K: A High-level Knowledge Base Programming Language for Advanced Database Applications

Yuh-Ming Shyy, Stanley Y.W. Su

Database Systems Research and Development Center
Department of Computer and Information Science
CSE 470, University of Florida
Gainesville, FL 32611

Email: yms@cis.ufl.edu, su@cis.ufl.edu

ABSTRACT

K is a high-level knowledge base programming language for doing general computation as well as for defining, querying, and manipulating databases in nontraditional application domains. The main features of K are: (i) knowledge abstraction facilities for supporting an extensible object-oriented semantic association knowledge model, (ii) a modularization mechanism for programming in the large, (iii) knowledge retrieval facilities for querying the knowledge base, and (iv) multi-paradigm programming constructs for specifying object-oriented, parallel, non-deterministic, and rule-based computations. This paper presents the linguistic facilities provided in K and the knowledge model it supports.

1. INTRODUCTION

1.1 Motivation

Nontraditional database application domains such as CAD/CAM, Artificial Intelligence, Office Information System, and Software Engineering have suffered from the lack of a powerful knowledge base management system (KBMS) support. For example, structural and behavioral properties of a complex system are often subject to design, operational, and system rules (constraints and triggers). If these rules are captured and specified in the conceptual schema instead of being buried in the application codes, then they can be used for automatically maintaining constraint and triggering predefined actions. For complex system development, tools must also be provided to a developer to define, modify and test the system as well as to store each system component and related information (e.g., test result, documentation, requirements, and design decision) in a persistent database. Querying facilities must also be provided to allow ad hoc inquiries of the system and the data gathered about it. When modification is made to the system, consistency checking and integrity enforcement need to be carried out. Transaction management for supporting concurrent access of the KBMS (e.g., via the execution of a parallel program or by a cooperative group of designers) are also needed. Similar goals have been set up for the so-called *third generation database systems* [40] which term will be used interchangeably with "KBMS" in this paper. While existing database programming languages do not have adequate facilities for supporting the functionalities mentioned above (e.g., declarative and set-oriented query processing, parallel processing, and rule processing), a new knowledge base programming language (KBPL) is needed as a high-level interface to the KBMS. In this paper, we propose a KBPL called K (previously called MPL/0 in [38]) for serving this purpose.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1991 ACM 0-89791-425-2/91/0005/0338...$1.50

1.2 Design Principles

The design of K is guided by the following principles:

(1) **Knowledge Abstraction and Encapsulation:** For any application domain, K provides facilities to capture (i) structural properties as associations (such as generalization and aggregation), and (ii) behavioral properties as methods (both specification and implementation) and rules (constraints and triggers). All the knowledge is incorporate into a rich class system.
(2) **Modularization**: In order to efficiently manage the name-space and provide encapsulation at a larger granularity than classes, we need a *modular* system as in Galileo [4] to facilitate programming in the large.
(3) **Wide-spectrum** (for both specification and implementation): K should be a uniform language for high-level knowledge definition, declarative knowledge retrieval, knowledge manipulation, and general computation.
(4) **Multi-paradigm**: K should be a multi-paradigm language to be *wide-applicable* to various application domains. Object-Oriented, parallel, non-deterministic, and rule-based computations should be supported.
(5) **Readability and Maintainability:** the software written in K should have readable syntax and stable semantics so that it can be easily understood and maintained.
(6) **Strongly-typed**: K should be a strongly-typed language so that as many type errors can be checked by static type checking as possible [6].

Many of the features mentioned above can be found in the fields of database management system, programming language, software engineering, and artificial intelligence. Our major contribution lies in providing a clean fusion of the advances in all these fields within an object-oriented framework. Along with the language, we have extended the OSAM* model [46,47,48] into an extensible object-oriented semantic association model which provides a rich class system to support persistence, knowledge abstraction, encapsulation, and multiple inheritance. Everything including both the specification and implementation of a target system are treated as objects and stored in the database. A context construct based on [3,19,48] has also been incorporated into K for programming with associations. It can be used to specify modules, queries (including recursive queries [1]), and rules. A unified execution model for supporting multi-paradigm computation based on nested transaction and two-phase locking has also been developed [39]. K is being implemented for use as both the high-level interface and the development tool of a third generation extensible KBMS under development at the Database Systems Research and Development Center of the University of Florida.

1.3 Related Works

As a KBPL, K shares with other database programming languages [2, 4, 5, 6, 9, 17, 20, 24, 31, 34, 35, 36, 37, 41] the goal of avoiding the *impedance mismatch* problem between traditional programming languages and DDL/DML [14] by providing a single language for data definition, data manipulation (including the capability of handling persistence), and general computation. K is closely related to O++ [2] in that both provide facilities for querying the database and associating rules with object classes. Unlike O++ which is a superset of C++, K is designed to be a *high-level* programming language in the sense that (i) K is based on an extensible object-oriented semantic association knowledge model rather than C++ data model, (ii) K provides more declarative and expressive constructs based on association patterns for specifying queries and rules, (iii) K uses address-independent object identifiers OIDs (soft pointers) as object surrogates rather than using physical address pointers (hard pointers), and (iv) it puts more emphasis on readability and maintainability by providing readable syntax and modularization mechanism rather than using the cryptic and non-intuitive C++ syntax.

K supports the same features found in existing semantic and object-oriented data models [23, 33, 42], such as a rich type system, structural associations, operational specifications, object identification, encapsulation, and inheritance. K also supports knowledge rules as in other research efforts of developing third generation database systems [13, 16, 22, 26, 43, 46, 47]. K is also related to *extensible* data model and database projects [8, 11, 12, 15, 28, 43] in that both enable the user to specify new data types and special-purpose operations and to integrate them into existing model easily. It is also possible in K to define new association types and thus extend the model itself.

In this paper, we concentrate on the linguistic facilities provided in K and the knowledge model it supports. For the purpose of this paper, the execution of each object method will be considered to be a single transaction. A detailed description of the computation model is out of the scope of this paper due to space constraint. The rest of this paper is organized as follows. Section 2 describes the knowledge abstraction facilities provided in K. Section 2.1 gives an overview of the underlying knowledge model of K. Section 2.2 discusses the context mechanism in terms of the basic context expression and recursive association pattern. Section 2.3 describes the type system and expressions of K. Section 2.4 describes the modularization mechanisms. Section 3 presents the query processing facilities. Section 4 describes the multi-paradigm computation constructs. parallel, rule-based, and non-deterministic computations are described in Section 4.1, 4.2, and 4.3, respectively. Our conclusions are given in Section 5.

2. KNOWLEDGE ABSTRACTION CONSTRUCTS

2.1 Knowledge Model

2.1.1 Overview: The underlying knowledge model of K is an extension of OSAM* [45,46,48] whose key features can be summarized as follows. It has a semantically rich class system which allows an object class to be defined in terms of its structural properties, operational properties, and knowledge rules in an integrated fashion. Objects are grouped into *entity classes* and *domain classes*. The sole function of a domain class is to form a domain of possible values from which descriptive attributes of objects draw their values. An entity class, on the other hand, forms a domain of objects which occur in an application's world and can be physical entities, abstract things, events, processes, and relationships. Each object of an entity class is given a unique object identifier (OID) which is internally represented as a timestamp (based on a linear and discrete representation of time) to record the creation time of this object (the object manager will make sure that no two objects are created at exactly the same time). The user is also given the flexibility to specify a *primary key* (either single or composite) for each entity class. The structural properties of each object class and thus its instances are uniformly defined in terms of its associations (e.g., aggregation, generalization, and others) with other object classes. Each type of association represents a set of generic rules that governs the knowledge base manipulation operations on the instances of those classes that are defined by the association types. The behavioral properties of each object class are defined as methods and rules applicable to the instances of this class. Each method is defined by the operational specification (or *signature*) and its corresponding method body (fully coded programs or a pre-stored tables which generate legitimate output given some input data). Each rule of class X is specified by an active specification, trigger condition, if condition, then condition, and corrective action with the semantics that during the period that this rule is active, when the trigger condition occurs, if the "if condition" is true, then the "then condition" must also be true; otherwise, the "corrective action" will be taken and we say this rule is *triggered*. A detailed description of rules will be given in Section 4.2. Since rules applicable to the instances of a class are defined with the class, rules relevant to these instances are naturally distributed and available for use when instances are

processed. All the structural properties, operational properties, and rules defined by a class are *inherited* by its subclasses where the superclass-subclass relation is defined by the generalization association.

2.1.2 Extended Class System: We extend the OSAM* class system to achieve *extensibility* by uniformly modeling any application domain (including the model itself) as classes. An overall class structure is shown in Figure 2.1. Note that the figure shows the structural relationship among the object classes relevant to the presentation of this paper. In our graphic representation, (i) entity classes and domain classes are represented as rectangular nodes and circular nodes, respectively, and (2) generalization association is represented by a "G" link from the superclass to the subclass(es). In general, a class definition consists of three parts: association section, method section, and rule section as shown in Figure 2.2. The detailed explanation of Figure 2.2 will be given in the following sections.

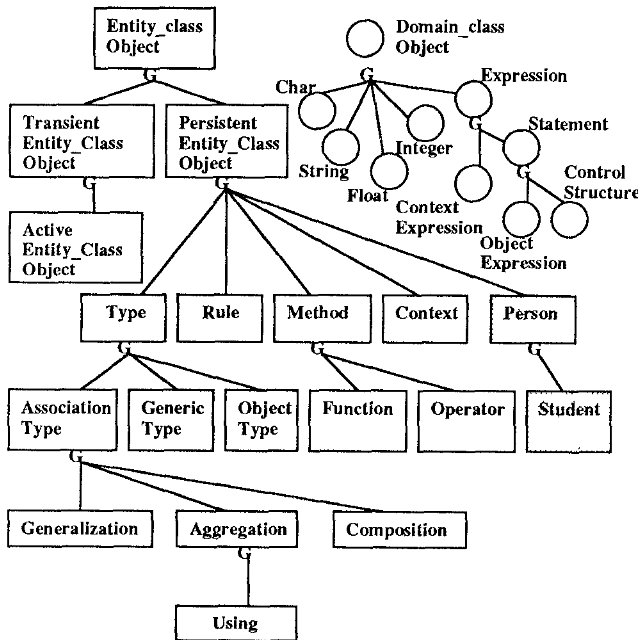

Figure 2.1 The Extended System Classes Structure of K

Any user-defined class (e.g., "Person" and "Student" in Figure 2.1) will be added to the class structure as an immediate or non-immediate subclass of either "Entity_Class_Object" or "Domain_Class_Object" while at the same time the objects corresponding to the class definition, associations, methods, and rules will be created as instances of the system-defined classes "Object_Type", "Association_Type", "Method", and "Rule", respectively. Structural associations, methods, and rules are all modeled as classes as we will describe in the following. Because of the uniformity of objects, the user can further modify and extend the class structure at any level of abstraction. An entity class can be specified with the key words "**persistent**" and "**active**" as we will explain in the following.

```
/* Class Definition */

define persistent entity_class Student is
  associations
    superclasses Person;
    instance_variables
            unchangeable S# : S#_Value;
            optional major : Department;
            optional enrolled : Set[1..6] of Course;
    key S#;
  end_associations;

  methods
    function straight_A_student() : Boolean is
      begin
       if not exist
         Student={this} <- Transcript[grade_point < 4.0]
           /* "this" is the default pseudo variable  */
           /* that denotes the receiver of a message */
         then return true;
         else return false;
       end_if;
      end;
    end_function straight_A_student;

    function GPA() : GP_Value;
    function suspended() : void;
    private function update_transcript() : void;

  end_methods;

  rules
    rule academic_standard_1 is
      if exist
         Student={this}->majorDepartment[name=="CIS"];
      then this.GPA() >= 3.0;
      corrective_action
        this.suspended();
    end_rule academic_standard_1;
  end_rules;

end_define Student;


/* implement a method outside a class definition */

function Student::GPA() : GP_Value is
 local s1,s2 : Real init_value 0;
 begin
  context
    Student={this} <- t:Transcript -> c:Course
    begin
         s1 := s1 + (c.credits * t.grade_point);
         s2 := s2 + c.credits;
    end;
  end_context;
  return (s1/s2);
 end;
end_function GPA;
```

Figure 2.2 Class Definition of Student

Persistent and Transient Entity Classes: We adopt the same rationale that persistence is orthogonal to types [6] as in

Exodus/E [34], ODE/O++ [2], and ONTOS [32], i.e., persistence is an object property rather than a type property. Any persistent entity class (subclass of "Persistent_Entity_Class_Object" as specified by the key word "**persistent**") automatically inherits the persistence mechanism and it is up to the user to specify each of its instance to be either persistent or transient. Instances of domain classes can be persistently stored in the database only if they serve as attribute values of other persistent objects. Contrary to other C++-based persistent languages mentioned above, K uses (i) *OIDs* uniformly for any entity class objects and (ii) *values* for any domain class objects, no matter whether these objects are in main memory (transient) or on disk (persistent). The advantage is that K provides a better object-oriented flavor and data abstraction than C++-based languages by enabling the users to (i) manipulate objects at the logic level instead of going to the physical level and (ii) navigate through the database using OIDs instead of pointer chasing. For each persistent entity class, in addition to a persistent object table which matches OIDs to secondary storage address for its persistent objects, a run time object table is also needed which maps OIDs of transient objects to their main memory address. For example, if we define two variables s and ps of type Student, then the statements "s := **new** Student(S# := "1234", major := EE)" and "ps := **pnew** Student(S# := 1238", major := ME)" will create two objects of Student and assign their OIDs to variables s and ps, respectively. The difference between these two objects is that ps is a persistent object and therefore any modification will be written to the database while s is a transient object and only exists in main memory. One can copy from a persistent object to a transient object or vise versa by sending the message **copy_object**(from,to) to object manager (e.g., for the purpose of reducing secondary storage accesses). A transient object can be made persistent by sending the **put_object**(OID) message to object manager which will update its object table and store this object into secondary storage. Note that by using OIDs uniformly, no dual pointers as in ODE [2] is necessary.

Active Entity Classes: Subclasses of the system-defined class "Active_Entity_Class_Object" (specified by the key word "**active**") are active classes whose instances are concurrent processes and can be used to model *concurrent tasks* [18] or concurrent software modules as we will describe in Section 4.1.

2.1.3 Structural Abstraction: Structural properties are modeled as objects which can be categorized into *association types* "Aggregation", "Generalization" and so on as shown in Figure 2.1. The user can thus modify the definition of existing structural associations or add new types of structural associations just like modeling any other classes to *extend* the knowledge model.

K supports the basic association types generalization, composition, and aggregation. Generalization association is specified as "**superclasses** <classes> **subclasses** <classes>" which specifies the immediate superclasses and/or subclasses of the class being defined. Composition association is specified as "**metaclasses** <classes> **constituent_classes** <classes>". For example, we can specify that the metaclass "Campus_Group" has two instances "Student" and "Professor" as "**constituent_classes** Student, Professor". Note that properties defined by a metaclass apply to each constituent class as a whole instead of each instance of the constituent class. We use the quote notation "'<class>" to refer to a class as a whole, e.g., "'Student.chairman" refers to the chairman of Campus_Group Student. For each class, we can also define a set of attributes which are instances of the association type Aggregation in the form "**instance_variables** <aggregations> **key** <identifiers>". Each attribute is defined as "{specification} <name> : <range>" followed by a list of optional properties defined by the class "Aggregation". For example, we can give the default value of "nationality" as "USA" as "nationality : Country **default** "USA"". Besides, we can also specify: (1) **class_value** which specifies the attribute value common to all the instances of this class, (2) **init_value** which specifies the common initial value among all the instances of this class, and (3) **inherited_from** which specifies one of the superclasses from which this attribute is inherited to solve the name conflict problem in multiple inheritance. An attribute can be specified to be "**optional**", "**unchangeable**", "**private**", or "**dependent**". An attribute of class X must be given a value when an instance of X is created unless it is specified to be "optional". Once given a value, it cannot be changed if the attribute is specified to be "unchangeable". An attribute cannot be seen from outside the class by which it is defined if it is specified to be "private". An attribute is "dependent" if its attribute values are entity class objects which cannot exist by themselves. For example, the attribute "kids_under_three" of class "Employee" is dependent in a company database if whenever a particular employee is deleted from the database, all his kids must also be deleted. To access an attribute value, we use the conventional dot notation "<object>.<attribute>".

2.1.4 Behavioral Abstraction: Behavioral properties of a class are specified as methods and rules. As shown in Figure 2.1, both methods and rules are modeled as objects of system-defined classes "Method" and "Rule", respectively. The advantages of this approach are as follows. First, we incorporate the behavioral properties uniformly into the knowledge model where everything is treated as an object. Secondly, based on the uniformity, we can model methods and rules at any level of abstraction. For example, The user can extend the definition of Method or Rule and organize methods and rules using structural associations just like modeling any other classes. The user can define such methods as "activate", and "deactivate", and some meta-rules that govern the behaviors of rules themselves.

A list of methods are specified in the method section in which each method is defined as "{specification} **function**|**operator** <name> <parameter-list> : <range>" followed by a list of optional properties defined by the meta class "Method" and the method body followed the key word "**is**". A method can be used only within the implementation of the class by which it is defined if it is specified to be "**private**" (e.g., function "update_transcript()" in Figure 2.2). A method can also be specified to be "**virtual**" in the definition of class X in which case the actual implementation will be given in the subclasses of class X. Each parameter and local variable can be

defined in the same way as a public and private attribute of the method class, respectively. For example, a parameter p with range Integer and initial value 0 can be defined as "p : Integer **init_value** 0". We will explain the method body in the following subsections. Finally, a list of rules are defined in the rule section in which each rule is defined as "{active_specification} **rule** <name> **is** <rule-body>". An active specification defines "when" this rule is active as (1) "**always**" (the default specification) which specifies that this rule is always active during its whole life-span, i.e., as long as it is not deleted. (2) "**once_only**" which specifies that this rule will be automatically deactivated whenever it is triggered until it is explicitly activated again by receiving a message "**activate()**". A detailed description of rules will be given in Section 4. Associations, methods, and rules can also be defined outside a class definition. For example, the function "GPA" is defined outside the definition of class "Student" as shown in Figure 2.2, and it is represented by the notation "Student::GPA() : GP_Value **is** <body>" as in C++ [44].

2.2 Context Mechanism

2.2.1 Basic Context expression: We adopt the "context" concept of OQL [3, 19, 26, 48] for specifying modules, queries, and rules. A context expression defines a "sub-knowledgebase" by specifying an *intensional association pattern* among classes followed by an optional where-clause in the form "class_1[<intra-class selection condition>] <op><association-name> class_2[<intra-class selection condition>] <op><association-name>...**where** <var1> = <value1>, <var2> = <value2>,...,<inter-class selection condition>", where <op> could be either an "associate" ("->" and "<-") or a "non-associate" ("!>" and "<!") operator. Note that in the where-clause, we can assign values (e.g., the results of some aggregation functions which are applied to the entire sub-knowledgebase as a whole) to some variables which will be used for further processing of this sub-knowledgebase as we will describe in Section 3. A sub-knowledgebase consists of the selected classes and their associations together with the objects that exist in the specified association pattern. While the default range variable over a class is the class name as in SQL, we can also specify it in the association pattern as "<var>:<class>". For example, "g:**Grad**[major == "CIS"] ->advisor p:**Professor** <!instructor **Course**" specifies a subknowledgebase that contains all the graduate students of CIS department who has an advisor (i.e., there is an "advisor" association connecting this student with a professor) who does not teach any course (i.e., this professor is not connected through the "instructor" association with any course object), as well as their advisors and courses that these advisors do not teach. Note that the association direction is explicitly specified. We use the "->" operator in "Student ->advisor Professor" because "advisor" is an association defined by class "Student". Similarly, we use the "<!" operator in "Professor <!instructor Course" because "instructor" is an association defined by class "Course". Instead of using a selection condition, we can also directly *designate* objects in the form "<class>={<objects>}". For example, "Student={**this**} ->advisor Professor" specifies a context which consists of either the particular student denoted by "**this**" and his/her advisor, or nothing (if this student has no advisor). An association name can be omitted when it has the same name as the class to which it connects. For example, "Student <- Transcript -> Course" is a legal expression if Transcript has aggregation associations called "Student" and "Course" with classes Student and Course, respectively. A context can also be thought of as a normalized table whose columns are defined over the participating classes (for the above example, "Grad", "Professor", and "Course"). In the original OQL we do not specify association direction. It is added to comply with the design principle of readability and maintainability.

2.2.2 Recursive Association Pattern: The least fixpoint operator is considered to be an essential addition to relational query languages [1] to express recursive queries. For example, to find all the descendants of John Smith, we have to recursively traverse from class Person to itself through the aggregation association "kids" as shown in Figure 2.3. Figure 2.3(a) shows the conceptual pattern for solving this problem. Figure 2.3(b) shows that "John Smith" (p1) has kids p2 and p3, and p3 in turn has kids p4 and p5. Figure 2.3(c) shows the actual extensional pattern (i.e., the *instantiation* of an intensional pattern) where we use dot line to show that p4 and p5 are added back to the set containing p2 and p3. The above recursive association pattern can be easily expressed in K by using the key word "**recursive**" as the context expression "Person[name = "John Smith"] ->kids p:Person ->kids **recursive** Person" which tells the system to add persons participating the third Person class recursively back to the nearest class of the same name, i.e., the second Person class. In other words, it starts with John Smith and gets all his kids, then it recursively traverses through the "kids" association and add all the selected persons into the class Person denoted by p.

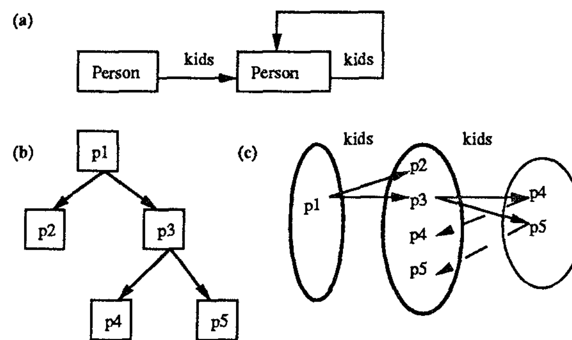

Figure 2.3 Recursive Association Pattern

2.3 Type System and Expressions

2.3.1 Type System: K distinguishes type and class in the sense that each type is treated as a class definition object which is an instance of the system-defined class "Type". There are three types of type: Object_Type, Generic_Type, and Association_Type as shown in Figure 2.1. Corresponding to each instance of Object_Type, there is a class in the class structure to serve as the container of all the instances of this class. Instances of Generic_Type are *parameterized* types which are used as *templates* or *type constructors* to construct actual

object_types by instantiating generic attributes. For example, "Set" is a system-defined generic type with generic attributes "setof" and "cardinality" (optional). We can construct a new class "`Set[1..6] of Course`" each of whose instances is a set of at least 1 and at most 6 Course instances. Generic types and inheritance are the two built-in mechanisms for achieving *knowledge reuse* in our model. An Association_Type instance defines a different type of association such as "Generalization", "Aggregation" and other user-defined association types.

Unlike general class definition mentioned in Section 2.1, *virtual* classes (i.e., those classes that can not be populated) can be defined as *"union"*, *"view"* [52], and *"selection"* [12]. A domain class can also be defined by *enumeration*. For example:

```
(1) define entity_class college_people is union (Student,
      Professor) end_define College_People;
(2) define entity_class Advisor is selection Professor of
      Grad ->advisor Professor end_define Advisor;
(3) define entity_class Advisor_info is view Professor of
       Grad ->advisor Professor with name, rank
    end_define Advisor_Info;
(4) define domain_class Color is
      enumeration ("Red", "Green", "White", "Yellow")
    end_define Color;
```

Note that a selection is specified by a class name and a context expression, e.g., instances of Advisor are those instances of class Professor each of whom is the advisor of some graduate student. Similarly, the class Advisor_info is defined with the same selection condition as "Advisor" except that only attributes "name" and "rank" are available (e.g., we want to hide the salary information for privacy and security concern).

2.3.2 Expressions: Expressions are the basic building blocks of K. There are two types of expressions: statements and context expressions, where statements can be either object expressions or high-level control structures which glue expressions to form complex programs as shown in Figure 2.1. Every object expression is evaluated to return an object and thus has a type which in general can be detected by the type checker by textual inspection (*static type checking*) to decide the type compatibility and thus prevent an operation from being applied to a value of an inappropriate type. Every variable must also be declared to have a type. Type compatibility means that (i) a variable of type X can only be assigned object expressions which represents objects of class X or any subclass of X, (ii) method parameters and returned values are checked against the method signature following the same rule as (i). If the type checker is not able to ascribe a type to an expression, the user must specify the type with "`<type>$<expression>`", e.g., `Real$(3+4)` asserts that the type of `(3+4)` is Real instead of Integer. An association, method, or rule can also be tagged with the class to which it belongs as "`<class>::<name>`" to define or refer to it outside a class definition (e.g., in order to solve *name conflict* in *multiple inheritance*). We also provide "**`type_case <var> of..`**" construct for handling variable of virtual class type. For example, to express the possibility that a variable of type "College_People" can be bound to either a student or a professor at run time, we can use the statement

```
type_case x of
    when s:Student =>
        printf("student GPA = %s", s.GPA());
    when p:Professor =>
        printf("professor salary = %d", p.salary);
end_type_case;
```

Note that both (i) *static type checking* through the use of tagged variables "s" and "p" as in CLU [29] and (ii) *late-binding* for variable "x" are supported. Two object types X and Y are *union-compatible* if X is equal to Y or a subclass or superclass of Y. As a special case, each context expression returns a sub-knowledgebase whose type can be thought of as an ordered list of all the participating types. Set operations (union, difference, and intersection) can be applied to two context expressions if their types are union-compatible in pair. For example, "`(A ->p B) union (C ->q D)`" is a legal expression if the types (A,B) and (C,D) are union-compatible, i.e., A and C, B and D are union-compatible, respectively.

2.4 Modularization Mechanism

As mentioned before, in addition to using classes as the general modularization mechanism as in C++ [44], K also provides the context mechanism for specifying sub-knowledgebases and an "Using" association type [10,27] for combining several classes into a big module and thus to facilitate programming in the large.

2.4.1 Context: As each type specifies a class, each context specifies a sub-knowledgebase in K. Each context definition is an instance of the class "Context", and it can be given a name and/or stored back to the persistent knowledge base in the form "`define context <identifier> is <context-expression> end_define <identifier>`". From any context (initially the *global* context), a programmer can enter any specific context by issuing a "`context <context> /*code*/ end_context`" statement, and any computation specified inside the statement will be evaluated in that current context. Note that context statement can be nested and at the end of a context statement, the current context is closed and the computation returns to the parent context. For example, if we are only interested in the information about professors of CIS department, we can enter the context by the statement "`context Professor ->faculty_of Department[name = "CIS"] ... end_context`" and the computation specified inside this statement will be applied to objects of the context. A more detailed description of context statement will be given in the next section on knowledge retrieval.

2.4.2 Using Association: As shown in Figure 2.1, the association type "Using" is a subclass of "Aggregation". Similar to the "#include" macro in C++, an "Using" association from class A to class B specifies that objects of class A can send messages to objects of class B. In other words, we can use "Using" associations to combine several classes into a big module. Note that by using the "Using" association, we uniformly model software modules as classes which can be stored, retrieved, and manipulated in the same way as any other classes.

3. QUERY PROCESSING FACILITIES

As mentioned in Section 2.4.1, K provides a high-level *context* statement for declarative information retrieval. The syntax for a context statement is "**context** <context_expression> {<statements>} **end_context**" as shown in Figure 3.1. A context statement first identifies a sub-knowledgebase specified by the context expression. The statements following the context expression are set-oriented operations and are applied to each tuple (i.e., a set of bindings of range variables) of the sub-knowledgebase. For example, in Figure 3.1, the first context statement identifies the Department class, then the nested context statement is applied to each department. For each execution of the nested context statement, we first identify all the professors who are faculties of the particular department. In the where-clause of this context_expression, we also calculate the average professor salary of this department by calling the aggregation function "**avg**" over this context and assign this value to variable x. Then for each professor of that department, we print his/her name if his/her salary is less than the department average. Using traditional programming languages or existing DBPLs will require several times of looping over classes Professor and Department. As another example, the implementation of "GPA" of Figure 2.2 becomes clear now: we use a context construct to first identify the sub-knowledgebase which consists of those transcript objects which are related to a particular student (i.e., the receiver of this method, denoted by the pseudo variable "this") as well as those course objects which are related to these transcripts. We then apply a "**begin...end**" compound statement to each tuple of this sub-knowledgebase (i.e., a set of bindings of variables this, c, and t). We use two variables s1 and s2 to record the total credit numbers and the accumulated grade points, respectively. After the execution of this context statement, the average GPA is returned as the division of these two values.

```
Example: Print the names of all the professors with a
salary less than the average salary of their department.

local x : Integer;
begin
  context d:Department
     context p:Professor ->faculty_of Department={d}
            where x=avg(p.salary)
        if p.salary < x
            then printf("professor %s\n", p.name);
        end_if;
     end_context;
  end_context;
end;
```

Figure 3.1 Knowledge retrieval examples using context constructs

4. MULTI-PARADIGM COMPUTATION FACILITIES

Based on [51] and [25], we have "**Program = Data Structure + logic + control**". In our system, all the basic data structures such as List, Set, and Array are defined as generic classes and can be easily extended by the user. Logic and control are represented by operational properties, control structures (such as "do..while.." and "if..then..else..", defined as subclasses of "Control_Structure") and rules. Consequently, K could be used to write programs to implement any algorithm and perform any computation. In addition, K also supports parallel, rule-based, and non-deterministic computations.

4.1 Parallel and Non-deterministic Computation

The computation model of K is based on that of HiPAC [21] which in turn is based on nested transaction [30]. The execution of each object method is considered as a transaction which may contain any number of nested transactions or *subtransactions*, some of which may be required to perform sequentially, some concurrently, and all are organized as a *transaction tree* whose root is the top-level transaction. Operations can be explicitly applied in parallel by using the "**parbegin...parend**" construct).

We can also use active classes to model concurrent tasks or concurrent modules. Each active class is defined with a method called "main" which describes the behavior of all the instances of this class and is invoked automatically by the system whenever an instance is created. To activate a concurrent module (i.e., to initiate a concurrent process), we simply create an instance of this active class, which in turn invokes other modules to perform the desired computation. Communication and synchronization between active objects are based on *rendezvous* as in Ada [18]. For example, to solve the classical dinning philosophers problem, we can define "Dinning_Philosopher" as the main module which in turn uses "Fork" and "Philosopher", all of which are modeled as active classes as shown in Figure 4.1. "Fork" is defined by the methods "pick_up()", "put_down()", and a "main" method which specifies the behavior (or *script*) of each fork object in response to incoming messages as follows. Each fork object is a process (active object) that runs forever. If no message is coming, then it does nothing; if either of the "pick_up" or "put_down" message is received from a philosopher, then it executes the corresponding method (the fork and the calling philosopher can be viewed as meeting in a rendezvous); if both "pick_up" and "put_down" messages are received from two philosophers, then it *randomly* selects one of them to execute (while the fork and one of the calling philosopher meet in a rendezvous, the other philosopher is suspended until the rendezvous occurs). Similarly, each philosopher is modeled as an active object which has attributes "id", "left", "right", and "LIFE_LIMIT", where LIFE_LIMIT specifies the maximal times that each philosopher can eat in his life time and is given a class_value 10000. The behavior of each philosopher is defined in the "main" method as repeatedly picks up the right fork, picks up the left fork, eats, puts down the left fork, and puts down the right fork during his/her life time. Similarly, the top level module for this problem is also modeled as an active entity class "Dinning_philosopher". To run this program, we just create an instance of "Dinning_philosopher" by "new Dinning_Philosopher();" which in turn creates all the five forks and then the five philosophers as concurrent processes.

4.2 Rule-based Computation

Each rule definition consists of active specification, trigger condition, if condition, then condition, and corrective action. An active specification can be either "**always**" or "**once-only**" as described in Section 2.1. A trigger condition consists of two parts: (1) timing specification, which can be declared to be "**before**", "**after**", or "**in_parallel**", and (2) event specification, which can be a database operation or user-defined method. Both the "if condition" and "then condition" are boolean expression that returns true or false. We also allows the expression "**exist** <context>" which returns true if the specified context expression does not return an empty sub-knowledgebase. Finally, the corrective action is a simple or compound statement.

Each active rule of class X will be executed (i.e., rule checking and triggering) after the execution of each applicable method applied to an instance of class X if no trigger condition is specified. Otherwise, it will be executed *before*, *after*, or *in parallel with* the execution of the triggering event. The execution of rules associated with an event execution T is considered as linear extension of T. All the applicable rules are executed in parallel and thus can be thought of as concurrent subtransactions of T. More complex *coupling modes* of trigger condition, if condition, then condition, and corrective action as in HiPAC [21] are not considered at this stage and will be added in the future version of K. For example, the rule "academic_standard_1" of Figure 2.2 is always active and will be checked at the end of each method applied to a student to see that if this student participates in the context "Student={this} ->major Department={"CIS"}", i.e., if this student majors in CIS, then his/her GPA must be greater than 3.0 at all times. Otherwise, this student will be suspended.

4.3 Non-deterministic Computation

Several types of non-determinism are supported in K. First, if multiple methods are waiting to be executed, e.g., corrective actions of concurrently triggered rules and concurrent statements mentioned above, then these methods will be executed concurrently under the constraint of *serilizability*, i.e., the net result is equivalent to *some* serial execution non-deterministically decided by the underlying transaction management system. Secondly, if the method is defined with a pre-stored input/output table without fully-coded method body, then the table will be searched (similar to the *stored function* of Iris [50]). For example, we can define a method "square" for class Integer as "**function** Integer::square : Integer **table** ((0 0) (1 1) (2 4) (3 9)) **end_function** square" without giving the body for the actual computation. Finally, if the user explicitly use the "**select**" construct for specifying non-determinism occurring in concurrent applications, then the system will randomly choose one statement to execute as shown in the definition of "Fork" of Figure 4.1.

```
/* The Mortal Dinning Philosophers */

define active entity_class Fork is
   methods
     function pick_up();
     function put_down();
     function main() is
       begin
        while true
          select
           accept pick_up();
           or accept put_down();
           or terminate;
          end_select;
        end_while;
       end;
     end_function main;
   end_methods;
end_define Fork;


define active entity_class Philosopher is
   associations
     instance_variables
           id : Integer; left, right : Fork;
           LIFE_LIMIT : Integer class_value 10000;
     key id;
   end_associations;
   methods
     function main() is
       local times_eaten : Integer init_value 0;
       begin while times_eaten < LIFE_LIMIT
              begin
                times_eaten := times_eaten + 1;
                this.right.pick_up();
                this.left.pick_up();
                printf("Philosopher %d eats\n", this.id);
                this.left.put_down();
                this.right.put_down();
              end;
            end_while;
            printf("Philosopher %d dies", this.id);
       end;
     end_function main;
   end_methods;
end_define Philosopher;


define active entity_class Dinning_Philosopher is
   associations
     using Fork, Philosopher;
   end_associations;
   methods
     function main() is
       local j : Integer; f : Array[0..4] of Fork;
       begin
         for j from 0 until (j < 5)
             f[j] := new Fork();
         end_for;
         for j from 0 until (j < 5)
             new Philosopher( id := j, left := f[j],
                              right := f[(j+1) % 5] );
         end_for;
       end;
     end_function main;
   end_methods;
end_define Dinning_Philosopher;
```

Figure 4.1 An Active Class Example: Dinning Philosopher

5. CONCLUSIONS

A strongly-typed, wide-spectrum, and multi-paradigm high-level knowledge base programming language for advanced database applications along with the knowledge model it supports have been presented. A programmer can use this single language called K to (1) model any application domain, (2) perform any computation, (3) enter any specific module (sub-knowledgebase) as his/her computation environment, (4) query the knowledge base, (5) manipulate the knowledge base, and (6) exercise his/her programs repeatedly until the final target system is fully implemented and therefore facilitate *rapid prototyping* [7]. K is being implemented on top of ONTOS [32] for use as both the high-level interface and the development tool of a third generation extensible knowledge base management system under development at the Database Systems Research and Development Center of the University of Florida. We are also taking advantage of the extensibility of the knowledge model to modify the knowledge model itself and K to support versioning, temporal rules, and historical references [49]. After the first version of K is implemented, we shall use it to extend and rewrite K itself as well as the underlying KBMS to develop a third generation extensible knowledge base management system.

6. ACKNOWLEDGEMENTS

This research has been supported by the National Science Foundation grant #DMC-8814989 and Florida High Technology and Industry Council grant #UPN90090708.

7. REFERENCES

[1] Aho, A.V. and Ullman, J.D., "University of Data Retrieval Languages", *Proc. ACM Symp. Principles of Programming Languages*, Jan. 1979, pp. 110-120.

[2] Agrawal, R. and Gehani, N., "ODE (Object Database and Environment): The Language and the Data Model", *Proc. 1989 ACM SIGMOD Int'l. Conf. on Management of Data*, 1989, pp. 36-45.

[3] Alashqur, A., Su, S. and Lam, H., "OQL--A Query Language for Manipulating Object-oriented Databases", Proc. of 15th Int. Conf. Very Large Data Bases, Amsterdam, Netherlands, August, 1989.

[4] Albano, A., Cardelli, L. and Orsini, R., "Galileo: A Strongly-typed, Interactive Conceptual Language", *ACM Transaction on Database Systems*, 10(2), June 1985, pp. 230-260.

[5] Andrews, T. and Harris, C., "Combining Language and Database Advances in an Object-Oriented Development Environment", *Proc. 2nd Int'l Conf. on OOPSLA*, October 1987, pp. 430-440.

[6] Atkinson, M.P. and Buneman, P.O., "Types and Persistance in Database Programming Languages", *ACM Computing Surveys*, July 1987, pp. 105-190.

[7] Balzer, R., et al, (Common Prototyping Working Group), *Draft Report on Requirements for a Common Prototyping System*, 1988.

[8] Batory, D.S., Leung, T.Y. and Wise, T.E., "Implementation Concepts for an Extensible Data Model and Data Language", *ACM Transaction on Database Systems*, 13(3), September 1988, pp. 231-262.

[9] Bloom, T. and Zdonik, S.B., "Issues in the Design of Object-oriented database Programming Languages", *Proc. 2nd Int'l Conf. on OOPSLA*, October 1987, pp. 441-451.

[10] Booch, G., *Object-Oriented Design with Applications*, Benjamin/Cummings, Menlo Park, CA 1990.

[11] Carey, M., et al., "The EXODUS EXtensible DBMS Project: An Overview", *Readings in Object-Oriented Database Systems*, Zdonik, S., and Maier, D., (eds.), San Mateo, CA: Morgan Kaufmann Publishers, Inc., 990, pp. 474-499.

[12] Caseau, Y., "The Laure System: Documentation", Technical Report, Bellcore, 1990.

[13] Chakravarthy, U.S., "Rule Management and Evaluation: An Active DBMS Perspective", *SIGMOD RECORD*, Vol. 18(3), September 1989, pp. 20-28.

[14] Copeland, G. and Maier, D., "Making Smalltalk a Database System", *Proc. 1988 ACM SIGMOD International Conference on Management of Data*, June 1984, pp. 316-325.

[15] Dayal, U., and Smith, J., "PROBE: A Knowledge-Oriented Database Management System", in *On Knowledge Base Management Systems--Integrating Database and AI Systems*, Brodie, M. and Mylopoulos, J., (eds.), New York: Springer-Verlag, 1986.

[16] Dayal, U., et al., "The HiPAC Project: Combining Active Databases and Timing Constraints", *SIGMOD RECORD*, Vol. 17(1), March 1988, pp. 51-70.

[17] Deux, O., etc., "The Story of O2", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2(1), March 1990, pp. 91-108.

[18] US Department of Defense, *Reference Manual for the Ada Programming Language*, Springer-Verlag, ANSI/MIL-STD-1815A-1983.

[19] Guo, M.S., Su, S.Y.W., and Lam, H., "An Association Algebra for Processing Object-Oriented Databases", to appear in *Proc. 7th IEEE International Conference on Data Engineering*, Kobe, Japan, April 1991.

[20] Hammer, M., and Berkowitz, B., "DIAL: A Programming Language for Data Intensive Applications", in *Proc. ACM SIGMOD Conf. on Management of Data*, 1980.

[21] Hsu, M., Ladin, R., and McCarthy, D.R., "An Execution Model for Active Data base Management Systems", Proc. 3rd Int'l Conf. on Data and Knowledge Bases, 1988.

[22] Hudson, S.E. and King, R., "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System", *ACM Trans. on Database Systems*, Vol. 14(3), September 1989, pp. 291-321.

[23] Hull, R. and King, R., "Semantic Database Modeling: Survey, Application, and Research Issues", *ACM Computing Surveys*, 19(3), September 1987, pp. 201-258.

[24] Kim, W., et al., "Integrating an Object-Oriented Programming System with a Database System", *Proc. 3rd Int'l Conf. on OOPSLA*, September 1988, pp. 142-152.

[25] Kowalski, R., "Algorithm = Logic + Control", *Comm. of ACM*, July 1979, pp.424-475.

[26] Lam, h., Su, S. and Alashqur, A., "Integrating the Concepts and Techniques of Semantic Modeling and the Object-Oriented Paradigm", *Proc. 13th Int'l Computer Software & Applications Conference (COMPSAC)*, October, 1989, pp. 209-217.

[27] Law, S.F., "Object-Oriented Design and Implementation of a Kernel Object Manager", Master Thesis, Electrical Engineering Department, University of Florida, 1991.

[28] Lindsay, B., et al., "A Data Management Extension Architecture", *Proc. 1987 SIGMOD Conf.*, San Francisco, CA. 1987, pp. 220-226.

[29] Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C., "Abstraction Mechanisms in CLU", *CACM*, 20(8), August 1977, pp. 564-576.

[30] Moss, J., "Nested Transactions: An Approach to Reliable Distributed Computing", MIT Laboratory for Computer Science, MIT/LCS/TR-260, 1981.

[31] Mylopoulos,J., Bernstein, P.A. and Wong, H.K.T., "A Language Facility for Designing Database-Intensive Applications", *ACM Transaction on Database Systems*, Vol.5, No.2, June 1980, pp. 185-207.

[32] Ontologic Inc., "ONTOS Product Description", Manual, MA, 1990

[33] Peckham, J. and Maryanski, F., "Semantic Data Models", *ACM Computing Surveys*, 20(3), September 1988.

[34] Richardson, J. and Carey, M., "Programming Constructs for Database System Implementation in EXODUS", *Proc. 1987 ACM SIGMOD Int'l Conf. on Management of Data*, 1987, pp. 208-219.

[35] Schmidt, J.W., "Some High Level Language Constructs for Data Type Relation", *ACM Trans. on Database Systems*, September 1977, Vol. 2(3), pp. 247-281.

[36] Schaffert, C., et al., "An Introduction to Trellis/Owl", *Proc. 3rd Int'l Conf. on OOPSLA*, September 1988, pp. 9-16.

[37] Shipman, D.W., The Functional data Model and the Data Language DAPLEX", *ACM Transaction on Database Systems*, 6(3), September 1981.

[38] Shyy, Y.M. and Su, S.Y.W., "MPL/0: A Multi-paradigm Language Facility for Data/Knowledge Base Programming", in *Data and Knowledge Base Integration,* Deen, S.M. and Thomas, G.P., (eds.), London, England: Pitman Publishers, 1990, pp. 63-83.

[39] Shyy, Y.M. ,"A Unified Execution Model for Supporting Parallel and Rule-Based Computations in Third Generation Database Programming Languages", working paper.

[40] Committee for Advanced DBMS Function, "Third-Generation Database System Manifesto", SIGMOD Record, Vol. 19(3), September 1990, pp. 31-44.

[41] Smith, J.M., Fox, S., and Landers, T., *ADAPLEX: Rational and Reference Manual*, 2nd edition, Computer Corporation of America, Cambridge, MA., 1983.

[42] Stefik, M. and Bobrow, D., "Object-Oriented Programming: Themes and Variations", *AI Magazine*, 6(4), 1986, pp. 40-64.

[43] Stonebraker, etc., "The Implementation of POSTGRES", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2(1), March 1990, pp. 125-142.

[44] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[45] Su, S.Y.W., "SAM*: A Semantic Association Model for Corporate and Scientific-Statistical Databases", *Journal of Information Sciences*, 29, 1983, pp. 151-199.

[46] Su, S., Krishnamurthy, V. and Lam, H., "An Object-Oriented Semantic Association Model OSAM*", in *Artificial Intelligence Manufacturing Theory and Practice*, S. Kumara et al. (eds.), American Inst. of Indus. Engr., 1989, Chap. 17, pp. 463-494.

[47] Su, S.Y.W., "Extensions to the Object-Oriented Paradigm", *Proc. 13th Int'l Computer Software & Applications Conference (COMPSAC)*, October, 1989, pp. 197-199.

[48] Su, S.Y.W. and Alashqur, A.M., "A Pattern-Based Constraint Specification Language for Object-Oriented Databases", *Proc. IEEE's COMPCON 91*, San Francisco, Feb. 25-March 1, 1991.

[49] Su, S.Y.W. and H.H. Chen, "A Temporal Object-Oriented Data Model and its Query Language", Technical Report, Database Systems R&D Center, University of Florida, 1991.

[50] Wilkinson, K., Lyngbaek, P., and Hassan, W., "The Iris Architecture and Implementation", *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2(1), March 1990, pp. 63-75.

[51] Wirth, N., *Programs = Algorithms + Data Structures*, Prentice-Hall, 1976.

[52] Zdonik, S. and Wegner, P., "Language and Methodology for Object-Oriented Database Environments", *Proc. 19th Annual Hawaiian Conference on Systems Science*, 1986.