conventional computational systems in the sense that it solves problems by operating on data representing the structural and computational aspects of the system itself and returning information regarding the course of computation. In the object-oriented paradigm, it expresses the knowledge and information about the system itself at meta-level and this knowledge can be processed in the same way as ordinary objects are. Several system-defined types need to be defined at meta-level and they include information such as strategies of message passing, run-time system control, organization of type hierarchy and specifications of object types. This extension to the system can be easily done through defining subtypes of pre-existing types.
Appendix A

Examples of KOM Programs

A.1 Introduction

This chapter contains a collection of KOM programs and the generated C++ code. The purpose of it is to help readers to understand KOM design and possibly to write their own programs.

KOM type definition are specified in two files. One is the .def file which provides the specification of a type. It is translated into a .h file in C++. The other is the .fun file which contains the implementations of the methods for that type. The general functions of applications are also given in .fun files. They are translated into .c files in C++. When triggers are defined in a type, Clips rules are generated and stored in .rule files. Some auxiliary definitions and programs are generated when necessary.

The compile command is called kom and has syntax which is very close to cc. The -o option provides the name for the executable file.

kom [-o objectname] filename [filename [filename] ...].

In the following, we mainly discuss how to write .def and .fun files and what the generated .h and .c files look like. Two sections are included which give the type definitions of Lecturer and take separately.

A.2 Type Lecturer Definition

According to what we have in Figure 3.1 in Chapter 3, the following two files are required: lecturer.def and lecturer.fun, to define a complete type of lecturers. Four
files are generated after they are compiled: lecturer.h lecturer.c lecturer.rule lecturerfunc.c.

A.2.1 The lecturer.def File

type Lecturer isa Object begin
(attributes
   Employment: string;
   Salary: number;
   Position: string;
(constraints
   empCns: Employment() = "employed"
   or Employment() = "unemployed";
   salaryCns:
      Salary >= 30000 and Salary <= 76000 and
      (Position = "Professor"
      implies foreach Lct: Lecturer
         where Lct.Position() <> "Professor"
         enforce Lct.Salary() < Salary)
      and (Position <> "Professor"
      implies foreach Lct: Lecturer
         where Lct.Position() = "Professor"
         enforce Lct.Salary() > Salary)
except Position = "Volunteer"
   or Position = "Emeritus Staff";
(methods
   Lecturer(name: string, employment: string, 
   salary: number, position: string);
   newSalary(salary: number)->void;
   newPosition(position: string)->void;
(triggers
   changePosition:
      if update(Salary) and (not salaryCns())
      after-action
         position: string;
         read(string, "new Position", position);
         newPosition(position);
         endaction;
endrule;
   changeEmployment:
      if message(newPosition)
      after-action
         emp: string;
         read(string, "new State of Employment", emp);
A.2.2 The lecturer.fun File

predefined "lecturer.def";

Lecturer::Lecturer(name: string, employment: string, salary: number, position: string):(name)
begin
  Employment(employment);
  Salary(salary);
  Position(position);
end;

Lecturer::newPosition(position: string)->void
begin
  Position(position);
end;

Lecturer::newSalary(salary: number)->void
begin
  write("Salary is ", salary);
  transaction this do
    Salary(salary);
    if (not(Lecturer_Constraint()))
      then begin
        write("New Salary Violate Constraints.");
        write("Salary is:", salary);
        end;
    read(number, "Salary Again:", salary);
    Salary(salary);
  endtransaction;
end;

A.2.3 The lecturer.h File

 ifndef Def_lec.h
#define Def_lec.h
#endif
Appendix A. Examples of KOM Programs

#include "Object.h"
#include "Reference.h"
#include "Set.h"
#include "Directory.h"
#include "Iterator.h"

class Lecturer: public Object
{
    private:
        char* iEmployment;
        int iSalary;
        char* iPosition;
    friend:
        int counter;
    public:
        Boolean empCns(char *attribute_name);
        Boolean salaryCns(char *attribute_name);
        Boolean Lecturer_Constraint(char* attribute_name);
        Boolean forcondition_Lecturer0();
        Boolean forcondition_Lecturer1();
        Lecturer(APL *);
        Lecturer();
        Type* getDirectType();
        char* LecturerName();
        void Employment(char* x);
        char* Employment();
        void Salary(int x);
        int Salary();
        void Position(char* x);
        char* Position();
        Lecturer(char* name, char* employment,
            int salary, char* position);
        void newSalary(int salary);
        void newPosition(char* position);
        void changePosition();
        void changeEmployment();
};
#endif

A.2.4 The lecturer.c File

#include "stdio.h"
#include "string.h"
#include "stream.h"
#include "Object.h"
#include "Set.h"
Appendix A. Examples of KOM Programs

```c
#include "Directory.h"
#include "Iterator.h"
#include "Database.h"

#define true TRUE
extern void clipsmain(char *);
extern void save_fact(char *);
extern Boolean optimized(char* class_name, char* constraint_name,
                         char* attribute_name);

#include "globalfun.h"
#include "lecturer.h"

extern int Lecturer_Constructor;

Lecturer::Lecturer(APL* theAPL): (theAPL){}
Type* Lecturer::getDirectType()
    {return (Type*)DC_lookup("Lecturer");}
Lecturer::Lecturer()
    {initDirectType((Type*)DC_lookup("Lecturer"));}

Boolean Lecturer::empCns(char* attribute_name)
{
    if (optimized("Lecturer", "empCns", attribute_name))
        return TRUE;
    if (strcmp(Employment(), "employed")
        || strcmp(Employment(), "unemployed")
        return TRUE; else return FALSE;
}

Boolean Lecturer::salaryCns(char* attribute_name)
{
    if (optimized("Lecturer", "salaryCns", attribute_name))
        return TRUE;
    if (Salary() >= 30000 && Salary() <= 76000 &&
        (!(strcmp(Position(), "Professor")
            || forcondition_Lecturer0())
        && (!(strcmp(Position(), "Professor")
            || forcondition_Lecturer1())
        || !strcmp(Position(), "Volunteer")
        || !strcmp(Position(), "Emeritus Staff"))
        return TRUE; else return FALSE;
}

Boolean Lecturer::Lecturer_Constraint(char* attribute_name=NULL)
{
    if (empCns(attribute_name) && salaryCns(attribute_name))
```
return TRUE;
else return FALSE;
}
void Lecturer::changePosition()
{
    if ((!salaryCns()))
    {char* position;
    cout<<"new Position"<<"\n";
    position=new char[READ_VAR_LEN];
    cin>>position;
    newPosition(position);
    }
}
void Lecturer::changeEmployment()
{
    if (true)
    {char* emp;
    cout<<"new State of Employment"<<"\n";
    emp=new char[READ_VAR_LEN];
    cin>>emp;
    Employment(emp);
    }
}
char* Lecturer::LecturerName()
{ return Name(); }
void Lecturer::Employment(char* x)
{
char fact[100];
if (!Lecturer_Constructor)
{ save_fact("UPDATE Lecturer Employment");
save_fact("Before Action");
strcpy(fact, "Object-Name ");
strcat(fact, "Lecturer");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("lecturer.rule");
}
iEmployment=new char[strlen(x)+1];
strcpy(iEmployment, x);
if (!Lecturer_Constructor)
{ save_fact("UPDATE Lecturer Employment");
save_fact("After Action");
strcpy(fact, "Object-Name ");
strcat(fact, "Lecturer");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("lecturer.rule");
if (!counter){
if (!Lecturer_Constraint("Employment"))
cout<<"Violation on Updating Type:Lecturer, Attribute:Employment"<<endl; }
}

char* Lecturer::Employment()
{ return iEmployment; }

void Lecturer::Salary(int x)
{
char fact[100];
if (!Lecturer_Constructor)
{ save_fact("UPDATE Lecturer Salary");
save_fact("Before Action");
strcpy(fact, "Object-Name ");
strcat(fact, "Lecturer");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("lecturer.rule");}
iSalary=x;
if (!Lecturer_Constructor)
{ save_fact("UPDATE Lecturer Salary");
save_fact("After Action");
strcpy(fact, "Object-Name ");
strcat(fact, "Lecturer");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("lecturer.rule");}
if (!counter){
if (!Lecturer_Constraint("Salary"))
cout<<"Violation on Updating Type:Lecturer, Attribute:Salary"<<endl; }
}

int Lecturer::Salary()
{ return iSalary; }

void Lecturer::Position(char* x)
{
char fact[100];
if (!Lecturer_Constructor)
{ save_fact("UPDATE Lecturer Position");
save_fact("Before Action");
strcpy(fact, "Object-Name ");
Appendix A. Examples of KOM Programs

```c
strcat(fact, "Lecturer");
strcat(fact, " ");
strcat(fact, Name());
save_fact(fact);
clipsmain("lecturer.rule");

iPosition=new char[strlen(x)+1];
strcpy(iPosition, x);

if (!Lecturer_Constructor)
{ save_fact("UPDATE Lecturer Position");
  save_fact("After Action");
  strcpy(fact, "Object-Name ");
  strcat(fact, "Lecturer");
  strcat(fact, " ");
  strcat(fact, Name());
  save_fact(fact);
  clipsmain("lecturer.rule");
  if (!counter){
    if (!Lecturer_Constraint("Position"))
      cout<<"Violation on Updating Type:Lecturer,
      Attribute:Position"
<<endl; }
}

char* Lecturer::Position()
{ return iPosition; }

Boolean Lecturer::forcondition_Lecturer0()
{
  // begin Instance Iterator
  Type *tp;
  tp=(Type*) OC_lookup("Lecturer");
  InstanceRefIterator anInstance(tp);
  Lecturer* Lct;

  while (anInstance.moreData())
  { Lct=(Lecturer*) (Entity*)anInstance();
    if (strcmp(Lct->Position(), "Professor"))
      if (!!(Lct->Salary())<Salary()))
        return FALSE; }
  return TRUE;
}

Boolean Lecturer::forcondition_Lecturer1()
{
  // begin Instance Iterator
  Type *tp;
  tp=(Type*) OC_lookup("Lecturer");
  InstanceRefIterator anInstance(tp);
```
Lecturer* Lct;

while (anInstance.moreData())
    { Lct=(Lecturer*) (Entity*)anInstance();
      if (!strcmp(Lct->Position(), "Professor"))
        if (!(Lct->Salary()>Salary()))
          return FALSE; }
      return TRUE;
}

A.2.5 The lecturer.rule File

;Begin Trigger Section
(defrule load_instances
 (initial-fact)
 =>
 (OLoadInstances Lecturer))

(defrule changePosition
 (Lecturer ?class-ptr)
 (Object-Name Lecturer ?object-ptr)
 (test (eq (OCallString ?class-ptr Lecturer LecturerName)
           (str-cat ?object-ptr)))
 (UPDATE Lecturer Salary)
 (After Action)
 =>
 (printout t "Running Clips Rules ...changePosition" crlf)
 (OCallVoid ?class-ptr Lecturer changePosition))

(defrule changeEmployment
 (Lecturer ?class-ptr)
 (Object-Name Lecturer ?object-ptr)
 (test (eq (OCallString ?class-ptr Lecturer LecturerName)
           (str-cat ?object-ptr)))
 (MESSAGE Lecturer newPosition)
 (After Action)
 =>
 (printout t "Running Clips Rules ...changeEmployment" crlf)
 (OCallVoid ?class-ptr Lecturer changeEmployment))

;End Trigger Section

A.2.6 The lecturerfun.c File

#include "stdio.h"
#include "string.h"
#include "stream.h"
#include "Object.h"
Appendix A. Examples of KOM Programs

#include "Set.h"
#include "Directory.h"
#include "Iterator.h"
#include "Database.h"

#define true TRUE
extern void clipsmain(char *);
extern void save_fact(char *);
extern void copy_object(Type*, Object*);
extern void remove_object(Type*, Object*);

#include "Transaction.h"
#include "globalfun.h"
#include "lecturer.h"
int Lecturer_Constructor=0;

Lecturer::Lecturer(char* name, char* employment,
int salary, char* position):(name)
{
  Lecturer_Constructor=1;
  initDirectType((Type*)OC_lookup("Lecturer"));

  Lecturer* obj;
  obj=(Lecturer*) OC_lookup(name);
  if (!obj)
  {
    cerr<<"ERROR: Object exists"<<endl; exit(1);
  } counter++;
  if (counter==1) copy_object(getDirectType(), this);
  Employment(employment);
  Salary(salary);
  Position(position);

  if (!Lecturer_Constraint())
    cout<<"Cannot Create Object due to Constraint Violation\n";
  else {cout<<"The Object is Created."
    Name(name);
    cout<<"********************"<<endl;
    Lecturer_Constructor=0;
    counter--;
  if (!counter) remove_object(getDirectType(), this);
  }
void Lecturer::newPosition(char* position)
{
char fact[100];

if (!Lecturer_Constructor)
{
  save_fact("MESSAGE Lecturer newPosition");
  save_fact("Before Action");
}
Appendix A. Examples of KOM Programs

```c
strcpy(fact, "Object-Name ");
strcat(fact, "Lecturer");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("lecturer.rule");

{ Position(position); }
if (!Lecturer_Constructor)
{save_fact("MESSAGE Lecturer newPosition");
save_fact("After Action");
strcpy(fact, "Object-Name ");
strcat(fact, "Lecturer");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("lecturer.rule");}
if (!counter){
if (!Lecturer_Constraint())
cout<<"Violation on Updating Type:Lecturer"<<endl; }
}

void Lecturer::newSalary(int salary)
{
    char fact[100];

    if (!Lecturer_Constructor)
    { save_fact("MESSAGE Lecturer newSalary");
        save_fact("Before Action");
        strcpy(fact, "Object-Name ");
        strcat(fact, "Lecturer");
        strcat(fact, " "); strcat(fact, Name());
        save_fact(fact);
        clipsmain("lecturer.rule");}

    { cout<<"Salary is "<<(salary)<<(\n"n";

    /* Transaction Begin */
counter++;
if (counter==1) copy_object(getDirectType(), this);
Salary(salary);
if (!this->Lecturer_Constraint())
{ cout<<"constraint violation"<<(\n"n";
    cout<<"Salary is:"<<(salary)<<(\n"n"; }
cout<<"Salary Again:"<<(\n"n";
cin>>salary; Salary(salary);
/* Constraint Checking */
if (!this->Lecturer_Constraint())
cout<<"Constraint Violated\n";
```
Appendix A. Examples of KOM Programs

counter--;  
if (!counter) remove_object(getDirectType(), this);  
/* Transaction End */  
if (!Lecturer_Constructor)  
{save_fact("MESSAGE Lecturer newSalary");  
save_fact("After Action");  
strcpy(fact, "Object-Name ");  
strcat(fact, "Lecturer");  
strcat(fact, " ");  
strcat(fact, Name());  
save_fact(fact);  
clipsrnain("lecturer.rule");}  
if (!counter)){  
if (!Lecturer_Constraint())  
cout<<"Violation on Updating Type:Lecturer"<<endl; }  

A.3 Relationship Type take Definition

Relationship take is defined by two files in KOM: take.def and take.fun. After compiling them, two extra files are generated in addition to the four types of files mentioned before. The C++ files created for relaionship type take are: take.h, take.c, take.rule, takefun.c, take_slices.h and take_slices.c. Since type take relates the student and the course types, we also provide the simplified KOM type definitions for them. In the following, a list of files are given either in KOM or C++.

A.3.1 The take.def File

predefined "student.def";  
predefined "course.def";

relationship take(std: Student, crs: Course) isa Object  
begin  
slices  
  allcourses: (std, *);  
tuple: (std, crs);  
aclass: (*, crs);

attributes  
facultynarne: string;  
grade of tuple: number;  
nopassed of allcourses: number;
enrollees of a class: set-of Student;
university: string;

constraints
Undergraduate:
   foreach s: Student in std
   enforce s.StudentNo() > 89000;

StudentGrade:
   foreach x: tuple
   enforce x.grade<=10 and x.grade>=0;

methods
take(facultyname: string);
GetGrade(student: Student, course: Course)->number;
AssignGrade(student: Student, course: Course,
            mark: number)->void;
NoPassed(student: Student)->number;
University(univ: string)->void;

triggers
monitor:
   if message(AssignGrade)
   after-action
      name:string;
      std:Student;
      read(string, "Student Name:", name);
      std:=find Student(name);
      NoPassed(std);
      endaction;
   endrule;
end;

A.3.2 The take.fun File

predefined "take.def";

take::take(fac: string):(fac)
begin
   facultyname(fac);
end;
take::GetGrade(student: Student, course: Course)->number
begin
Appendix A. Examples of KOM Programs

x: tuple;

x:=find tuple(student, course);
return x.grade();
end;

take::AssignGrade(student: Student, course: Course, mark: number)->void
begin
x: tuple;

x:=find tuple(student, course);
x.grade(mark);
end;

take::NoPassed(student: Student)->number
begin
x: allcourses;
y: tuple;
np: number;

np:=0;
x:=find allcourses(student, *);
foreach z: Course
begin
  y:=find tuple(student, z);
  if (y.grade()>=5) then begin np:=np +1; end;
end;

x.nopassed(np);
return np;
end;

A.3.3 The take.h File

#ifndef Def_take.h
#define Def_take.h

#include "Object.h"
#include "Reference.h"
#include "Set.h"
#include "Directory.h"
#include "Iterator.h"

#include "student.h"
#include "course.h"
#include "take_slices.h"

class take: public Object {
    
    private:
    Set* grade;
    Set* nopassed;
    Set* enrollees;
    char* ifacultyname;
    char* iuniversity;

    friend:
    
    int    counter;

    public:
    Boolean Undergraduate(char* attribute_name);
    Boolean StudentGrade(char* attribute_name);
    Boolean take_Constraint(char* attribute_name);
    Boolean forcondition_take0();
    Boolean forcondition_take1();
    take(APL *);
    take();
    Type* getDirectType();
    char* takeName();
    void facultyname(char* x);
    char* facultynname();
    void university(char* x);
    char* university();
    take(char* facultynname);
    int  GetGrade(Student* student, Course* course);
    void AssignGrade(Student* student, Course* course, int mark);
    int  NoPassed(Student* student);
    void University(char* univ);
    void monitor();
    Set* pattern1;
    Set* pattern0;
    Set* pattern2;
    Set* set_std;
    Set* set_crs;

    virtual void insert(Student* ins_std, Course* ins_crs);
    
};

#endif

A.3.4 The take.c File

#include    "stdio.h"
#include    "string.h"
#include "stream.h"
#include "Object.h"
#include "Set.h"
#include "Directory.h"
#include "Iterator.h"
#include "Database.h"

#define true TRUE
extern void clipsmain(char *);
extern void save_fact(char *);
extern Boolean optimized(char *class_name, char *constraint_name, char *attribute_name);

#include 
#include
extern int take_Constructor;
take::take(APL* theAPL):(theAPL){}
Type* take::getDirectType()
  {return (Type *)OC_lookup("take");}
take::take()
  {initDirectType((Type *)OC_lookup("take");)}
Boolean take::Undergraduate(char* attribute_name)
  { if (optimized("take", "Undergraduate", attribute_name))
    return TRUE;
    if (forcondition_take0())
    return TRUE; else return FALSE; }
Boolean take::StudentGrade(char* attribute_name)
  { if (optimized("take", "StudentGrade", attribute_name))
    return TRUE;
    if (forcondition_take1())
    return TRUE; else return FALSE; }
Boolean take::take_Constraint(char* attribute_name)
  { if (take_Constructor) return TRUE;
    if (Undergraduate(attribute_name)
      && StudentGrade(attribute_name))
    return TRUE; else return FALSE; }
void take::monitor()
  { if (true)
    {char* name;
     Student* std;
     cout<<"Student Name:"<<"\n"
     name=new char[READ_VAR_LEN];
     cin>>name;
     std=(Student*) OC_lookup(name);
     NoPassed(std); }
Appendix A. Examples of KOM Programs

```c
void take::insert(Student* ins_std, Course* ins_crs)
{
    Type *tp;
    char xxx[250];

    if (set_std==NULL) { tp=(Type*)OC_lookup("Student");
        set_std=new Set(tp);} 
    set_std->Insert(ins_std);
    if (set_crs==NULL) { tp=(Type*)OC_lookup("Course");
        set_crs=new Set(tp);} 
    set_crs->Insert(ins_crs);

    take_pattern0 *thisPattern0;
    take_pattern1 *thisPattern1;
    take_pattern2 *thisPattern2;
    int flag=0;

    if (pattern2==NULL)
    { tp=(Type*)OC_lookup("take_pattern2");
        pattern2=new Set(tp);
        pattern2->Name(cat(Name(), "take_pattern2"));} 
    SetIterator iter2(pattern2);
    while (iter2.moreData())
    { thisPattern2=(take_pattern2*)(Entity*)iter2();
        if (thisPattern2->crs.Binding(thisPattern2) == ins_crs)
        {thisPattern2->std->Insert(ins_std);
            flag=1; break;}}
    if (!flag) {strcpy(xxx, cat(ins_std->Name(), ins_crs->Name()));
        thisPattern2=new take_pattern2(cat("take_pattern2", xxx));
        tp=(Type*)OC_lookup("Student");
        thisPattern2->std=new Set(tp);
        thisPattern2->std->Insert(ins_std);
        if (!ins_crs)
            cerr<<"Null Student"<<endl;
        thisPattern2->crs.Reset(ins_crs, thisPattern2);
        pattern2->Insert(thisPattern2); } 
    flag=0;

    if (pattern1==NULL)
    { tp=(Type*)OC_lookup("take_pattern1");
        pattern1=new Set(tp);
        pattern1->Name(cat(Name(), "take_pattern1"));}
    SetIterator iter1(pattern1);
    while (iter1.moreData())
    { thisPattern1=(take_pattern1*)(Entity*)iter1();
        if (thisPattern1->std.Binding(thisPattern1) == ins_std)
```
Appendix A. Examples of KOM Programs

```c
{thisPattern1->crs->Insert(ins_crs);
flag=1; break; }
if (!flag) strcpy(xxx, cat(ins_std->Name(), ins_crs->Name()));
thisPattern1=new take_pattern1(cat("take_pattern1", xxx));
if (!ins_std) cerr<<"Null Student"<<endl;
thisPattern1->std.Reset(ins_std, thisPattern1);
pattern1->Insert(thisPattern1);
if (!ins_crs) cerr<<"Null Student"<<endl;
thisPattern1->crs->Insert(ins_crs);} flag=0;
if (pattern0==NULL) {
  tp=(Type*)OC_lookup("Course");
  thisPattern1->crs=new Set(tp);
  thisPattern1->crs->Insert(ins_crs);
  flag=0;
  if (!counter) {
    if (!take_Constraint())
      cout<<"Violation on Updating Type:take"<<endl; }
}
char* take::takeName()
{ return Name(); }
void take::facultyname(char* x)
{ char fact[100];
  if (!take_Constructor)
    { save_fact("UPDATE take facultyname");
```
save_fact("Before Action");
strcpy(fact, "Object-Name ");
strcat(fact, "take");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("take.rule");

ifacultynname=new char[strlen(x)+1];
strcpy(ifacultynname, x);
if (!take_Constructor)
{ save_fact("UPDATE take facultynname");
save_fact("After Action");
strcpy(fact, "Object-Name ");
strcat(fact, "take");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("take.rule");

if (counter){
    if (! take_Constraint())
        cout<<"Violation on Updating Type:take,
        Attribute:facultynname"<<endl; }
}

char* take::facultynname()
{ return ifacultynname; }

void take::university(char* x)
{ char fact[100];

if (!take_Constructor)
{ save_fact("UPDATE take university");
save_fact("Before Action");
strcpy(fact, "Object-Name ");
strcat(fact, "take");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("take.rule");

iuniversity=new char[strlen(x)+1];
strcpy(iuniversity, x);
if (!take_Constructor)
{ save_fact("UPDATE take university");
save_fact("After Action");
strcpy(fact, "Object-Name ");
strcat(fact, "take");
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("take.rule");

if (counter){
if (!take_Constraint())
  cout << "Violation on Updating Type: take,
    Attribute: university" << endl; }

char* take::university()
  { return iuniversity; }

Boolean take::forcondition_take0()
{
  // begin Instance Iterator
  SetIterator anInstance(set_std);
  Student* s;
  while (anInstance.moreData())
  { s=(Student*) (Entity*)anInstance();
    if (!s->StudentNo()>89000))
      return FALSE; }
  return TRUE;
}

Boolean take::forcondition_take1()
{
  // begin Instance Iterator
  SetIterator anInstance(pattern0);
  tuple* x;
  while (anInstance.moreData())
  { x=(tuple*) (Entity*)anInstance();
    if (!!(x->grade()<=10&&x->grade()>=0))
      return FALSE; }
  return TRUE;
}

A.3.5 The take.rule File

;Begin Trigger Section
(defrule load_instances
  (initial-fact)
  =>
  (OLoadInstances take))
(defrule monitor
  (take ?class-ptr)
  (ObjectName take ?object-ptr)
  (test (eq (OCallString ?class-ptr take takeName)
            (str-cat ?object-ptr)))
  (MESSAGE take AssignGrade)
  (After Action)
  =>
  (printout t "Running Clips Rules ...monitor" crlf)
  (OCallVoid ?class-ptr take monitor))
;End Trigger Section
A.3.6 The takefun.c File

```c
#include "stdio.h"
#include "string.h"
#include "stream.h"
#include "Object.h"
#include "Set.h"
#include "Directory.h"
#include "Iterator.h"
#include "Database.h"

#define true TRUE
extern void clipsmain(char *);
extern void save_fact(char *);
extern void copy_object(Type*, Object*);
extern void remove_object(Type*, Object*);

#include "globalfun.h"
#include "take.h"
int take_Constructor=0;

take::take(char* fac):(fac)
{
    take_Constructor=1;
    initDirectType((Type*)OC_lookup("take"));
    take* obj;
    obj=(take*) OC_lookup(fac);
    if (!obj)
    {
        cerr<<"ERROR: Object exists"<<endl;
        exit(1);
    }
    counter++;
    if (counter==1) copy_object(getDirectType(), this);
    { facultyname(fac); }
    if (!take_Constraint())
    cout<<"Cannot Create Object due to Constraint Violation\n";
    else {cout<<"The Object is Created."<<endl;
    Name(fac);
    cout<<"******************************"<<endl;}
    take_Constructor=0;
    counter--;
    if (!counter) remove_object(getDirectType(), this);
}

int take::GetGrade(Student* student, Course* course)
{
    char fact[100];
    if (!take_Constructor)
    { 
```
Appendix A. Examples of KOM Programs

```c
{ save_fact("MESSAGE take GetGrade");
 save_fact("Before Action");
 strcpy(fact, "Object-Name ");
 strcat(fact, "take");
 strcat(fact, ""); strcat(fact, Name());
 save_fact(fact);
 clipsmain("take.rule");}

{ tuple* x;
 x=(NULL);
 SetIterator iter1(pattern0);
 while(iter1.moreData())
 { x=(take_pattern0*)(Entity*)iter1();
 if (x->Participant1()==student && x->Participant2()==course)
 break; }
 if (!take_Constructor)
 {save_fact("MESSAGE take GetGrade");
 save_fact("After Action");
 strcpy(fact, "Object-Name ");
 strcat(fact, "take");
 strcat(fact, ""); strcat(fact, Name());
 save_fact(fact);
 clipsmain("take.rule");}
 if (!counter){
 if (!take_Constraint())
 cout<<"Violation on Updating Type:take"<<endl; }
 return x->grade(); }
 if (!take_Constructor)
 {save_fact("MESSAGE take GetGrade");
 save_fact("After Action");
 strcpy(fact, "Object-Name ");
 strcat(fact, "take");
 strcat(fact, ""); strcat(fact, Name());
 save_fact(fact);
 clipsmain("take.rule");}
 if (!counter){
 if (!take_Constraint())
 cout<<"Violation on Updating Type:take"<<endl; }
}
```

void take::AssignGrade(Student* student, Course* course, int mark)
{ char fact[100];
 if (!take_Constructor)
 { save_fact("MESSAGE take AssignGrade");
 save_fact("Before Action");
 strcpy(fact, "Object-Name ");
 strcat(fact, "take");
```
strcat(fact, " "); strcat(fact, Name());
save_fact(fact);
clipsmain("take.rule");
{
tuple* x;
x=(NULL);
SetIterator iter2(pattern0);
while(iter2.moreData())
{
x=(take_pattern0*)(Entity*)iter2();
if (x->Participant1()==student && x->Participant2()==course)
break;
}
x->grade(mark);
}
if (!take_Constructor)
{
save_fact("MESSAGE take AssignGrade");
save_fact("After Action");
strcpy(fact, "Object-Name ");
strcat(fact, "take");
strcat(fact, " ");
strcat(fact, Name());
save_fact(fact);
clipsmain("take.rule");
if (!counter)
if (!take_Constraint())
cout<<"Violation on Updating Type:take"<<endl; }
}
int take::NoPassed(Student* student)
{
char fact[100];
if (!take_Constructor)
{
save_fact("MESSAGE take NoPassed");
save_fact("Before Action");
strcpy(fact, "Object-Name ");
strcat(fact, "take");
strcat(fact, " ");
strcat(fact, Name());
save_fact(fact);
clipsmain("take.rule");
{
allcourses* x;
tuple* y;
int np;
np=0;
x=(NULL);
SetIterator iter3(pattern1);
while(iter3.moreData())
{
x=(take_pattern1*)(Entity*)iter3();
if (x->Participant1()==student)
break;
}
// begin Instance Iterator
Type *tp1;
tp1=(Type*) OC_lookup("Course");
Appendix A. Examples of KOM Programs

```c
InstanceRefIterator anInstance1(tp1);
Course* z;
while (anInstance1.moreData())
{ z=(Course*)(Entity*)anInstance1();
  y=(NULL);
  SetIterator iter4(pattern0);
  while(iter4.moreData())
  { y=(take_pattern0*)(Entity*)iter4();
    if (y->Participant1()==student && y->Participant2()==z)
      break;
  }
  if (y->grade()>=S)
  { np=np+1; }
}
x->nopassed(np);
if (!take_Constructor)
{ save_fact("MESSAGE take NoPassed");
  save_fact("After Action");
  strcpy(fact, "Object-Name ");
  strcat(fact, "take");
  strcat(fact, " "); strcat(fact, Name());
  save_fact(fact);
  clipsmain("take.rule");
}
if (!counter){
  if (!take_Constraint())
    cout<<"Violation on Updating Type:take"<<endl; }
return np; }
if (!take_Constructor)
{ save_fact("MESSAGE take NoPassed");
  save_fact("After Action");
  strcpy(fact, "Object-Name ");
  strcat(fact, "take");
  strcat(fact, " "); strcat(fact, Name());
  save_fact(fact);
  clipsmain("take.rule");
}
if (!counter){
  if (!take_Constraint())
    cout<<"Violation on Updating Type:take"<<endl; }
}
```

A.3.7 The take_slices.h File

```c
#ifndef Def_slices
#define Def_slices
#include "Object.h"
#include "Reference.h"
```
Appendix A. Examples of KOM Programs

```cpp
#include "Set.h"
#include "Directory.h"
#include "Iterator.h"
#include "student.h"
#include "course.h"

class take_pattern1: public Object
{
    private:
        int inopassed;
    public:
        Reference std;
        Set* crs;
        Type* getDirectType();
        take_pattern1(APL *);
        take_pattern1();
        take_pattern1(char* name);
        int nopassed();
        void nopassed(int x);
        Student *Participant1();
        Set *Participant2();
        void insert(Student *ins_std, Set *ins_crs);
};
class take_pattern0: public Object
{
    private:
        int igrade;
    public:
        Reference std;
        Reference crs;
        Type* getDirectType();
        take_pattern0(APL *);
        take_pattern0();
        take_pattern0(char* name);
        int grade();
        void grade(int x);
        Student *Participant1();
        Course *Participant2();
        void insert(Student *ins_std, Course *ins_crs);
};
class take_pattern2: public Object
{
    private:
        Set* ienrollees;
    public:
```
Appendix A. Examples of KOM Programs

Set* std;
Reference crs;
Type* getDirectType();
take_pattern2(APL *);
take_pattern2();
take_pattern2(char* name);
Set* enrollees();
void enrollees(Set* x);
Set *Participant1();
Course *Participant2();
void insert(Set *ins_std, Course *ins_crs);
}
typedef take_pattern1 allcourses;
typedef take_pattern0 tuple;
typedef take_pattern2 aclass;
#endif

A.3.8 The take_slices.c File
#include "take_slices.h"
#include "stdio.h"
#include "string.h"
#include "stream.h"
#include "Object.h"
#include "Set.h"
#include "Directory.h"
#include "Iterator.h"
#include "Database.h"

#define true TRUE
extern void clipsmain(char *);
extern void save_fact(char *);
#include "Transaction.h"

void take_pattern1::nopassed(int x)
{ inopassed=x; }
int take_pattern1::nopassed()
{ return inopassed;}
take_pattern1::take_pattern1(APL* theAPL):(theAPL){}
Type*take_pattern1::getDirectType()
{return (Type*)OC_lookup("take_pattern1");}
take_pattern1::take_pattern1()
{initDirectType((Type *)OC_lookup("take_pattern1");}
take_pattern1::take_pattern1(char* name):(name)
{ initDirectType((Type*)OC_lookup("take_pattern1");
take_pattern1* obj;
Appendix A. Examples of KOM Programs

```c
Appendix A. Examples of KOM Programs

obj=(take_pattern1*)OC_lookup(name);
if (!obj) {cerr<<"ERROR: Object exists"<<endl;
  exit(1);}
inopassed=0;
Name(name);
}
Student* take_pattern1::Participant1()
  { return (Student*)std.Binding(this);}
Set* take_pattern1::Participant2()
  { return crs;}
void take_pattern1::insert(Student *ins_std, Set *ins_crs)
  { std=ins_std; crs=ins_crs; }
void take_pattern0::grade(int x)
  { igrade=x; }
int take_pattern0::grade()
  { return igrade;}
take_pattern0::take_pattern0(APL* theAPL):(theAPL){
  return (Type*)OC_lookup("take_pattern0");
} take_pattern0::take_pattern0()
  {initDirectType((Type*)OC_lookup("take_pattern0"));
take_pattern0::take_pattern0(char* name):(name)
  { initDirectType((Type*)OC_lookup("take_pattern0");
    take_pattern0* obj;
    obj=(take_pattern0*)OC_lookup(name);
    if (!obj) {cerr<<"ERROR: Object exists"<<endl;
      exit(1);}
    igrade=0;
    Name(name);
  }
Student* take_pattern0::Participant1()
  { return (Student*)std.Binding(this);}
Course* take_pattern0::Participant2()
  { return (Course*)crs.Binding(this);}
void take_pattern0::insert(Student *ins_std, Course *ins_crs)
  { std=ins_std; crs=ins_crs; }
void take_pattern2::enrollees(Set* x)
  { ienrollees=x; }
Set* take_pattern2::enrollees()
  { return ienrollees;}
take_pattern2::take_pattern2(APL* theAPL):(theAPL){
  return (Type*)OC_lookup("take_pattern2");
} take_pattern2::take_pattern2()
  {initDirectType((Type*)OC_lookup("take_pattern2"));
take_pattern2::take_pattern2(char* name):(name)
```
{ initDirectType((Type*)OC_lookup("take_pattern2"));
  take_pattern2* obj;
  obj=(take_pattern2*)OC_lookup(name);
  if (!obj) {cerr<<"ERROR: Object exists"<<endl;
    exit(1);}
  enrollees=NULL;
  Name(name);
}

Set* take_pattern2::Participant1()
{ return std;}

Course* take_pattern2::Participant2()
{ return (Course*)crs.Binding(this);}

void take_pattern2::insert(Set *ins_std, Course *ins_crs)
{ std=ins_std; crs=ins_crs; }
Bibliography


