The VLDB Journal (1996) 5: 181–195

The VLDB Journal

The design and implementation of K: a high-level knowledge-base programming language of OSAM*.KBMS

Yuh-Ming Shyy, Javier Arroyo, Stanley Y.W. Su, Herman Lam

CSE 470, Database Systems R&D Center, Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL 32611, USA

Edited by Dennis McLeod. Received July 1992 / Accepted August 1995

Abstract. The OSAM*.KBMS is a knowledge-base management system, or the so-called next-generation database management system, for non-traditional data/knowledge-intensive applications. In order to define, query, and manipulate a knowledge base, as well as to write codes to implement any application system, we have developed an object-oriented knowledge-base programming language called K to serve as the high-level interface of OSAM*.KBMS. This paper presents the design of K, its implementation, and its supporting KBMS developed at the Database Systems Research and Development Center of the University of Florida.

Key words: Knowledge-base programming language – Abstractions – Object-oriented knowledge model – Structural associations – Association patterns

1 Introduction

1.1 Motivation

With a view to widening the applicability of database technology to non-traditional application domains such as computer-aided software engineering (CASE), computer-aided design and manufacturing (CAD/CAM), office information systems, and knowledge representation systems, many so-called next-generation database management systems (DBMSs) (Atkinson et al. 1990; Comittee for advanced DBMS Function 1990; Special issue on next-generation database systems 1991) have been proposed in recent years. In general, a next-generation DBMS extends the functionalities of traditional DBMSs (such as persistent data management, query processing, concurrency control, and recovery) in either or both of the following aspects. Firstly, object-oriented data modeling constructs are introduced to model complex application domains, and the behavioral specifications are also incorporated into the domain and functionality of a DBMS in terms of user-defined methods. Secondly, rule management facilities are introduced to manage and process a large number of knowledge rules which maintain the database in a consistent state or trigger some predefined actions when certain events occur. Although object-oriented programming languages can be used for defining object classes and implementing methods, they generally do not have integrated facilities to support persistence, query processing, and the specification and execution of rules. In order to solve the *impedance mismatch* problems (Copeland and Maier 1984) between database languages (which include data definition languages, query languages, and rule languages) and traditional programming languages, next-generation *database programming languages* are also needed for defining, querying, and manipulating a database, as well as for supporting object-oriented and rule-based computations in an integrated fashion. Application objects, methods and rules are integrated in a database which we shall call a knowledge base. In this paper, we shall use the term "knowledge-base management system" (KBMS) and "knowledge-base programming language" (KBPL) to refer to such a next-generation DBMS and database programming language, respectively.

In our earlier research effort, we have developed a prototype KBMS (Lam et al. 1989a, b; Yassen et al. 1991) which used OSAM* (Su 1983; Su 1989; Su et al. 1989; Su and Lam 1992) as its underlying knowledge model, OQL (Alashqur et al. 1989; Guo et al. 1991] as its query language, and the language reported by Alashqur et al. (1990) and Su and Alashqur (1991) as its constraint specification language. In that system, the implementation of methods still needed to be done in such traditional programming language as C++ (Stroustrup 1986). Because the method implementation language does not directly support the OSAM* knowledge model, all the impedance mismatch problems still exist. For example, one cannot combine the programming constructs of the method implementation language and the querying constructs of the query language within a method to directly retrieve and manipulate the database. Moreover, the execution of rules is not well integrated with the execution of methods. To solve these problems, we have developed a single integrated object-oriented knowledge-base programming language called K (Arroyo 1991; Shyy and Su 1991; Shyy 1992) to serve as the high-level interface of a

Correspondence to: S.Y.W. Su

new version of KBMS called OSAM*.KBMS (Su and Lam 1992; Su et al. 1993) for defining, querying, and manipulating the knowledge base, as well as for coding methods and rules of any data/knowledge-intensive application system. In addition to such well-known object-oriented features as abstract data types, information hiding, complex objects, relationships, inheritance, and reusable codes, K provides (1) powerful abstraction mechanisms for supporting the underlying knowledge model which captures any application domain knowledge in terms of the structural associations (such as generalization and aggregation), methods, and knowledge rules, (2) a strong notion of address-independent object identifiers (oid) instead of physical pointers, (3) a persistence mechanism for supporting both persistent and transient objects uniformly without the dangling references problem, (4) a flexible type system which supports both static type checking and multiple views of objects in multiple classes, (5) a declarative knowledge retrieval mechanism based on object association patterns for querying the knowledge base, and (6) basic data structures (set, list, and array) and multiparadigm programming constructs for specifying procedural and rule-based computations.

1.2 Design principles

The design of K is guided by the following general principles.

1. *Direct support of the OSAM*. KBMS kernel knowledge model*. K should provide knowledge abstraction mechanisms to support an extensible kernel knowledge model which will be described in Sect. 2. All the semantic constructs such as classes, associations, methods, and rules of the model should be treated as first class objects in the same way as any other objects in K.

2. *Wide-spectrum for both specification and implementation.* K should be a uniform language for knowledge definition, knowledge retrieval, knowledge manipulation, and general-purpose computation involving persistent/transient objects.

3. *Computationally complete.* K should provide all the basic data structures (set, array, and list), control structures (sequence, repetition, and condition), and rule specification constructs for the users to implement any algorithm and to perform any computation.

4. *Maintainability and readability.* The software written in K should have readable syntax and stable semantics so that it can be easily understood and maintained.

5. *Seamless incorporation of query/rule language.* Instead of simply embedding the existing query and rule language of OSAM*.KBMS into K, a uniform and well-integrated syntax is necessary to provide set-oriented and declarative query and rule specification facilities without any conflict or ambiguity with other programming constructs of K. New constructs should be introduced only if we can demonstrate one or more of the following points: *readability*, *new concept*, and *conciseness*. Moreover, new constructs must satisfy the *orthogonality* principle, i.e., any combination of the programming constructs is allowed.

6. *Strongly typed.* Since K is to be used for the development of complex software systems, it should be a strongly typed language so that as many type errors as possible can be checked by static type checking at compilation time. On the other hand, the type system should be flexible enough to support multiple representations of OSAM* objects in multiple classes as will be discussed in Sect. 2.

7. *More emphasis on functionalities rather than efficiency.* As a high-level programming language, K should put more emphasis on function than efficiency so that complex application systems can be rapidly constructed by the use of those high-level facilities of K. With the rate of hardware progress, we do not feel that efficiency will be a serious concern in the future.

Two versions of K and its supporting OSAM*.KBMS have been implemented on Sun 4 in C++ as a first step toward a complete KBMS-supported software development system (Shyy 1992; Su and Shyy 1993). This paper presents the design and implementation of the latest version of K. Our contribution lies in providing a clean fusion of the techniques introduced in knowledge-base management system, programming language, and software engineering in an object-oriented framework.

1.3 Related works

Many "database programming languages" (Atkinson and Buneman 1987; Bloom and Zdonik 1987) have been proposed in recent years [e.g., Pascal/R, Rigel, Taxis, Dial, Plain, Daplex, Adaplex, PS-Algol, GemStone, Galileo, Trellis/Owel, Vbase, E, Orion, Proquel, O++, OQL[X], Ontos, IRIS, ObjectStore, and O2, as described respectively by Schmidt (1977), Rowe and Shoens (1979), Mylopoulos et al. (1980), Hammer and Berkowitz (1980), Wasserman et al. (1981), Shipman (1981), Smith et al. (1983), Atkinson et al. (1983), Copeland and Maier (1984), Butterworth et al. (1990), Albano et al. (1985), Schaffert et al. (1988), Andrews and Harris (1987), Richardson and Caray (1987), Kim et al. (1988), Lingat and Rolland (1988), Agrawal and Gehani (1989), Blakeley et al. (1990), Ontologic Inc. (1991), Fishman et al. (1987), Wilkinson et al. (1990), Annevelink (1991), Lamb et al. (1991), and Deux et al. (1991)]. Their aim has been to overcome the infamous *impedance mismatch* problem between traditional programming languages and DDL/DML (Copeland and Maier 1984; Maier 1989) by integrating data definition, data manipulation, and general computing facilities in a single language. A detailed survey can be found in the work of Atkinson and Buneman (1987).

Most of the existing works are based on either relational, functional, or object-oriented data models, with the extension of persistence, associative access (using either iterators or SQL-like construct), and the computation facilities of some traditional programming languages such as Pascal, Lisp, and C/C++. They generally do not support rules which is considered to be one of the major requirements for the next-generation database systems. While research works in deductive database systems [e.g., LDL (Chimenti et al. 1990), LOGRES (Cacace et al. 1990), and Glue-Nail (Phipps and Derr 1991)] and active database systems [e.g., Postgres (Stonebraker and Kemnitz 1991), Starburst (Lohman et al. 1991), Ariel (Hanson 1989) and HiPAC (Dayal 1989; Chakravarthy 1989) have extended relational or object-oriented database

systems with rules, they provide separate rule languages as extensions of their query languages instead of integrated database programming languages.

Among the existing works, K is most closely related to O++ (Agrawal and Gehani 1989; Gehani and Jagadish, in press); both provide persistence, querying, and rule facilities in an object-oriented framework. O++ extends C++ with the facilities for creating persistent and versioned objects, defining sets, iterating over sets and clusters of persistent objects, and associating constraints and triggers with object classes. Unlike O++, which is a superset of C++, K is designed to be a *high-level* programming language with the following differences. Firstly, while O++ extends C++ data model with rules, K supports a high-level knowledge model OSAM* where everything, including classes, associations, methods, and rules, is all uniformly treated as objects. A user can use the query facility of K to query the meta-information from the kernel schema in the same way as one queries any application domain. Secondly, while O++ extends the "for" loop construct to iterate over sets, K provides more declarative and concise constructs for specifying queries and rules based on *object association patterns* (Alashqur et al. 1989; Guo et al. 1991). Thirdly, K uses address-independent object identifiers oids (soft pointers) as object surrogates rather than using physical address pointers (hard pointers). Persistent and transient objects are transparent to the users and are treated in the same way. For example, a query will retrieve both types of objects instead of only persistent objects as in O++. Fourthly, K provides a more flexible type system which supports both static type checking and multiple representations of objects in multiple classes which is not possible in O++. Lastly, more emphasis is put on readability and maintainability by providing readable syntax rather than using the non-intuitive syntax of C++.

1.4 Paper organization

The rest of this paper is organized as follows. Section 2 gives an overview of K in terms of the underlying knowledge model, knowledge definition facilities, persistence, and type system. Query processing facilities are described in Sect. 3 in terms of object association patterns, context looping statements, and existential and universal quantifiers. Procedural and rule-based computation facilities are described in Sect. 4. Section 5 describes the computation model of K. The system architecture and the implementation of K and its supporting OSAM*.KBMS are given in Sect. 6. Section 7 gives our conclusion and the future research directions. A parts-manufacturing knowledge base, as suggested by Atkinson and Buneman (1987), is given as an example throughout this paper to illustrate the expressiveness of K.

2 Language overview

In this section, we present an overview of the features of the K language in terms of its knowledge model and knowledge definition facilities, its support of persistence, and its underlying type system. The knowledge model of K is an extensible object-oriented model in which classes are used as an abstraction to classify objects. Semantic relationships among objects are modeled by semantic associations; different types of semantics are modeled by different association types. Behavioral properties of objects can be defined declaratively by using knowledge rules, or procedurally by using methods. In K, persistence is an object property rather than a class property, i.e. objects in the same class could be either persistent or transient. Persistence is transparently supported, thus relieving the application developer from the burden of specifying any procedure or data structure for storage management.

2.1 Knowledge model

2.1.1 Semantic constructs

Classes. We use *classes* as the knowledge abstraction unit to classify objects by their common structural and behavioral properties in an integrated fashion. Classes are categorized as *entity classes* (E-Classes) and *domain classes* (D-classes). The sole function of a domain class is to define a domain of possible *values* from which descriptive attributes of objects draw their values. Both *primitive* domain classes (e.g., Integer, Real, and String) and *complex* domain classes (e.g., Date and Address) are supported. 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, functions, events, processes, and relationships. The structural properties of each object class (called the *defining class*), and thus its instances are uniformly defined in terms of its *structural associations* [e.g., aggregation and generalization (Smith and Smith 1977)] with other object classes, called the *constituent classes*). Each type of structural association represents a set of generic rules that govern the knowledge-base manipulation operations on the instances of those classes that are defined by the association type. Manipulation of the structural properties of an object instance is done through methods, and the execution of methods is automatically governed by rules to maintain the system in a consistent state or to trigger some predefined actions under certain conditions. In other words, the behavioral properties of an object class are defined as methods and rules applicable to the instances of that class. 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. A *schema* is defined as a set of class associations.

Objects and instances. In the kernel model of K, objects are categorized as domain class objects (DClassObject) and entity class objects (EClassObject). Domain class objects are self-named objects which are referred to by their values. Entity class objects are system-named objects each of which is given a unique oid. We adopt a *distributed* view of objects to support generalization as by Lam et al. (1989) and Yassen et al. (1991) by visualizing an instance of class X as the *representation* (or *view*) of some object in the class X. Each object can be instantiated in different classes with different representations but with the same oid. Each instance is identified by a unique *instance identifier* (iid) which is the concatenation of cid and oid, where cid is a unique number

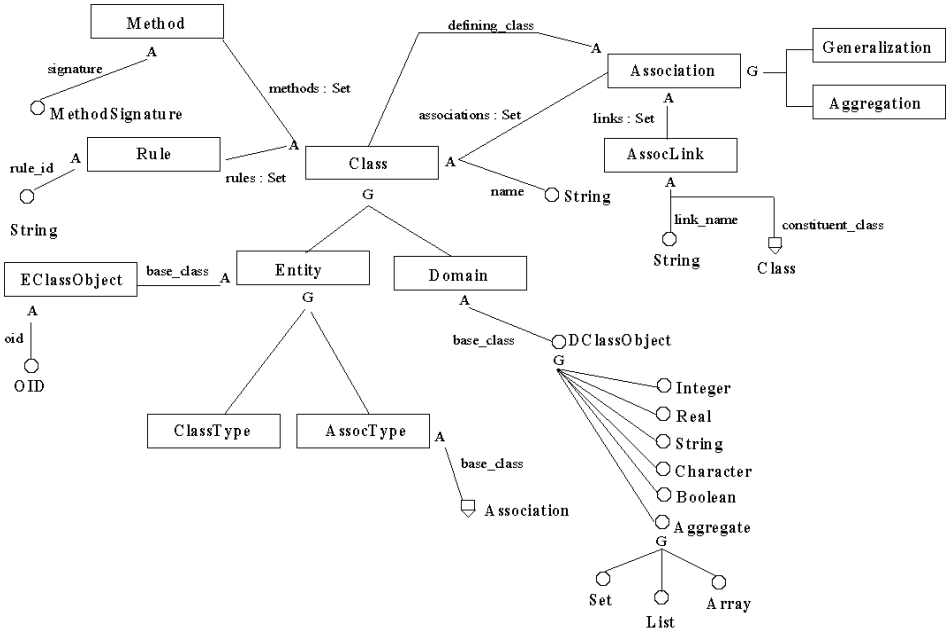

Fig. 1. Overview of the kernel class structure of K

assigned for each class in the system and is defined as the *type* of this instance. For two entity classes A and B, if class A is a superclass of class B (generalization association), then for each object which has an instance in the class B, it must also have an instance in the class A. Both instances have the same oid and are conceptually connected by a generalization association link. A detailed discussion of the advantages of the distributed view of objects can be found in the work of Yassen et al. (1991). Each entity class is associated with an *extension* which is the set of all its instances.

Encapsulation and inheritance. We adopt the C++ three-level *information hiding* mechanism (Stroustrup 1986) by classifying aggregation associations (which are referred to as *attributes*, *data members*, or *instance variables* in other object-oriented programming languages to describe the *state* of an object instance) and methods as either "public", "private", or "protected". All the rules associated with a class are treated as protected by definition. At the class level, all the rules and public/protected aggregations and methods defined by a class are *inherited* by its subclasses. At the instance level, an instance of entity class A stores only the attributes defined for A, and it inherits (i.e., gets access to) all the public/protected attributes from its corresponding instances (with the same oid) of all the superclasses of A.

2.1.2 Model extensibility

Model extensibility is achieved via a *self-describing* kernel model, shown in Fig. 1, in which all the data model constructs such as classes, associations, methods, and rules are modeled as first-class objects. One can extend the data model by modifying this set of meta-classes. This kernel model also serves as the *dictionary* of the OSAM*.KBMS as all the object classes in the system are mapped into this class structure. One can therefore browse and query any user-defined schema as well as the dictionary uniformly. In our graphic *schema notation*, (1) entity classes and domain classes are represented as rectangular nodes and circular nodes, respectively, (2) a generalization association is represented by a G link from a superclass to a subclass, and (3) an aggregation association is represented by an A link from the defining class to a constituent class. After compilation, any user-defined class will be added to the class structure as an immediate or non-immediate subclass of either EClassObject or DClassObject, while at the same time the objects corresponding to the class definition, associations, methods, and rules of the defining class will be created as instances of the system-defined entity classes named Class, Association, Method, and Rule, respectively. Note that this class structure is self-describing in the sense that we use the model to describe the model itself.

As any application domain (including the model itself) is uniformly modeled and mapped into the kernel model, one can use the kernel model to *incrementally* extend the model itself to meet the requirements of various applications by either (1) adding new structural association types [e.g., Interaction, Composition, and Crossproduct Su et al. (1989)] or introducing subtypes of existing association types, or (2) extending the definition of existing association types [e.g., add new attributes default_value, null_value, optional, unchangeable, dependent (Shyy et al. 1991), etc., for the association type Aggregation] so that more semantics can be captured in the schema and maintained by the KBMS instead of being buried in application codes. Once a new association type is defined, it becomes a semantic construct of the extended knowledge model and can be used in the definition of any object class.

2.2 Knowledge definition facilities

In Atkinson and Buneman (1987), a set of four tasks is proposed to evaluate the expressiveness of database programming languages using a manufacturing company's parts database. The four tasks are:

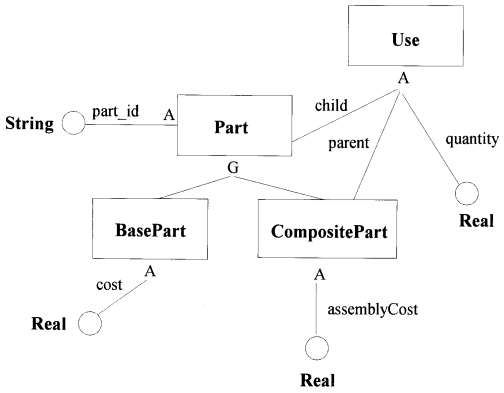

Fig. 2. The parts knowledge base schema

1. Describe the database
2. Print the name and cost of all base parts that cost more than $100
3. Compute the total cost of a given composite part
4. Record the decomposition of a new part in the database, i.e., how a new composite part is manufactured from its subparts

This example has been successfully implemented in K. In this section, we give an overview of the knowledge definition facilities of K by showing how to use K to perform the first task, Describe the database. The other three tasks will be described in the following sections after the appropriate programming constructs have been described. The schema of the parts knowledge base is shown in Fig. 2, and the corresponding class definition in K is shown in Fig. 3. We define entity classes Part, CompositePart, and BasePart to model the parts, and an entity class Use to model the many-to-many relationships among parts. Each Use instance has three attributes, parent, child, and quantity, to record the relationship that the parent part uses a certain number of the child part. Each part has an attribute part_id and must be either a base part or a composite part. A base part has an attribute cost, and a composite part has an attribute assemblyCost.

2.2.1 Association definition

Based on the knowledge model described in Sect. 2.1, each class definition in K generally consists of an *associations* section, a *methods* section, and a *rules* section, as shown in Fig. 3. The following two kernel structural association types are supported in K. Other association types can be introduced by the model extensibility feature of the KBMS.

Aggregation. For each object class, one can define a set of aggregation associations (attributes) to describe the *state* of its instances, as shown in Fig. 3. Each aggregation-group is specified by a protection or encapsulation level (*public:*, *private:*, and *protected:*, as described in Sect. 2.1) followed by a list of aggregation specifications. Each aggregation specification corresponds to an instance of the class Aggregation. An aggregation association defines either (1) a *value attribute* if its constituent_class is a domain class, or (2) a *reference attribute* if its constituent_class is an entity class. At the instance level, we store values and iids for value attributes and reference attributes, respectively. *Multi-valued* attributes are specified using the constructor classes Set, List, or Array. Note that in addition to the constructor Set which is critical in object-oriented databases, we also provide List and Array to capture the semantics of order which is useful in real-world applications (Atkinson et al. 1990; Committee for advanced DBMS Function 1990). Note that an aggregation association between two entity classes is interpreted as a *bi-directional* link. For example, for the aggregation association called parent from entity class Use to entity class Part, the system will automatically define and maintain a set-valued aggregation from Part to Use (for each Part instance, the system records all the Use instances that refer to this Part instance). The system will use this information to support bi-directional navigation and to maintain the *referential integrity* of the knowledge base. Note that similar to the ER model (Chen 1976), when an association is expected to carry descriptive data or behavioral information (in terms of methods and/or rules), the knowledge-base designer can explicitly model the association as an entity class.

Generalization. For each object class, one can use the generalization (G) association to specify its immediate superclass. Note that the generalization association is bi-directional and can be specified in either direction. When class B is specified as a subclass of class A, we say that class B is a *specialization* of class A. For example, to say "BasePart is a specialization or subclass of Part" is equivalent to saying "Part is a generalization or superclass of BasePart". In general, specialization is used to construct object classes in a top-down and step-wise refinement approach by giving more and more structural and behavioral properties. Note that we also allow the user to define specializations of primitive domain classes (e.g., a subset of integer) by using rules to specify range, enumeration, or other constraints.

2.3 Persistence

K supports persistence so that objects can live after the execution of a program is terminated. Persistence in K is based on the following rationales:

1. Persistence is *orthogonal* to classes as in E (Richardson and Carey 1987), O++ (Agrawal and Gehani 1989), ONTOS (Ontologic Inc. 1991), and OQL[X] (Blakeley et al. 1990), i.e., persistence is an object property rather than a class property. Any subclass of E_Class_Object automatically inherits the persistence mechanism and it is up to the user to specify each of its instances to be either persistent or transient using the *pnew* or *tnew* operator, respectively, when creating a new object. When a transaction is terminated, the object manager will delete all the transient objects and remove all the references to these transient objects by following the inverse aggregation links mentioned in Sect. 2.2. Contrary to other existing systems, K automatically takes care of the dangling references problem (i.e., some transient instances are referred to by persistent instances after program execution is terminated) instead of leaving it to the user's responsability.

2. Persistence is orthogonal to oid. The advantages are twofold. Firstly, K provides a better object-oriented flavor

```
define CompositePart : Entity
  associations:
    Specialization { Part };
  public:
    Aggregation
    {
       assemblyCost : Real;
    }
  methods:
    public:
      method getCost() : Real;
      method askUse();
       method askChildPid(u : Use);
      method askChildQty(u : Use);
end CompositePart;


define BasePart : Entity
  associations:
     Specialization { Part };
end BasePart;


define Use : Entity
  associations:
    public:
      Aggregation
      {
         parent : CompositePart;
         child  : Part;
         quantity : Real;
      }
end Use;
```

```
define Part : Entity
  associations:
    public:
      Aggregation
      {
         part_id : String;
         cost : Real;
      }

  methods:
    method deleteDependents();

  rules:
    rule comp_part_no is
    triggered after create(), immediate_after update(part_id)
    condition (this.is_a(CompositePart), this.part_id != "" |
                        part_id[1] = "C")
    otherwise
          "All composite part ids must start with a C\n".display();
          this.del();
    end;

    rule uniquePid is
    triggered immediate_after update(part_id)
    condition exist p in p:Part where p.part_id = this.part_id
                      and p != this
    action "Part_id cannot be redundant".fdisplay("*** %s *** \n");
              this.del();
    end uniquePid;

    rule delDependents is
    triggered before del()
    condition exist u in this *<[child] u:Use
    action
        this.deleteDependents();
    end delDependents;
end Part;
```

Fig. 3. Specification of the parts knowledge base in K

than C++-based languages by enabling the users to (1) manipulate objects at the logical level instead of going to the physical level, and (2) navigate through the database using oids instead of pointer chasing. Secondly, as oids are independent of physical address, K enforces the *immutability* requirement of identity (Khoshfian and Copeland 1986). Therefore, unlike C++-based persistent languages which put extra burden of managing two or three types of pointers (persistent, transient, and/or dual pointers) on the users, persistence in K is transparent to the user in managing entity class instance variables.

3. Persistence is orthogonal to queries and any object manipulation. For example, a selection query over an entity class should return both its persistent and transient instances as long as they satisfy the selection condition as in Blakeley et al. (1990).

2.4 Type system

K directly manipulates objects at the instance level. We define the *type* of an instance as the class to which this instance belongs. Every variable in K is bound to some instance and therefore must be declared to have a type. Similarly, the result of an expression in K is an instance whose type is determined by the return type specified for the methods and operators that are invoked in the expression.

2.4.1 Type compatibility

The type of an expression in general can be detected by textual inspection to decide on the type compatibility which means that a variable of type X can only be assigned expressions which represent instances of class X or any subclass of X. In the latter case, the system will automatically convert the type of the instance from the right-hand-side expression to class X during run time to actually refer to it as an instance of class X. Method parameters and returned values are checked against the method signature following the same rule as above.

2.4.2 Type conversion

If the type checker is not able to ascribe a type to an expression, the user must use the *cast* operator $ to specify the type in the form <class>$<expression>. The cast operator is useful for the user to temporarily convert the type of an expression or refer different representations of the same entity class object in different classes. For example, suppose p is an instance variable of type Part, then BasePart$p asserts that the type of p is BasePart instead of Part. Similarly, to resolve any *name conflict* in inheritance, one must specify from which superclass a particular property is inherited, by casting the type of an expression to that particular superclass to refer to the corresponding instance explicitly. For example, suppose both classes Part and BasePart define a method called getCost(). To invoke getCost on a BasePart instance b, one must use Part$b.getCost() for the system to unambiguously invoke the correct method to corresponding Part instance of b. Note that in the case when no name conflict occurs, the system will automatically find the appropriate superclass and perform the casting to support inheritance. In other words, inheritance at run time is supported by casting an instance of class X to be an instance of class Y (which is a superclass of class X) before accessing a property defined by class Y.

For domain classes, only *upward* type casting is allowed, i.e., a value from the domain class X can be converted to a value of any superclass of X, but not vice versa. For entity classes, type casting can be performed either upward or downward along a generalization hierarchy. Note that a null value might be returned from a cast operator in the cases when there does not exist any corresponding instance in the target class. For example, BasePart$p will return null if the Part instance referred to by p has a corresponding instance in CompositePart rather than in BasePart. It would be the user's responsibility to handle null values explicitly.

2.4.3 KBMS operations

After an entity class is defined, we can insert instances into that class. An instance can be created from scratch by invoking the tnew or pnew methods (defined in the meta-class Entity of Fig. 1), followed by a list of attribute assignments, to create a new transient or persistent object along with an instance of this object. The following K-program block illustrates the basic OSAM*.KBMS operations and the concept of object/instance by using our example parts knowledge base:

```
local
        p1:Part,
        p2: Part,
        c: CompositePart,
        b: BasePart;
begin
        p1 := Part.tnew() { part_id := "LS741" };      // (1)
        b := basePart.pnew()
             { part_id = "LS745" , cost := 25.00 };    // (2)
         c := p1.insert(CompositePart);                // (3)
         p2 := Part$c;                                 // (4)
         c.del();                                      // (5)
         p1.destroy();                                 // (6)
end;
```

Statements (1) and (2) create two new objects, insert their instances in the class Part and class BasePart, update their attribute values, and return the iids to variables p1 and b, respectively. Note that by inserting a BasePart instance, the system object manager automatically inserts a corresponding Part instance (with the same oid) because of the generalization association. The difference between the instances referred to by "b" and "p1" is that the former is a persistent instance and therefore any update (using the assignment operator) will be written to the database, while the latter is a transient instance which resides only in main memory. Notice that the pnew and tnew methods are invoked using conventional dot notation following the name of the class. This is interpreted by K as follows: return the instance of the meta-class Entity corresponding to the given class, and invoke the method pnew or tnew for that instance. This is similar to SmallTalk "class methods" or C++ "static member functions". Statement (3) inserts a new CompositePart instance of the object referred to by variable p1 (assume we learn that this part is a CompositePart) and returns the iid to variable c. Note that no new object is created and the iid returned to c has the same oid as p1. Statement (4) casts the CompositePart instance referred to by c as a Part instance whose iid is then assigned to p2. In statement (5), we delete the instance c and the object manager will automatically delete all the instances of the object referred to by c in all the subclasses of CompositePart (if any) following the generalization association. All the references (association links) to these deleted instances from other object instances will be automatically removed to maintain the *referential integrity* constraint. Note that for this particular object, even though it lost its instance in CompositePart, it still has its instance in Part. For example, we can use the variable p2 from statement (4) to refer to its Part instance. In other words, we allow an object to have different representations in different classes (even spans more than one branch of the generalization lattice), and we can insert or delete these representations dynamically. This is a property that cannot be expressed in most object-oriented programming languages except those systems which support multiple views of objects [e.g., Aspect (Richardson and Schwartz 1991), IRIS (Fishman et al. 1987; Wilkinson et al. 1990; Annevelink 1991), and Clovers (Stein and Zdonik 1989)]. At the implementation level, it is easier to delete an instance without extra copying data and changing addresses. Note that we achieve this flexibility without losing the advantages of static type checking by directly manipulating instances rather than objects. In statement (6), a destroy statement will automatically delete all the instances of the object referred to by variable p1, including those in the superclasses of Part (if any). An implementation of the object manager has been reported by Arroyo (1991).

3 Query processing facilities

Object retrieval in K is based on structural association patterns among object classes in the form of (1) a context looping construct for querying and manipulating the knowledge base, and (2) existential and universal quantifiers for posing logical questions upon the knowledge base.

3.1 Association patterns

Since K serves as a high-level interface of OSAM*.KBMS, the execution of a K program would generally involve the processing of a persistent knowledge base. For knowledge-base retrieval and manipulation, a knowledge-base programming language should include some knowledge manipulation constructs in addition to general programming constructs. In our work on K, we use pattern-based querying constructs for this purpose. We modify the context expression of OQL (Alashqur et al. 1989; Guo et al. 1991) as the primitive construct for specifying structural association patterns based on which the system can *navigate* through the knowledge base to identify the corresponding *contexts* (sub-knowledge-bases) that satisfy the specified *intensional* patterns.

In general, each association pattern is specified by the *context* clause which has a set of classes and operators, and optional *where* and *select* clauses. Operators can be either association operators like *, *>, and *<, or non-association operators like !, !<, and !>. The > and < symbols that follow the association operators are used to explicitly indicate a

direction, which is the defining class of a given association, where > stands for left-to-right, and < stands for right-to-left direction. For example, the pattern *context* Part *> CompositePart specifies all the Part objects that are associated with some CompositePart objects, where Part is the defining class of the association (in this case an unnamed Generalization association). Similarly, the pattern *context* CompositePart !< [parent] Use specifies all the Use objects that are not associated to any CompositePart objects through the association link called parent, defined in the class Use.

One can also explicitly specify a range variable over a class in an association pattern. For example, *context* p1:Part *> c:CompositePart !< [parent] Use *> [child] p2:Part *where* p2.part_id='LS741' *select* p1 specifies a subknowledge-base that contains all the parts that are composite parts (i.e., there is generalization link connecting a part with a composite part) which do not use (i.e., composite parts not connected through the parent association with any Use instance) part number 'LS741' (i.e., Use instances connected through the child link to part whose part_id equals 'LS741'). Here, p1 and p2 are variables that represent the Part instances that satisfy the association pattern specification. Notice that the name of an association link can be explicitly stated within square brackets following an association operator. Notice also that a select clause is used to perform a projection operation over the class Part referred to by variable p1.

Instead of using a class notation, one can also directly *designate* objects by replacing class name with any user-defined variable which is bound to a single or a collection of entity class instances. For example, s : Set of Part declares a variable s whose value will be a set of Part instances, and s *< [parent] Use specifies a context which consists of all the parts denoted by s that are the parents in the relationship defined by the class Use. Note that both *implicit* sets (denoted by a class name followed by an optional selection condition in a context expression) and *explicit* sets (user-defined set variables as the above example) are supported in K. Explicit sets can be manipulated by using set operators + (union), & (intersection), and − (difference). One can also use the overloaded + (add) and − (remove) operators to add and remove a single instance to and from a set, respectively. A more detailed description of complex association patterns can be found by Shyy (1992).

Note that in existing object-oriented database systems, navigation is expressed by using the *dot expression* for implicit joins. However, the use of dot expressions is limited by the following factors. Firstly, navigation is done only in one direction unless *inverse* attributes are supported and explicitly defined in the system. For example, suppose class Part defines an aggregation association whose name is subPart and whose constituent class is Part itself. Then, one can use the association patterns (1) this *> [subPart] p:Part to identify the subparts of a particular Part denoted by this, and (2) this *< [subPart] p:Part to identify all the parts of which this is a subpart. Such bi-directional navigation is not possible in a dot expression. Secondly, navigation cannot continue after a multi-valued attribute is met. For example, the dot expression c.components.quantity is not allowed, since components is a multi-valued aggregation association defined from CompositePart to Use. Thirdly, dot expressions cannot express navigation via *negation*, or the *non-associate* relationships. For example, it is not possible to express all the parts that composite part c *does not* use using a simple dot expression. K supports all the above cases using association patterns.

3.2 Context looping statement

A context corresponding to an association pattern can be thought of as a normalized relation whose columns are defined over the participating classes and each of its tuples represents an *extensional* pattern of iids that satisfy the intensional pattern. After a context is identified, one can use the context looping *do* statement provided by K to manipulate objects over each extensional pattern. For example, the following statement will print the part_id of each composite part whose assembly cost is greater than the cost of any of its components:

```
context c:CompositePart *< [parent] Use *> [child] p:Part
   where c.assemblyCost > p.cost select c do
      c.part_id.fdisplay("Composite part id: \n");
end;
```

Note that the where-clause is used to specify the inter-class selection condition between CompositePart and Part and the select-clause is used to project over CompositePart column and remove the redundant tuples so that each qualified CompositePart will appear only once even if it has more than one component. Also note that c.part_id returns a string and fdisplay() is a method of the domain class String with the same format notation as the C printf function.

Similarly, the second task of the Parts database example (Atkinson and Buneman 1987M i.e., to print the name and cost of all the base parts which cost more than $100) can be expressed straight-forwardly as follows:

```
context b:BasePart where b.cost > 100.00 do
   b.part_id.fdisplay(" Base Part %s );
   b.cost.fdisplay(" with cost = %5.2f \n");
end;
```

As another example, the following statement will print the name of all the parts which are not used by any composite part. Note that the use of the ! non-associate operator provides a more concise expression because, otherwise, we have to iterate over each part and, for each looping, we have to in turn iterate over each composite part to test if any composite part uses this particular part as a subpart:

```
context p:Part !<[child] Use *> [parent] CompositePart do
   p.part_id.fdisplay("Part %s is not used by any
                       composite part\n");
end;
```

3.3 Existential and universal quantifiers

Statements for the retrieval and manipulation of a knowledge base may involve *existential* and *universal* quantifiers. Quantifiers make it much easier for the users to declaratively and concisely pose logic questions upon the knowledge base. For

example, one can ask if there exists a part, used by a composite part, whose cost is greater than the assembly cost of the composite part by *exist* p *in* c:CompositePart *< [parent] Use *> [child] p:Part *where* p.cost > c.assemblyCost. Similarly, one can ask if *all* the CompositeParts whose assembly cost is greater than the cost of its components have an assembly cost greater than $100 by *forall* c *in* c:CompositePart *< [parent] Use *> [child] p:Part *where* c.assemblyCost > p.cost *suchthat* c.assemblyCost > 100.00. Notice that the *suchthat* clause in the *forall* expression is mandatory since, given an association pattern, it is necessary to check that a given condition is satisfied by all the objects that satisfy the condition specified in the *where* clause. In the case of *exist*, it is only necessary to check for the existence of at least one object that satisfies the condition specified in the *where* clause. Note that by following the *orthogonality* principle, both the context looping statements and quantifiers are treated as normal K statements and expressions, and they can be nested in an arbitrary number of levels or combined with other boolean expressions. For example, the following statement will print the name of all the composite parts which use only base parts:

```
context p1:CompositePart do
  if forall p2 in p2:Part *<[child] Use *> [parent] p1
    suchthat not p2.is_a(BasePart) then
      p1.part_id.fdisplay(Part %s uses only base part\n);
  end_if;
end;
```

Note that we use an if-then-else statement inside the context looping statement, which uses a universal quantifier as its test condition. Also note that since the type of p2 is Part, we invoke the is_a method to test if p2 is a BasePart.

4 Object-oriented and rule-based computation facilities

Both object-oriented and rule-based computations are supported in K by the use of methods and rules as will be described in Sects. 4.1 and 4.2, respectively. Corresponding to each executable software system, the user has to define a named K *program* (similar to the "main" program of C++) as the starting point of execution as will be described in Sect. 4.3.

4.1 Method definition

Each method definition consists of two parts: (1) a *signature* which is given in the methods section of a class definition and specifies the name of the method, the type of the parameters, and the type of the return value, and (2) the actual program *body* which is given in the implementation section of a class definition and is a sequence of K statements that contains local variable declarations and general computations. Both method and operator overloading are allowed in K.

As a computationally complete programming language, K provides the basic data structures (set, list, and array) and control structures (sequential, condition, repetition, and context looping). In this section, we use the third task of the parts knowledge base (Atkinson and Buneman 1987) as an example to illustrate the object-oriented computation facilities of K. A detailed and complete description of the language constructs can be found in the work of Shyy (1992). Note that since the current version of K does not support recursive queries, we define a method getCost() of class CompositePart, which will recursively call itself to make a depth-first traverse of all the immediate or non-immediate subparts of a given composite part and return the total cost.

```
method CompositePart::getCost() : real is
  local sum : real;
  begin
    sum := this.assemblyCost;
    context this *>[components] u:Use *>[child] p:Part do
      case
      when p.is_a(BasePart) do
        sum := sum + p.cost * u.quantity;
      when p.is_a(CompositePart) do
        sum := sum
              + CompositePart$p.getCost() * u.quantity;
      end_case;
    end;
    return sum;
  end;
end getCost;
```

The program body of getCost is a single *local* statement, where we define a local variable sum to record the total cost. We first initialize the local variable sum to be the assemblyCost value of the given composite part (which is the receiver of this method and denoted by the pseudo-variable this, like in C++). We then use a context looping statement described in Sect. 3.2 to retrieve the immediate subparts used by this. For each subpart, we use a case statement to compute its contribution to the total cost (which is the multiplication of its own cost and the quantity used) based on whether this subpart is a base part or composite part. This condition is tested by the use of the is_a method. If it is a base part, its own cost can be directly returned as the value of its cost attribute; if it is a composite part, we recursively call getCost to compute its own total cost. No otherwise clause is used in the case statement because each part must be either a base part or a composite part.

4.2 Rule definition

Rules serve as a high-level mechanism for specifying declarative knowledge that governs the manipulations of objects made by KBMS operations, updates, and user-defined methods. Note that, although the semantics represented by rules can be implemented in methods, high-level declarative rules make it much easier for a database designer to clearly capture the semantics instead of burying the knowledge in the implementation codes and thus simplify the tasks of implementation, debugging, and maintenance. Moreover, rules can be used to dynamically modify the control flow among program modules without having to modify the codes of each module and thus improve the system modularity.

Each rule is given a name for its identification, which must be unique within its defining class. A rule is specified by a set of trigger conditions and a rule body. Each trigger condition consists of a timing specification and a sequence of

knowledge-base event specification. A timing specification (or coupling mode) can be *before*, *after*, or *immediate_after*. An event specification can be a KBMS operation described in Sect. 2.4 or a user-defined method. Note that in the case of a name conflict in multiple inheritance caused by a redefined attribute or method, the user must attach the proper class name with the attribute or method name to unambiguously specify the knowledge-base event. The rule body consists of a *condition* clause, an *action* clause, and an *otherwise* clause, both of which can be a sequence of any K computation statement. The *condition* clause of a rule may contain any valid K boolean expression, and may return either *true*, *false* or *skip*. If a rule condition is *true*, then the action part of it is executed. If it returns *false*, then the *otherwise* part is executed. If *skip* is returned, then the whole rule is skipped. A condition may return *skip* if a *guard expression* is specified. A guard expression is in the form (guard1, guard2,...,guardN | target). Each guard in the expression is evaluated from left to right, and the evaluation stops as soon as one of the guards evaluates to *false*. The evaluation of a guard expression can return either (1) true, if all the guards and the target (all of which are boolean expressions by themselves) are true, (2) skip, if any of the guards are false when they are evaluated from left to right, or (3) false, if all the guards are true but the target is false. Although the semantics of a guard expression can be implemented by nesting of if-then-else constructs, the guard expression is a simpler and more concise construct to use, particularly when the number of guards is large. Besides, we feel that rules should be specified as declaratively as possible, and we would like to make a clear distinction among the condition, action, and otherwise parts of a rule instead of mixing them in a nested if-then-else procedural statement. Similarly to method invocation, rule checking is performed at the instance level, and the pseudo-variable this can be used in a rule body to represent a certain instance of the defining class to which some event occurs.

Figure 3 presents examples of K rule specifications. The rule comp_part_no defined in the class Part is triggered either after a new part instance is created (or before committing the transaction where this operation was performed), or immediately after an update is done to attribute part_id. It uses a guard expression to state that if a Part is a CompositePart, and the part_id is not a null string, then the part_id should start with a C, otherwise the part will be deleted from the knowledge base. The deletion operation is performed by invoking the del method.

All the rules are assumed to be *active* when a user session begins. However, during the execution of a user program, one can invoke the activate or deactivate methods to temporarily activate or deactivate any particular rule, respectively. For each knowledge-base event occuring to instance this of class X, all the applicable rules will be triggered (i.e., the evaluation of the rule body) according to the trigger conditions of each rule (1) before the triggering event, (2) immediately after the triggering event, or (3) not immediately after the triggering event, but at the end of the parent event that causes the triggering event. Note that the use of the after mode allows for temporary violation of constraints (which is likely to happen when a constraint on an object depends on two inter-related values and when one of the values is updated) by deferring the rule checking until the end of a higher level operation. In the case that multiple rules satisfy a trigger condition, such rules will be triggered in some unspecified order which is dependent on the implementation.

Another example of a rule specification is presented in Fig. 3. The rule uniquePid in the class Part specifies the constraint that part_id is the user-defined *key* of class Part, i.e., each part should have a unique part_id. This rule will be triggered immediately after the part_id attribute value of a Part instance denoted by the pseudo-variable this is updated. We use a existential quantifier in the condition-clause, which tests if there exists a part (denoted by the range variable p) which has the same part_id as this, but does not have the same iid as this, i.e., p != this. The action clause will be executed if the condition clause returns true, i.e., if there exists some part with the same part_id as this.

Note that rules specified in a class definition can conflict with other rules for the same knowledge-base event or cause infinite looping during execution. This is a knowledge-base *validation* problem (Wu 1993) which is not in the scope of this paper. We assume that it is the user's responsibility to make sure that such logic errors do not happen as in the work of Gehani Jagadish (in press). We will present the rule execution model in more detail in Sect. 5.

4.3 An example

An application can be specified and implemented in K by uniformly modeling as object classes all the objects used by the application, and the components (software modules) of the application itself. For example, we can define an entity class PartHandler as the top-level software system that manipulate the parts knowledge base as shown in Fig. 4. Each class definition is represented by a .k file, and one can use the include statement to include the necessary files for compilation. Each schema can also be represented by a .k file which contains a list of include statements to include all the related classes into one module. Note that PartHandler includes a file PartSchema.k, which in turn includes all the classes Part, BasePart, CompositePart, and Use. The functionality of PartHandler is represented by a method called main, which displays a menu and asks the user to choose among various tasks (create a new base part, create a new composite part, delete a part, display a part, etc.).

In addition to the definition of object classes, one also has to define a named K program as the starting point of execution. In general, a K program contains few statements which create instances of the entity classes that model the software system, and invoke the main method on the newly created instances to obtain the functionality as shown in Fig. 4. After compilation, each K program is translated into an executable file, which can be activated by just typing in the program name at the Unix shell.

Note that because the user is allowed to manipulate the knowledge base only via the interface of PartHandler, certain system constraints can be implicitly enforced by PartHandler. For example, by providing the user with only the options to create a base part and a composite part, we enforce the *total participation* and *set exclusion* constraints (Su et al. 1989) that each part must be either a base part or

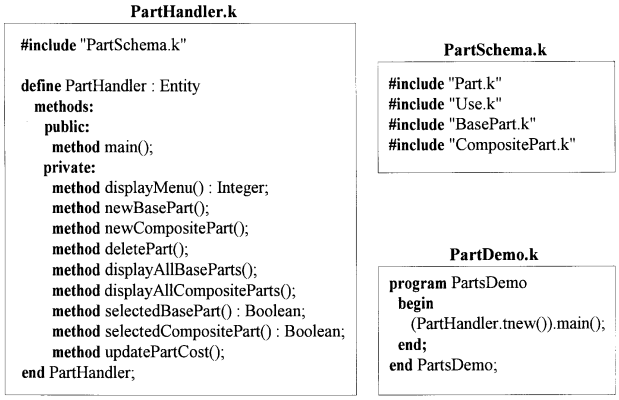

Fig. 4. Software system specification in K

a composite part. Some other constraints and triggers can be explicitly specified as rules. For example, as shown in Fig. 3, a rule delDependents enforces the constraint of class Part that, before a part is deleted, all the composite parts which directly or indirectly use this part must also be deleted. The following is the implementation of the method deleteDependents:

```
method Part::deleteDependents() is
  context this *< [child] u:Use *>
                  [parent] c:CompositePart do
    u.del();
    c.del();
end deleteDependents;
```

The fourth task of the parts knowledge-base example, i.e., to create a new composite part in the knowledge base, can now be easily expressed. When the user selects the new composite part option from the main menu, PartHandler will in turn invoke its private method newCompositePart to perform this task. The method newCompositePart will create a new persistent CompositePart instance, and invoke the methods askPid (which is inherited from class Part) and askUse (which is defined by class CompositePart) to ask the user to provide the part_id and subparts information. The implementation of newCompositePart and askUse is shown as follows:

```
method PartHandler::newCompositePart() is
  local c : CompositePart;
  begin
    c := CompositePart.pnew();
    c.askPid();
    c.askUse();
  end;
end newCompositePart;

method CompositePart:askUse() is
  local
    u : Use,
    more : String;
  begin
  Any subpart? (y/n) ==> .fdisplay( /n %s );
  more.read(); /* read the input from the terminal */
  case
    when more = y do
      begin
        u := Use.pnew(); /* create a new Use instance */
        u.parent := this;
        this.askChildPid(u); /* ask the subpart */
        this.askChildQty(u); /* ask the quantity used */
        this.components := this.components + u;
            /* add element to a set */
        this.askUse(); /* ask again */
      end;
    when more = n do
      return;
    end_case;
  end;
end askUse;
```

Note that in the implementation of askUse, we use a method called read, which is a method of the system-defined domain class String, to obtained the string value typed in by the user. Also note that we use the askChildPid and askChildQuantity methods to record each subpart information in a Use instance. Each Use instance will then be added (using the + operator) to the components attribute (whose value is a set of Use instances) of the composite part instance.

5 Computation model

5.1 Overview

The computation model of K is based on an object-oriented paradigm (Stefik and Bobrow 1986) and nested transactions (Moss 1981) to model the behavior of the combined execution of methods and triggered rules in an object-oriented framework. Transactions can be nested to an arbitrary number of levels by explicitly defining new transactions using the *begin_trans* and *end_trans* statements. As a result, a transaction may contain any number of nested transactions or subtransactions, and all are organized as a transaction tree whose root is the top-level transaction.

Changes to the knowledge base made by a nested transaction are contingent upon the successful commitment of all of its ancestral transactions. Aborting any of its ancestors invalidates all of its changes. If a nested transaction aborts, the knowledge-base state seen by its parent is the same as it was immediately prior to starting the nested transaction. K provides the user with the *abort* statement to *undo* any update to the knowledge base (i.e., the states of persistent entity class instances) made between the beginning of the current transaction and the abort statement. Note that the abort statement has no effect on the update to the value of any local variable itself. When a transaction is committed, both persistent and transient objects are removed from the memory cache, but only persistent objects are written into secondary storage.

5.2 Rule execution model

As mentioned in Sect. 4, a knowledge-base event could be a KBMS operation (tnew, pnew, insert, delete, and destroy), an update, or a method invocation. The occurrence of a

knowledge-base event P in transaction T on instance this of class C consists of the following steps: (1) get, bind, and trigger all the applicable before rules of P, (2) execute the event P itself, (3) get, bind, and trigger all the applicable immediate_after rules of P, and (4) get, bind, but delay the triggering of all the applicable after rules until the end of transaction T. Note that the execution of a rule body might invoke certain events which might in turn trigger other rules. Notice that both the event P and the triggering of the corresponding rules are considered part of transaction T.

After compilation, each triggering event is represented as either <iid, SourceClass, operation> for both KBMS operations and method invocations, or <iid, SourceClass, *update*, operand> for updates, where iid is the instance to which the event occurs, and SourceClass could be (1) the class, one of whose instances is being created, deleted, destroyed, or inserted, (2) the defining class of the method which is being invoked, or (3) the defining class of the attribute which is being updated (the operand followed by the *update* keyword). Note that in the case of inherited attributes or methods, SourceClass will be a superclass of the class to which iid belongs. Name conflict in multiple inheritance and subclass-redefined attribute/method will be explicitly resolved by the user in the event specification as mentioned in Sect. 4.2. Similarly, the event specifications of rules are internally represented as <SourceClass, operation> or <SourceClass, *update*, operand>. In general, *applicable rules* of event P with certain coupling mode must satisfy the following conditions. Firstly, the coupling mode and knowledge-base event must match one of the trigger conditions of this rule. Secondly, there must be an instance with the same oid as iid in the defining class of this rule. In other words, for any triggering event <iid, SourceClass, operation> or <iid, SourceClass, *update*, operand>, we match the triggering event with all the active rules of SourceClass and any of its subclasses which has an instance with the same oid as iid.

The advantage of this approach is threefold. Firstly, the search space is reduced in the sense that we start from SourceClass instead of the root class to avoid searching those inherited rules which are impossible to match the triggering event. For example, the event <BasePArt, update, cost> will never trigger any rule defined by class Part because cost is defined by BasePart and not visible from Part. Secondly, redefined attributes or methods can be easily identified and thus avoid incorrectly triggering rules. For example, if class CompositePart redefines the method deletePart of Part, then invoking deletePart in a CompositePart instance will not trigger any rule of Part which has <Part,deletePart> as its trigger condition because the SourceClasses are different. Thirdly, different from existing rule-based systems such as ODE (Agrawal and Gehani 1989) and HiPAC (Chakravarthy 1989), the choice of applicable rules in our paradigm is not limited to inherited rules. For example, suppose class Part defines an attribute cost with the constraint that the cost of a part must be greater than $10.00. As a subclass of Part, class CompositePart inherits the cost attribute and defines a more restrictive constraint that the cost value of a CompositePart must be greater than $20.00. In other words, in the case that a part is also a CompositePart, the CompositePart constraint overwrites the Part constraint. Since cost is defined by class Part, both rules have <Part,update,cost> as the event specification. Then, the update of the cost value of a Part instance must trigger *both* rules of Part and CompositePart as long as this part is also a CompositePart. Failing to trigger the CompositePart rule may leave the knowledge base in an inconsistent state. None of the existing systems addresses this problem adequately.

In the case that an update is made to an attribute of some entity class instance whose underlying domain is a complex domain class, such an update will be subjected to the constraints of the rules specified in all the enclosed domain classes as well as the hosting entity class via the navigation path. For example, suppose class Part defines a value attribute called purchase_date whose type is a complex domain class Date. The class Date defines three value attributes, month, day, and year, all of which have type integer. Then, the update event p.purchase_date.year := 1992 should trigger not only any update date::year rule defined by Date, but also any update Part:purchase_date rule defined by Part.

6 System architecture and implementation

6.1 Overview

A prototype of the K compiler has been implemented on Sun4 using C++. After bootstrapping, some of the system components were implemented in the K language itself. All the linguistic facilities described in Sects. 3–5 have been implemented. The only limitation is that in our current implementation, we treat the execution of each K program as a single transaction because the nested transaction model is not well supported in the Storage Manager level. We are currently extending the underlying KBMS to support nested transactions. The specialized tools Lex and Yacc were used to generate the lexical analyzer and parser.

The K compiler maps K code to C++ code with calls to KBMS functions. Each K class is mapped to a C++ class. As C++ does not support the distributed view of objects as described in Sect. 2, we cannot use the C++ class inheritance facilities (denoted by the keyword public) to represent a class inheritance lattice in K. Therefore, all association links, including generalization links, are mapped to C++ data members that contain, for each object, the *references* (or oids) to the associated objects.

Method declarations are directly mapped into C++ with the same encapsulation level. Similarly, operator declarations are mapped into C++ methods with special names (e.g., _KOPGT for the > operator). Each K program definition is defined as a C++ class with a main method. For every K method, two additional C++ methods are generated: (1) _KBEGIN_<method_name>, which contains calls to the Rule Processor and is invoked at the beginning of the method to trigger the applicable before rules, and (2) _KEND_<method_name>, which also contains calls to the Rule Processor and is invoked at the end of the method to trigger the applicable immediate_after and after rules.

Rules are mapped to C++ methods with special names (e.g., _KRULE_valid_salary for the rule valid_salary). Each C++ method corresponding to a rule contains the C++ code that belongs to the rule body. A *member-function pointer* is

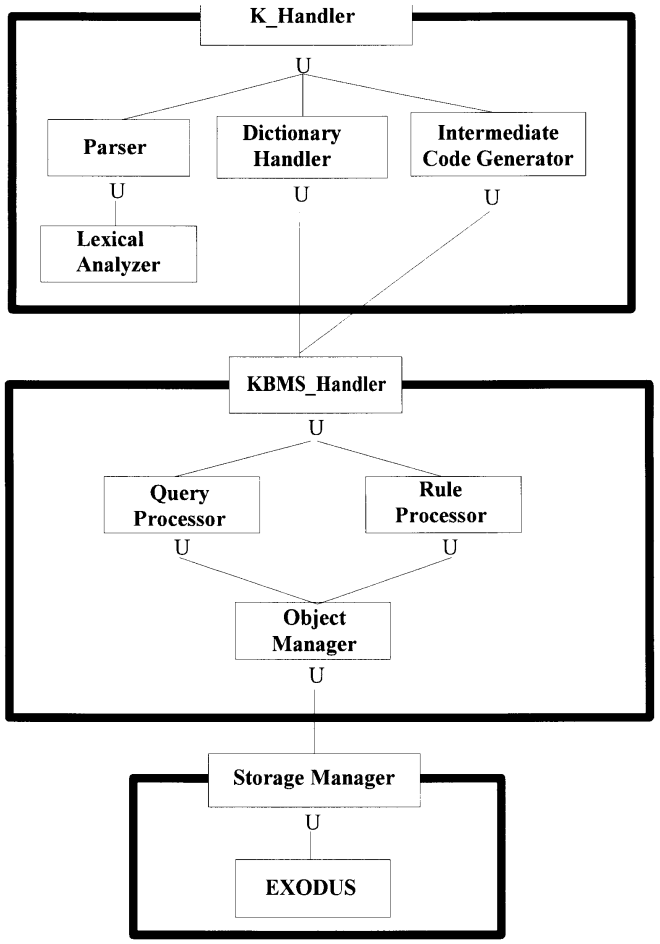

Fig. 5. The system architecture of K

defined for every method that corresponds to a rule. These pointers are used by the Rule Processor to trigger rules. For delayed (after) rules, the Rule Processor maintains a queue of rule pointers which is processed at the end of every transaction.

Basic control structures (block, if-then-else, for-loop, while-loop, break, continue, and return) are mapped into their C++ counterpart statements. Local variable declarations are mapped into C++ variable declarations.

Most of the C++ code generated by K consists of calls to the KBMS library functions. The K compiler generates an executable file which is linked with a library of KBMS functions. In the next section, we will present the components of the KBMS library.

6.2 System architecture

The implementation of K and its supporting KBMS is based on an open, modular, and extensible architecture as shown in Fig. 5. Note that each major component of the system is represented by an object class, and we use the Using (U) association to represent the client-server relationship among these components. The StorageManager is currently supported by Exodus (Richardson and Carey 1987), an object-oriented DBMS, to provide the low-level storage management (e.g., access method, data organization, and file management) and transaction management (concurrency control and recovery) facilities to the KBMS_Handler. The functionalities of the current implementation of K_Handler and KBMS_Handler are described below

The K_Handler is responsible for compiling any K program. It serves as the main interface to the K compiler. It uses the Parser module to parse an input K specification (which is tokenized by the Lexical Analyzer) and generate a standard tree representation which will be used by: (1) the Semantic Checker module, to check the semantic correctness of a K specification, (2) the Intermediate Code Generator, to generate corresponding C++ code, and (3) the Dictionary Handler module, which creates the new classes, associations, rules and methods defined in a K program as objects in the knowledge base.

As the top-level interface class of all the KBMS components, KBMS_Handler hides all the details of its constituent classes and serves as the controller of all incoming messages by dispatching them to appropriate KBMS components which actually implement the corresponding methods. KBMS_Handler uses the following classes: (1) *Object Manager*, which performs the basic KBMS operations such as tnew, pnew, insert, delete, and destroy, described in Sect. 2.4, as well as transaction management, (2) *Query Processor*, which takes as an input a query tree and evaluates a query returning the corresponding contexts as tables of extensional association patterns, as described in Sect. 3.2, and (3) *Rule Processor*, which performs the triggering of applicable rules, as described in Sect. 4.2.

At the lower level, the Storage Manager provides an interface to the underlying storage manager, which gives basic get/put semantics, indexing, and transaction management functions. The interface has been designed to provide transparency to the underlying storage manager interface functions. This allows to easily replace the underlying storage manager without affecting the upper layers (i.e., the Object Manager).

7 Conclusion

In this paper, we have described the features and implementation of the object-oriented knowledge-base programming language K. K serves as a high-level interface of the OSAM*.KBMS knowledge-base management system to define, query, and manipulate the knowledge base as well as to write codes to implement any data/knowledge-intensive application system. Starting from a query language and rule language, K seamlessly incorporates the query processing, rule processing, persistence, and general computation facilities within an object-oriented framework. K provides (1) knowledge abstraction mechanisms for supporting the underlying OSAM* knowledge model which captures any application domain knowledge in terms of the structural associations, methods, and knowledge rules, (2) a strong notion of address-independent oids, (3) a persistence mechanism for supporting both persistent and transient objects without the dangling references problem, (4) a flexible type system which supports both static type checking and multiple views of objects, (5) a declarative knowledge-retrieval mechanism based on object association patterns, and (6) an

extended computation paradigm for supporting both procedural and rule-based computations. Two versions of K and its supporting OSAM*.KBMS have been implemented at the Database Systems Research and Development Center of the University of Florida. We are currently extending the language with query optimization, generic rules, abstract class for supporting dynamic binding, and model extensibility. Another effort is to integrate K with the graphic user interface of OSAM*.KBMS (Lam et al. 1992) toward a complete KBMS-supported software development system.

Acknowledgements. This research is supported by the National Science Foundation under grant CCR-9200756. The research and development effort on the KBMS technology was supported by the National Science Foundation under grant DMC-8814989.

References

Agrawal R, Gehani N (1989) ODE (Object Database and Environment): the language and the data model. In: Proc ACM SIGMOD Int Conf Manage Data, Portland, OR, pp 36–45

Alashqur A, Su S, Lam H (1989) OQL–A Query Language for Manipulating Object-oriented Databases. Proc 15th Int Conf Very Large Databases, Amsterdam, Netherlands, August, pp 433–442

Alasqur AM, Su SYW, Lam, H (1990) A Rule-based Language for Deductive Object- Oriented Databases, Proc. of the sixth Int Conf on data Engineering, Los Angeles Calif, 5–9 February

Albano A, Cardelli L, Orsini R (1985) Galileo: A strongly-typed, interactive conceptual language ACM Trans Database Syst 10:230–260

Andrews T, Harris C (1987) Combining language and database advances in an object- oriented development environment. In: Proc 2nd Int Conf on OOPSLA, October, pp 430–440

Annevelink J (1991) Database programming languages: a functional approach. In: ACM SIGMOD Int Conf Manage Data, pp 318–327

Arroyo J (1991) The design and implementation of K.1: a third-generation-database base programming language, (Technical Report), Database Systems R&D Center, University of Florida

Atkinson MP, Bailey PJ, Chisholm KJ, Cockshott PW, Morrison R (1983) An approach to persistent programming. Comput J 26:360–365

Atkinson MP, Buneman PO (1987) Types and persistence in database programming languages. ACM Comput Surv 19:105–190

Atkinson M, Bancilhon F, DeWitt D, Dittrich K, Maier D, Zdonik S (1990) The object-oriented database system manifesto. In: Kim W, Nicolas JM, Nishio S (eds). Deductive and object-oriented databases. Elsevier (North-Holland), Amsterdam, pp 223–240

Blakeley JA, Thompson CW, Alasqur AM (1990) OQL[X]: extending a programming language X with a query capability, (Technical Report 90-07-01) Information Technologies Laboratory, Texas Instruments, Dallas, TX

Bloom T, Zdonik SB (1987) Issues in the design of object-oriented database programming languages. In: Proc 2nd Int Conf on OOPSLA, Orlando, FL, pp 441–451

Butterworth P, Otis A, Stein J (1991) The GemStone object database management system. In: CACM 34:64–77

Cacace F, Ceri S, Crespi-Reghizzi S, Tanca L, Zicari R (1990) Integrating object-oriented data modeling with a rule-based programming paradigm. In Proc ACM SIGMOD 1990, pp 225–236

Chakravarthy US (1989) Rule management and evaluation: an active DBMS perspective. SIGMOD Rec 18:20–28

Chen PP (1976) The entity-relationship model - toward a unified view of data. ACM Trans Database Syst 1:9–36

Chimenti D, Gamboa R, Krishnamurthy R, Naqvi SA, Tsur S, Zaniolo C (1990) The LDL system prototype. IEEE J Data Knowl Eng 2:76–90

Committee for Advanced DBMS Function (1990) Third-Generation database system manifesto. In: SIGMOD Rec 19:31–44

Copeland GP, Maier D (1984) Making Smalltalk a Database System, Proc. 1988 ACM SIGMOD Int Conf Manage Data, Boston, MA, pp 316–325

Dayal U, Blaustein BT, Buchmann AP, Chakravarthy US, Hsu M, Ledin R, McCarthy DR, Rosenthal A, Sarin SK, Carey MJ, Livny M, Jauhari R (1988) The HiPAC project: combining active databases and timing constraints. In: SIGMOD Rec 17:51–70

Deux O (1991) The O2 system. CACM 34:34–48

Fishman DH, Beech D, Cate HP, Chow EC, Connors T, Davis JW, Derrett N, Hoch CG, Kent W, Lyngbaek P, Mahbod B, Neimat MA, Ryan TA, Shan MC (1987) IRIS: An object-oriented database management systems. ACM Trans Off Inf Syst 5

Gehani NH, Jagadish HV (1991) Ode as an active database: constraints and triggers. In: Proc. 17th Int Conf on Very Large Data Bases, Barcelona, Catalonia, Spain, September 3–6, pp 327–336

Guo MS, Su SYW, Lam H (1991) An association algebra for processing object-oriented databases. In: Proc 7th IEEE Int Conf Data Eng, Kobe, Japan

Hammer M, Berkowitz B (1980) DIAL: A programming language for data intensive applications. In: Proc ACM SIGMOD Conf Manage Data, Santa Monica, CA, pp 75–92

Hanson E (1989) An initial report on the design of Ariel: a DBMS with an integrated production rule system. SIGMOD Rec 18:12–19

Khoshfian S, Copeland G (1986) Object identity. In: Proc ACM OOPSLA, Portland, OR, November, pp 406–414

Kim W, Ballou N, Banerjee J, Chou HT, Garza JF, Woelk D (1988) Integrating an object-oriented programming system with a database system. Proc 3rd Int Conf OOPSLA, September, pp 142–152

Lam H, Su S, Alashqur A (1989) Integrating the concepts and techniques of semantic modeling and the object-oriented paradigm. Proc 13th Int Comput Software Appl Conf (COMPSAC), October, pp 209–217

Lam H, Su SYW (1989) Prototype implementation of an object-oriented knowledge base management system (extended abstract). In: Proc 2nd Florida Conf Prod Comput Integrated Eng Manuf November, Florida, pp 68–70

Lam H, Su SYW, Ruhela V, Pant S, Ju SM, Sharma M, Prasad N (1992) GTOOLS: an active GUI toolset for an object-oriented KBMS. Int J Comput Syst Sci Eng 7:69–85

Lamb C, Landis G, Orenstein J, Weinreb D (1992) The ObjectStore database system. CACM 34:50–63

Lingat J, Rolland C (1988) Rapid application prototyping: the proquel language. In: Proc 14th VLDB Conf, Los Angeles, Calif, USA, pp 206–217

Lohman GM, Lindsay B, Pirahesh H, Schiefer KB (1991) Extensions to Starburst: objects, types, functions, and rules. CACM 34:94–109

Maier D (1989) Why database languages are a bad idea. In: Bancilhon F, Buneman P (eds) Workshop Database Program Lang, Addison-Wesley

Maier D, Stein J, Otis A, Purdy A (1986) Development of an object-oriented DBMS. OOPSLA ’86 Proc, Portland, OR, November, pp 472–482

Moss J (1981) Nested transactions: an approach to reliable distributed computing. MIT Laboratory for Computer Science, MIT/LCS/TR-260, Cambridge, Mass

Mylopoulos J, Bernstein PA, Wong HKT (1980) A language facility for designing database-intensive applications. ACM Trans Database Syst 5:185–207

Ontologic Inc (1991) ONTOS 2.0 product description. Burlington, Mass

Phipps G, Derr M (1991) Glue-Nail: a deductive database system. In: ACM SIGMOD Int Conf Manage Data, pp 308–317

Richardson J, Carey M (1987) Programming constructs for database system implementation in EXODUS. Proc 1987 ACM SIGMOD Int Conf Manage Data, pp 208–219

Richardson J, Schwartz P (1991) Aspects: extending objects to support multiple, independent roles. In: ACM SIGMOD Int Conf Manage Data, pp 298–307

Rowe LA, Shoens KA (1979) Database abstractions, views, and updates in RIGEL. In: Proc ACM SIGMOD Conf Manage Data, Boston, Mass., USA, pp 71–81

Schaffert C, Cooper T, Bullis B, Killian M, Wilpolt C (1986) An introduction to Trellis/Owl. In: Proc 3rd Int Conf on OOPSLA, Portland, OR, November, pp 9–16

Schmidt JW (1977) Some high level language constructs for data type relation. ACM Trans Database Syst 2:247–281

Shipman DW (1981) The functional data model and the data language DAPLEX. ACM Trans Database Syst 6:140–173

Shyy YM (1992) K: an object-oriented knowledge base programming language for software development and prototyping. PhD dissertation, Computer and Information Science Department, University of Florida, Gainesville

Shyy YM, Su SYW (1991) K: a high-level knowledge base programming language for advanced database applications. In: ACM SIGMOD Int Conf Manage Data, pp 338–347

Smith J, Smith C (1977) Database abstractions: aggregation and generalization. ACM Trans Database Syst 2:105–133

Smith JM, Fox S, Landers T (1983) ADAPLEX: rational and reference manual, 2nd edn. Computer Corporation of America, Cambridge, Mass

Special issue on next-generation database systems (1991). Commun ACM 34:30–120

Stefik M, Bobrow D (1986) Object-oriented programming: themes and variations. AI Mag 6:40–64

Stein LA, Zdonik SB (1989) Clovers: the dynamic behavior of types and instances (Technical Report CS-89-42). Brown University, Providence, RI

Stonebraker M, Kemnitz G (1989) The POSTGRES next generation database management system. CACM 34:78–92

Stroustrup B (1986) The C++ programming language. Reading, Mass: Addison-Wesley

Su SYW (1983) SAM*: a semantic association model for corporate and scientific-statistical databases. J Inf Sci 29:151–199

Su SYW (1989) Extensions to the object-oriented paradigm. Proc 13th Int Comput Software Appl Conf (COMPSAC), October, Orlando, FL, pp 197–199

Su SYW, Alashqur AM (1991) A pattern-based constraint specification language for object-oriented databases. In: Proc IEEE COMPCON'91, San Francisco, Calif, 25 February–1 March

Su SYW, Lam H (1992) An object-oriented knowledge base management system for supporting advanced applications. In: Proc 4th Int Hong Kong Comput Soc Database Workshop, Hong Kong, December, pp 3–22

Su SYW, Shyy YM (1993) An object-oriented knowledge model for KBMS-supported evolutionary prototyping of software systems. In: Adam NR, Bhargava B (eds) Advanced database systems. Berlin Heidelberg New York: Springer, pp 105–125

Su SYW, Krishnamurthy V, Lam H (1989) An object-oriented semantic association model OSAM*. In: Kumara ST, Soyster AL, Kashyap RL (eds) Artificial intelligence manufacturing theory and practice. American Institute of Industrial Engineering, pp 463–494

Su SYW, Lam H, Eddula S, Arroyo J, Prasad N, Zhuang R (1993) OSAM*.KBMS: an object-oriented knowledge-base management system for supporting advanced applications. In: Proc 1993 ACM SIGMOD Int Conf Manage Data, Washington, DC, pp 540–541

Wasserman, AI, Sheretz DD, Kersten ML, Van de Riet RP, Dippe MD (1981) Revised report on the programming language PLAIN. ACM SIGPLAN Not 16:59–80

Wilkinson K, Lyngbaek P, Hassan W (1990) The Iris architecture and implementation. IEEE Trans Knowl Data Eng 2:63–75

Wu P (1993) Rule validation in object-oriented knowledge bases. PhD dissertation, Department of Electrical Engineering, University of Florida, Gainesville

Yassen R, Su SYW, Lam H (1991) An extensible kernel object management system. In: Proc ACM SIGPLAN OOPSLA'91, pp 247–263