The Active KDL Object-Oriented Database System and Its Application to Simulation Support

John A. Miller∗
Walter D. Potter∗†
Krys J. Kochut∗†
Ali A. Keskin∗‡
Ender Ucar∗‡

∗Department of Computer Science
†Artificial Intelligence Program
University of Georgia
Athens, Georgia 30602
USA

‡Now at UniSQL Inc., Austin, Texas

Phone: (404) 542-3440
Email: jam@csun1.cs.uga.edu

The Active KDL Object-Oriented Database System and Its Application to Simulation Support

Abstract

Active KDL (Knowledge/Data Language) is an object-oriented database system. It evolved from earlier work on KDL which has been ongoing since 1986 [Pott86]. The foundations of Active KDL are threefold: object-oriented programming, functional programming, and hyper-semantic data modeling. These areas strongly influenced the design of Active KDL's three sub-languages: the schema definition language (SDL), the query language (QL), and the database programming language (DBPL). Because of the capabilities and elegance of these sub-languages, Active KDL is able of supporting demanding applications (e.g., simulation, model management, CAD, and intelligent database applications such as a university data/knowledge base capable of advising students). The power and versatility of these sublanguages are concretely demonstrated by showing how they can be used to handle a complex application, namely simulation support.

1. Introduction

A renaissance in semantic data modeling [Hull87] research began in the mid-1980's with the merger of programming language, database, and artificial intelligence interests [Ditt86, Kers84]. As a result, research in object-oriented databases has become very intensive [Kim89b, Zdon90] and focuses on the integration of object-oriented programming with data modeling†. In addition, data modeling and artificial intelligence integration has received much attention [Kers87, Kers88, Mylo89]. Integrating all three areas via Active KDL (Knowledge/Data Language) is the focus of our work, and represents the primary difference between Active KDL and other object-oriented database systems.

The underlying foundation of KDL [Pott87, Pott88a] as well as Active KDL [Mill90, Mill91b, Pott91] is the hyper-semantic data model KDM (Knowledge/Data Model) [Pott86, Pott88b, Pott89]. Hyper-semantic data models provide extensive modeling capabilities that

---

† Specific project examples include [Agra89, Andr89, Cope84, Fish87, Kim89a, and Maie86].

include those of standard semantic data models and object-oriented data models [Atch89, Lee90, Pott88b, Pott89]. The scope of modeling capabilities for hyper-semantic data models includes knowledge, data, and model management [Mill90, Pott90, Tidr88, Wood90].

The KDM evolved from the Functional Data Model (FDM) [Kers76, Ship81]. This evolution was motivated by the need for knowledge management facilities to be incorporated with advanced data modeling facilities in a tightly coupled manner. Consequently, the KDM has a highly functional nature. Note that the functional paradigm is the focal point of many research efforts (e.g., Vision [Caru87], HiPAC [Daya88], O_2 [Lecl88], Syntel [Risc88], and OODAPLEX [Daya89]). Because of Active KDL's focus on data, knowledge, and model management support, it is suitable for knowledge/data intensive applications. Active KDL's functional nature allows it to handle these demanding applications concisely and elegantly. Furthermore, the high level nature of its functional sublanguages insulates users from procedural details and allows for sophisticated query optimizations (a criticism of object-oriented database systems whose languages have an imperative character [Date90, Good89, Stei88]).

The functional paradigm focuses on data values described by means of functional expressions. Expressions are constructed from function applications. This leads to routines built only from functions, and therefore free of evaluation side-effects. This approach has many advantages over the traditional imperative approach [Back78]. In addition to DAPLEX [Ship81], the design of Active KDL was influenced by functional programming languages such as Hope [Burs80], Standard ML [Harp86], and Miranda [Turn85].

The following goals underline the design decisions that guided the development of Active KDL.

1. Active KDL serves equally well as a knowledge/data modeling language, a query language, and a manipulation/application language. Specifically, it consists of a schema definition language (SDL), a query language (QL), and a database programming language (DBPL). The database programming language is a superset of the query language. The power of Active KDL comes from its DBPL which is Turing complete.
2. Active KDL supports hyper-semantic data modeling which is based on the following abstraction mechanisms: classification/instantiation, specialization/generalization, aggrega-

tion, association (or membership), knowledge (in the form of constraints and heuristics), and concurrent/temporal capabilities.

3. The language is functional at the schema design level (SDL), the query level (QL), and at the application level (DBPL). The violation of strict referential transparency resulting from combining the functional and the object-oriented paradigms is precisely identified and kept to a minimum.

4. The language is simple, yet powerful. Active KDL schema specification, queries, and application routines are compact and easy to understand and design. The simplicity of the language enhances the ability of the user to concentrate on conceptual design decisions, instead of getting too involved in the procedural (implementation) details.

5. When used as a query language, Active KDL is a closed language, that is, it is possible to use the result of a query as an argument to another query. This important closure property, which is a cornerstone of the relational model, is missing from many object-oriented database systems (see [Alas89] for a discussion of OQL which was specifically developed to overcome the closure deficiency).

6. Active KDL supports active objects, that is, objects performing some task. These facilities are designed to support high concurrency [Mill91a].

In the remainder of the paper, we focus our discussion on the three Active KDL sublanguages: the schema definition language (SDL), the query language (QL), and the database programming language (DBPL). Issues to be addressed include the type system, inheritance, the meta-data, schema specification, abstraction mechanisms, query formulation, and complex applications. Both the SDL and QL sublanguages allow new object-types (or classes) to be created, statically in the case of the SDL, and dynamically in the case of the QL. Active KDL treats the creation of object-types consistently. They are inserted into the appropriate place in a collection of *inheritance lattices*. This paper formally defines this process. In particular, we define the intensional and extensional effects of using abstraction mechanisms in data modeling and using constructs in the query language in querying the database. The recognition of the importance of precisely defining these effects, especially for object-oriented query languages, is relatively recent [Beec88, Kim89a, Shaw89, Banc90]. Finally, the capabilities of the DBPL are demon-

strated by its application to the problem of simulation support [Mill89, Mill90, Pott90, Mill91b, Mill91c, Koch91a].

2. The Active KDL Schema Definition Language

A schema is used to specify the object-types available to database users. Object-types may be built up using primitive types, collection types, or other object-types.

2.1. Primitive Types

Active KDL has four primitive data types. These are INTEGER, REAL, CHAR, and BOOLEAN. The primitive types represent sets of fixed point and floating point numbers, ASCII characters, and the two logical values, respectively. The primitive types are used to form other, non-primitive types.

2.2. Object-Types

Object-types (or classes) form the foundation of Active KDL. They are the main building block of a database schema specification. An Active KDL database consists of a collection of objects. As prescribed by the classification abstraction, similar objects are grouped into object-types (or classes). Instances of an object-type (or class) are called objects. In general, an object (as a value) is an entity composed of other values whose types may be different. The syntax of an object-type definition (object-type ::=) is shown in Figure 1. Each object-type specification begins with the definition of the name of a class in the OBJECT_TYPE clause.

The optional characteristics of an object-type correspond to the KDM modeling primitives while the class-name uniquely identifies the object-type. Any object-type may be defined as a specialized (derived) form of one or more other object-types, called *supertypes*. We distinguish single and multiple inheritance, in case one or more supertypes were provided, respectively.

2.2.1. Functions

The functional approach is present in all aspects of Active KDL. An object-type (or class) may be viewed as an encapsulation of functions. In Active KDL there are four flavors of functions:

```
object-type ::=
    OBJECT_TYPE class-name HAS
        [ SUPERTYPES:          // Generalization
            class-name { , class-name };]
        [ SUBTYPES:            // Specialization
            class-name { , class-name } [ HIDING function-list ];]
        [ ATTRIBUTES:          // Aggregation
            { attribute-name: type-name
                [ WITH CONSTRAINT: constraint ];}]
        [ MEMBERS:             // Membership (Association)
            { member-name: [ SET OF | LIST OF ] class-name
                [ INVERSE OF member-name [(class-name)]]
                [ WITH CONSTRAINT: constraint ];}]
        [ CONSTRAINTS:         // Knowledge to enforce integrity
            { constraint; }]
        [ HEURISTICS:          // Knowledge to derive/infer information
            { rule; }]
        [ METHODS:  // Specifications of computations and behavior
            { method; }]
    END class-name;
```

Figure 1: Syntax of Object-Type Definition.

i) attributes, which are stored functions (if the attribute refers to an independently existing object(s) it is called a member and identified as such),

ii) constraints, which are Boolean functions,

iii) heuristics, which are functions or rules expressed using the query language, and

iv) methods, which are quasi-functions (may have limited side-effects) expressed using the database programming language.

Information about or properties of an object are stored in its attributes. Formally, a single-valued attribute maps an object to either a value of primitive type or to another object. Active KDL also supports multi-valued attributes which map an object into a set or list. An object-type (or class) has zero or more attributes that are defined in the ATTRIBUTES clause. Each attribute must have a defined type.

In Active KDL, constraints may be imposed in very general ways. Simple constraints may be attached to an attribute to ensure that it takes on only appropriate values. Complex constraints may be used to ensure that new objects inserted into an object-type are compatible with current objects in this or any referenced type. In general, a constraint in Active KDL is any Boolean function that is constructible using the query language. Constraints may be called (just

like an ordinary function), and will return a Boolean result indicating whether or not the constraint is currently satisfied. Implementation options will allow for various types of constraint enforcement to be carried out (e.g., update-driven, query-driven, and, in certain circumstances, user-initiated).

Heuristics allow information about an object to be inferred, rather than retrieved using a traditional query against stored data. Thus, heuristics provide an information derivation mechanism that results in greater informational content than is present in the stored data alone. A particular heuristic for an object-type is manifested as a rule written in a declarative manner following the query language syntax. Because the rules are expressed declaratively (functionally), their evaluation is not fixed as it would be in an imperative language, but rather is determined by our query evaluation/optimization algorithms, which are currently under development [Kesk90a]. Heuristics give Active KDL fundamentally more expressive power than a relationally complete language like SQL.

Formally, a heuristic is a function that is expressed as a parameterized query. The parameterization is in the form of passing an object of the class as a parameter to the heuristic. Note that heuristics are overloaded so that they can also apply to any subset of objects from the class. In this case, meta-rules may be specified to control and manage further results (e.g., providing an abstract query response or an inference explanation). Methods which are more general than heuristics, will be discussed in the section on the database programming language.

2.3. Collection Types

Active KDL supports two predefined collection types: SET OF and LIST OF. The predefined Active KDL functions and operators (defined later in this paper) work on operands of these types.

1. A *set type* allows for the creation of subsets of values from some given type T. Specifically, the value of a set of type SET OF T is a collection of values from types *conforming* to T. [A formal definition of the meaning of type S conforming to type T will be given in the next section.] The element type T on which a set is defined may be any primitive type or object-type. For example, declaring an attribute to be of type SET OF INTEGER allows the attribute to return as its value a subset of INTEGERs. In Active

KDL, anything that is set-valued (i.e., an object-type or a set type) may not contain duplicates.

2. A *list type* is similar to a set type except that elements are ordered and duplicate values are permitted to exist in the list. As with sets, a list is a collection of values of *conforming* types. A STRING object is simply a list whose elements are of type CHAR. That is, STRING is a system defined alias for LIST OF CHAR.

2.4. Inheritance in Active KDL

In general, a set of object-types (classes) declared for a schema specification may form an inheritance hierarchy. Since object-types with multiple supertypes are allowed, the hierarchy is generalized into a lattice. Hence, Active KDL supports multiple inheritance. In the database and programming language literature, the definitions and restrictions on the use of inheritance vary considerably. Most systems use it for specialization, while some also use it for aggregation. Some languages like C++ are quite liberal with inheritance, allowing it for the purpose of code reusability in case of partial inheritance [Nier89]. To make our discussion of inheritance in Active KDL more precise, it is imperative to isolate two aspects of inheritance: the intensional aspect and the extensional aspect.

The intensional state of the database specifies the type structure for all the types (or classes) defined for the database. This state is modified both by schema evolution (e.g., adding a new subtype of an existing object-type) and by query processing when results are saved (e.g., such as when retrieving and saving information on student workers from the student object-type and the employee object-type). The extensional state then is the set of objects (instances) that currently exist in the database. Note, because of our use of lazy evaluation†, some of the objects that exist may not actually be stored, but rather are manifested only upon request.

In Active KDL, inheritance is coupled with the specialization/generalization abstraction mechanism. From an extensional point of view, types are related to each other. The relationship between types is that of a supertype/subtype (`is-a`) relationship. Consequently, the extension

---

† In lazy evaluation, an expression is not evaluated until absolutely needed. This allows representation of infinite (or very large) data structures and construction of flexible expressions with the use of such infinite data structures.

of a (sub)type is a *subset* of the extension of each of its supertypes. This can be captured as a binary relation on types. Specifically, the relation that exists between types is that of a *family of join lattices*. Formally, the i^{th} join lattice

$$L_i = L(TYPES_i, \subseteq, lub)$$

is a partial order (reflexive, antisymmetric, and transitive) where every pair of types has a least upper bound (lub). The lub property may be expressed as follows:

$$lub(S, T) \in TYPES_i$$

A natural way to represent relations is with a *digraph*. In particular, it is advantageous to represent L_i with a minimal digraph. The minimal digraph D_i for which $D_i^* = L_i$ will serve as the representation. [The * operator is the reflexive and transitive closure.] The antisymmetric property of the lattice implies that the digraph must be *acyclic*, while the lub property implies that the digraph must be *rooted*. Hence the minimal digraph representation for a lattice must be a *rooted DAG*. Note, Active KDL avoids the disadvantage of enforcing the lub property once and for all using a global dummy root (e.g., Object), whereas other systems, such as Orion [Kim89b, chapter 11], do not avoid this disadvantage.

Formally, a rooted DAG, $D = (N, A)$, is a set of ordered pairs of nodes and arcs where the nodes are connected, there is one unique node (the root) with no incoming arcs, and no cycles exist in the digraph. Each node represents an object-type, while each arc represents a supertype/subtype (`is-a`) relationship. All of the functions of a supertype are inherited by its subtypes. Furthermore, any object that is an instance of a type, is a member of the extension of that type, and also a member of the extension of all its supertypes.

An object-type S *conforms* to type T if either S and T are the same, or T is an ancestor of S in some inheritance digraph, D_i. Because function redefinition in a derived object-type is permitted, late binding of function names with the functions implementing them is required.

Since Active KDL has the ability to describe itself (i.e., provide its own meta-description), the inheritance digraphs are stored in a special Active KDL OBJECT_TYPE called Digraph. Some of the object-types used for storing the Active KDL meta-data are shown in a partial listing of the meta-data schema, shown in Figure 2. The Node object-type, in particular, shows the inherent modeling control mechanism used to handle inheritance in Active KDL.

```
OBJECT_TYPE Digraph HAS                 // Inheritance Digraph
        ATTRIBUTES:
                Root: Node;
        HEURISTICS:
                Classes (d: Digraph): SET OF Node = fam (Root (d));
                NumNodes (d: Digraph): INTEGER = COUNT (Classes (d));
END Digraph;

OBJECT_TYPE Type HAS
        ATTRIBUTES:
                Name: STRING;           // Type name
END Type;

OBJECT_TYPE Primitive_Type HAS
        SUPERTYPES:
                Type;
        METHODS:
                CreateAll (): SET OF Primitive_Type;
                        // Create BOOLEAN, CHAR, INTEGER, and REAL
END Primitive_Type;

OBJECT_TYPE Collection_Type HAS
        SUPERTYPES:
                Type;
        ATTRIBUTES:
                Kind: STRING WITH CONSTRAINT:
                        OneOf (c: Collection_Type) = Kind (c) IN {"LIST", "SET"};
        MEMBERS:
                Base_Type: Type;
END Collection_Type;

OBJECT_TYPE Node HAS                            // Each Object_Type is a Node in a Digraph
        SUPERTYPES:
                Type;
        ATTRIBUTES:
                newfun: SET OF Function;        // Newly defined functions
                redfun:  SET OF Function;       // Redefined functions
        MEMBERS:
                dig:  Digraph;                  // Containing Digraph
                sup:  SET OF Node;              // Supertypes in SUPERTYPES clause
                sub:  SET OF Node;              // Subtypes in SUBTYPES clause
        CONSTRAINTS:
                FullInheritance (n: Node): BOOLEAN =
                        fun (n) >= fun (Sup (n));
                Acyclic (n: Node): BOOLEAN =
                        NOT n IN SUP (n);
                Location (n: Node): BOOLEAN =
                        n IN Classes (dig (n));
        HEURISTICS:
                Sup (n: Node): SET OF Node =                    // Immediate supertypes
                        sup (n) +
                        FOR ALL m IN Classes (dig (n)) WHERE n IN sub (m) APPLY m END;
                Sub (n: Node): SET OF Node =                    // Immediate subtypes
                        sub (n) +
                        FOR ALL m IN Classes (dig (n)) WHERE n IN sup (m) APPLY m END;
                SUP (n: Node): SET OF Node =                    // All supertypes
                        Sup (n) + SUP (Sup (n));
                SUB (n: Node): SET OF Node =                    // All subtypes
                        Sub (n) + SUB (Sub (n));
                fam (n: Node): SET OF Node =                    // Sub-digraph family
                        SUB (n) + n;
                top (n: Node): Node =                           // Root above node n
                        Root (dig(n));
                fun (n: Node): SET OF Function =                // All functions
                        newfun (n) + fun (Sup (n));
        METHODS:
                lub (n: Node; m: Node): Node;   // Least Upper Bound (must exist)
                glb (n: Node; m: Node): Node;   // Greatest Lower Bound (may exist)
END Node;
```

```
OBJECT_TYPE Function HAS
        ATTRIBUTES:
                Name:    STRING;
        MEMBERS:
                ResType: Type;
END Function;
```

Figure 2: Important Meta-Data Object-Types.

In Active KDL, a type represents an *intensional* aspect of the database. The intensional information for types defined in a schema is stored in the meta-data. Additionally, intensional information about queries is also generated, and may be stored in the meta-data if the result of the query is saved.

The function fam(T) returns the transitive closure of all subtypes for T, including the type T itself. It represents a sub-digraph of nodes that are rooted at the node representing T in the inheritance lattice. In particular, if T does not have any subtypes, fam(T) is a singleton set containing only T.

Besides the functions specified in the meta-data, we need to define two additional functions. One that maps intensional information into extensional information, and a second that does just the opposite. The set of values (objects) for a given type T, denoted $val(T)$, represents an *extensional aspect* of the database. In particular, for a terminal object-type T (i.e., T has no subtypes),

$$val(T) = \{\ obj\ \mid type\text{-}of\ (obj) = T\ \}$$

The function *type-of* returns the most specific type of *any* extensional element. In particular, the element may be a number, a list of characters, or an object instance. More generally, if T is not a terminal object-type (i.e., it has additional object-types derived from it) then

$$val(T) = \bigcup_{S \in fam(T)} val(S)$$

2.5. Support for Abstraction Mechanisms

The data or schema definition language (SDL) is used for data modeling. During initial schema design or later schema evolution, designers are able to use the powerful conceptualizations provided by Active KDL to realistically model objects in the domain of interest. This section focuses on how new object-types can be defined from existing object-types. In particular, new types can be created using any one of four abstraction mechanisms: specialization,

generalization, association and aggregation. As discussed earlier, only the former two abstraction mechanisms are coupled with inheritance. The latter two are manifested by attribute nesting, and are independent of the (is-a) inheritance lattices.

2.5.1. Specialization

Subtypes (or subclasses) of an existing type (referred to as the supertype) can be defined at any time. The object instances in the subtype are also instances of the supertype. Consequently, anything that can be done to an object in the supertype can also be done to an object in the subtype. The subtype inherits all of the functions available to the supertype. Furthermore, since an object in the subtype is more particularized or specialized, it is only logical that additional functions can be applied to it. In Active KDL, specialized types are created from existing types by adding one or more functions (i.e., attributes, constraints, heuristics, or methods).

OBJECT_TYPE *SpecialType* HAS
    SUPERTYPES: *Type*;
    ...
END *SpecialType*;

Formally, functions are added and a subset relationship exists between the two types.

$$fun(SpecialType) = fun(Type) \cup \{ f_1, \cdots, f_n \}$$
$$val(SpecialType) \subseteq val(Type)$$

2.5.1.1. Multiple Specialization

Multiple inheritance allows a class to inherit functions from multiple superclasses. If types S and T currently exist and are in the same lattice, then a new joint type can be created that combines the two.

OBJECT_TYPE *JointType* HAS
    SUPERTYPES: *S*, *T*;
    ...
END *JointType*;

The functions defined on *JointType* are simply the union of those defined for the supertypes and those added by the type itself. If functions have the same signature, then the first one encountered, by following the ordering given in the SUPERTYPES clause, is the one taken. This gives the user some control over managing conflicts.

$$fun(JointType) = fun(S) \cup fun(T) \cup \{ f_1, \cdots, f_n \}$$

From an extensional point of view the situation is also straightforward. Since *JointType* `is-a`

S and *JointType* `is-a` T, objects that are instances of *JointType* must also be instances of S and instances of T. If $dig(S) = dig(T)$, then the two types have some fundamental compatibility, since they belong to the same lattice. Therefore, their combination is legal and, extensionally speaking, results in the following:

$$val(JointType) \subseteq val(S) \bigcap val(T)$$

For example, if the classes S = Student and T = Employee are in the same lattice (e.g., $top(S) = top(T)$ = Person), then a new type called StudentEmp can be defined. An object that is in *val*(StudentEmp), must also be in *val*(Student) and in *val*(Employee). However, the situation is different if S and T are from separate lattices. This naturally occurs when we try to define a composite or aggregate object. In such cases, the supertypes may be fundamentally different. For example, let S = Engine, T = Body, and then define Car to be composed of an Engine and a Body. Conceptually speaking, the relationships between these object-types are better described with `part-of` relationships rather than with `is-a` relationships. [In some systems (including Active KDL), `part-of` relationships are coupled with (aggregation) nesting through, for example, a class composition hierarchy [Kim89b].] In Active KDL, `part-of` relationships are manifested through attributes (see below).

2.5.2. Generalization

Sometimes it is advantageous to create a class that generalizes one or more existing classes. Generalized types are produced by hiding functions of an existing type. This is the reverse of what specialization does.

OBJECT_TYPE *GeneralType* HAS
    SUBTYPES: S, T HIDING $f_1, \cdots, f_n$;
END *GeneralType*;

The *fun* and *val* definitions are naturally just the reverse of what they were for specialization.

$$fun(GeneralType) = fun(S) \bigcap fun(T) - \{ f_1, \cdots, f_n \}$$
$$val(GeneralType) \supseteq val(S) \bigcup val(T)$$

2.5.3. Association

Associations or relationships between objects are fundamentally important for any semantic, hyper-semantic, or object-oriented data model. Objects without connections to other objects are typically insignificant. Memberwise attributes (or members) whose type is an object-type

enable associations between objects to be formed. For example, associations between students and courses can be easily established as shown below.

```
OBJECT_TYPE Student HAS
        ATTRIBUTES:
                SSN:        INTEGER;
                Name:       STRING;
        MEMBERS:
                Courses:    SET OF Course;          // An enrollment relationship
END Student;

OBJECT_TYPE Course HAS
        ATTRIBUTES:
                CourseNum: INTEGER;
                CourseName: STRING;
        MEMBERS:
                Students:   SET OF Student INVERSE OF Courses (Student);
END Course;
```

Figure 3: Student and Course Association.

This is an example of a many-to-many relationship (or association). The INVERSE OF clause specifies that Students and Courses are just opposite ends of the same relationship.

Active KDL directly provides for binary relationships of the many-to-many, many-to-one, and one-to-one variety. In addition, higher arity relationships may be simulated by defining an object-type for this purpose. This has been called *elevating* an attribute to an associative entity set in the FDM [Hech81]. Therefore, all forms of relationships or associations provided by popular semantic data models such as the Entity-Relationship Model (ERM) [Chen76] can be straightforwardly handled in Active KDL. Furthermore, the powerful constraint specification capabilities of Active KDL allow for a greater variety of constraints to be handled in Active KDL as opposed to, for example, the ERM.

2.5.4. Aggregation

Active KDL provides aggregation using attributes. In this sense, it looks much like association. However, association is a weak coupling between objects, not nearly so strong as what is suggested by the meaning of `part-of`. The fact that a course is deleted, will in all probability not have dire consequences on a student object that happens to be enrolled in the course. Hence associations can be established and broken at any time. Conversely, aggregation is a strong coupling. If a car is taken to a junkyard and smashed into scrap metal, thereby deleting the car, the

chances are good that the engine was smashed as well and should also be deleted. Since this engine is a proper attribute and not a member, it is a dependent object, and will be automatically deleted when the car object containing it is deleted. The example below illustrates the point.

```
OBJECT_TYPE Car HAS
        ATTRIBUTES:
                E: Engine;
                B: Body;
        METHODS:
                CreateCar(make: STRING; model: STRING; color: STRING): Car;
                        // Constructor -- create a car and its components
                DestroyCar(Car): BOOLEAN;
                        // Destructor -- destroy the car and its components
END Car;
```

Figure 4: Engine and Body Aggregation.

3. The Active KDL Query Language

Active KDL, in addition to being a data modeling language, is also a query language (QL). In this section, we build up the query language starting with simple queries. We discuss both the intensional and extensional aspects of the queries.

3.1. Simple Queries

A *simple query* is the building block of all queries. Depending on the desired result, a simple query may be a class retrieval query or a function application query.

3.1.1. Class Retrieval Queries

In case we need all current instances of type *T*, we may state:

Query: *T*

When used as a query, the name of some defined object-type T results in the set which is the union of all the sets of object values belonging to the object-type and also to its subtypes. Therefore, the extension of T may contain objects of heterogeneous (but related) types. T as a query represents the intension. The intension of T is T itself. The extension of the query T is defined as $val(T)$. Formally,

$$Intension(Query) = T$$
$$Extension(Query) = val(T)$$

3.1.2. Function Application Queries

In case we need all functional values of some *function* of all instances of type T, we write:

Query: $f(T)$

The intension of the query is defined as a *generalization* of type T called f_T. In fact, f_T is a new object-type where f is the only function. The extension of this query is a set of objects with function values of f equal to that of the corresponding f values of some object in T. Formally,

$$Intension(Query) = f\_T$$
$$Extension(Query) = \Pi_{f\_T}\, val(T)$$

Here, the concept of generalization, as discussed by [Kim89a], is closely related to *projection* in *relational algebra*. For every object in the extension of f_T, the function $type\text{-}of$ returns f_T or possibly one of its subtypes. If the argument of f is a single object rather than a collection of objects (i.e., a class), then its corresponding function value is returned. This form is restricted to FOR-query and heuristic specifications only (see below).

3.2. General Queries

The use of the FOR construct in conjunction with set operators allows very general and powerful queries to be formulated.

3.2.1. Projection Queries

Projection queries are intended to extract some subset of attribute values from instances of a given object-type.

Query: FOR ALL t IN T APPLY $f_1(t)$, ..., $f_n(t)$ END

$$Intension(Query) = f_1 \cdots f_n\_T$$
$$Extension(Query) = \Pi_{f_1 \cdots f_n\_T}\, val(T)$$

The new object-type $f_1 \cdots f_n\_T$ contains n functions, i.e. $f_1, \cdots, f_n$. The extension of this query is a set of objects with function values of $f_1, \cdots, f_n$ equal to that of the corresponding values of some object in T. The type $f_1 \cdots f_n\_T$ is viewed as a generalization of T. Note, the query

FOR ALL t IN T APPLY $f(t)$ END

is equivalent to the query $f(T)$, both in extensional and intensional aspects.

3.2.2. Selection Queries

Selection queries are intended to extract object instances of type T satisfying some additional constraint p.

Query: FOR ALL t IN T WHERE $p(t)$ APPLY t END

$$Intension(Query) = p\_T$$
$$Extension(Query) = \{ t \mid t \in val(T) \text{ and } p(t) \}$$

Here, the constraints defined on the object-type T are extended (using a logical *and*) to include the condition $p(t)$. The type p_T is a specialization of T (specialized via $p(t)$) since every object that is an instance of p_T will be a member of the extension of T. Note that finding the correct placement for this new query generated object-type requires constraint analysis [Urba90] to address the predicate subsumption question. Since this is a difficult problem (it is decidable since predicates may not use methods [Chan73, Kris88]), it will be done in the background and only if the results of the query are to be stored.

3.2.3. Set Theoretic Queries

Set theoretic queries may be formed using the typical set theoretic operators. Binary operators are applicable only to sets of objects of compatible types (i.e., the types must belong to the same lattice). For example, the *union* of two compatible types is expressed as follows:

Query: $S + T$

$$Intension(Query) = lub(S, T)$$
$$Extension(Query) = val(S) \cup val(T)$$

Note that the least upper bound of S and T, $lub(S, T)$, must exist since they are both in the same lattice. The *set difference* of two compatible types is expressed as follows:

Query: $S - T$

$$Intension(Query) = S$$
$$Extension(Query) = val(S) - val(T)$$

Finally, the *intersection* of two compatible types is expressed as follows:

Query: $S * T$

$$Intension(Query) = glb(S, T)$$
$$Extension(Query) = val(S) \cap val(T)$$

If the greatest lower bound of S and T, glb(S, T), does not exist and the intersection of S and T is nonempty, then the glb will be created if the result of the query is saved.

3.2.4. Join Queries

Join queries may be used to produce composite objects formed from all possible pairs (Cartesian product) of objects in types S and T. Just as with aggregation used for deriving new object-types in a schema, join queries build composite objects by nesting.

Query: FOR ALL s IN S, t IN T APPLY s, t END

$$Intension(Query) = S\_T$$
$$Extension(Query) = val(S) \times val(T)$$

The new type S_T is simply formed as follows:

```
OBJECT_TYPE S_T HAS
        ATTRIBUTES: S_: S; T_: T;
END S_T;
```

3.3. Active KDL Query Language Syntax

The Active KDL query language was designed to be simple, flexible, and easy to learn. Furthermore, it is a purely functional language. The syntax for the query language is given in Figure 5.

```
query ::=  constant-set | class-name | application | function-name (query) |
           query operator query | for-query
application ::= constant | variable | function-name (application)
for-query ::=      FOR ALL variable IN query { , variable IN query } [ WHERE predicate ]
                       APPLY application { , application }
                   END
function-name ::= built-in-fun | attribute-name | constraint-name | rule-name
operator ::= + | - | * | /
predicate ::= term { bool-op term }
bool-op ::= AND | OR
term ::= query rel-op query | NOT term | (predicate)
rel-op ::= = | <> | < | > | <= | >= | IN
constant-set ::= "{" constant .. constant "}" | "{" constant { , constant } "}"
```

Figure 5: Syntax of Query Language.

The standard built-in operators { +, -, *, / } apply to all three categories of types: primitive types, collection types, and object-types (i.e., they are overloaded). These operators share the same semantics for all set-valued operands (i.e., constant-set, class-name, or a function whose return type is SET OF any-type). For LIST OF any-type, the semantics are similar except that "+" allows for duplication. For the primitive types these operators have their natural semantics (e.g., "+" means addition for INTEGER operands). For operands of type BOOLEAN, these operators are not defined. Complex predicates can be formed from simple predicates using Boolean operators (AND, OR, and NOT).

The built-in relational operators { =, <>, <, >, <=, >=, IN } apply to object-types and to all primitive types. The first six are straightforward (e.g., < means less than for INTEGERs and proper subset for set-valued operands). The IN operator is special in that it requires either a first operand of primitive type or a singleton set, and a second operand which is set-valued. Notice that the requirement that a set be a singleton set is not always deducible at compile time, and hence may need to be checked at run time.

For the object-types, operands in expressions are compatible if they are in the same lattice. For the primitive types, we follow the same principle by implicitly defining the following lattices:

Lattice 1: CHAR `is-a` INTEGER `is-a` REAL
Lattice 2: BOOLEAN

Lattice 1 expressions will be automatically upgraded (e.g., INTEGER + REAL will give a REAL result).

4. The Active KDL Database Programming Language

To support demanding applications like simulation, a query language is not powerful enough. In the past, the remedy was to embed the query language in a general-purpose programming language such as C. The current trend is to reject this approach (see [Banc90] for reasons) in favor of incorporating a high level database programming language that is syntactically compatible with the query language (no impedance mismatch). Typically database programming languages are smaller than their general-purpose counterparts, have a higher level nature, and automatically support persistence of objects [Banc90].

General computations may be expressed in Active KDL using methods. Methods are closely related to heuristics. However, they are more general than heuristics in that they can take any number of parameters and, most importantly, allow limited forms of side-effects to occur. Hence, methods somewhat relax the strict functional paradigm followed by the rest of the features in Active KDL. In the schema, only the signature of a method must be given. The body of a method is implemented using the database programming language (DBPL). It may be written inline or separately. Methods are used to perform data manipulation, and implement applications.

The central assumption about the programming language aspect of Active KDL is that it should be functional at the application level. Since the language must manipulate persistent objects, i.e. values with mutable internal states, it was not feasible to eliminate all *side-effects*. [A side-effect occurs when the same expression (in our case a method, possibly with some arguments) returns different results when evaluated at different times.] In our opinion, it would not be reasonable to treat the whole database as a single value, where an updating function would take the whole database as one of its arguments and produce a single value -- a different (updated) database. Thus, in designing Active KDL we made a conscious effort to keep the clash between the functional and object-oriented paradigms (side-effects) as limited as possible.

Consequently, only select methods for a given object-type will affect the state and lifespan of objects.

FOR expressions are the central language construct provided by Active KDL. In principle, the FOR expression is similar to ZF notation (Zermelo-Fraenkel set notation) also used in other functional languages, such as Miranda [Turn85] and ML [Harp86].

FOR ALL *variable*$_1$ IN *query*$_1$, *variable*$_2$ IN *query*$_2$, . . . , *variable*$_n$ IN *query*$_n$
[WHERE *predicate*]
EVAL *expression*

A FOR expression may introduce a number of local variables. The scope of each variable is the FOR expression itself. Each of the variables ranges over the elements in the set returned as the value of the corresponding query. Moreover, the *expression* may contain free occurrences of any of the *variables*. Furthermore, *query*$_i$ may involve free occurrences of the preceding *variable*$_j$ $(1 \leq j < i)$.

Two additional constructs are useful in dealing with more complex logic. To enable decision making within methods, a conditional expression construct is available in Active KDL. The form of a conditional expression is as follows:

IF *predicate* THEN *expression*$_1$ ELSE *expression*$_2$

The type of *predicate* is BOOLEAN and the types of *expression*$_1$ and *expression*$_2$ must be conforming. The value of a conditional expression is *expression*$_1$ if the value of the *predicate* is TRUE, and *expression*$_2$, otherwise. Qualified expressions are used for introducing temporary names (variables). The form of a qualified expression is presented below:

LET *variable* = *expression* { ; *variable* = *expression* } IN *expression*

The value of the qualified expression is the *expression* after IN, which may contain free occurrences of the *variables*.

Update expressions are allowed only within methods and are used to modify the database. These are the only source of side-effects in Active KDL. The *create expression* is used to construct new instances of an object-type, while the *recreate expression* is used to replace existing object instances of an object-type.

CREATE *attribute = expression* { ; *attribute = expression* } END

RECREATE *attribute = expression* { ; *attribute = expression* } END

For RECREATE, all expressions are evaluated and then a new object instance with given attribute values is created, replacing the old one. Argument expressions may refer to function values of the old object instance. Both CREATE and RECREATE expressions return an object value. The type of the return value is the same as that of an enclosing object-type (CREATE and RECREATE must be used within a method, which in turn must be defined within an object-type).

Our heuristics and methods have direct analogues in POSTQUEL (the query language for POSTGRES [Ston86]). POSTQUEL supports complex attributes in the form of (i) *postquel attributes* (i.e., the attribute value in a tuple is the result of an arbitrary query) and (ii) *cproc attributes* (i.e., the attribute value is determined by executing an arbitrary C function). Note, since we follow the object-oriented paradigm, our rules and methods are attached to the object class as a whole and not to individual objects or tuples.

5. Process-Interaction Based Simulation

Active KDL allows simulation models to be developed that follow the process-interaction simulation world-view. In this world-view, processes are created, use system resources, interact with each other, and finally are destroyed. Basically, a process can be doing one of three things: changing the state of the system, performing some activity, or waiting for some condition to become true. It is only necessary for a process to have the (or a) CPU if it is changing the state of the system. Hence, it is sufficient to implement simulation processes as coroutines or more generally as threads that share a single CPU.

In Active KDL, simulation processes are known as Sim_Object's. Objects in the Sim_Object class are special types of active objects [Mill91a]. Such objects may suspend themselves on a queue, work (delay themselves) for a given amount of time, and reactivate other suspended Sim_Object's (see [Mill91b] for details). In Active KDL, active objects are implemented as threads (lightweight processes). Threads provide true concurrency on a uniprocessor and true parallelism on a multiprocessor or distributed system. In our uniprocessor C++ implementation, threads are available to us in two ways: They are provided in the *task class* [Duhw89]

which is part of the standard library that comes with the *AT&T C++ Language System, Release 2.0*. They have also been implemented by us to run under GNU C++ by using the SunOS 4.x Lightweight Processes Library [Sun90]. In a Sim_Object, concurrency comes about since invoking a class constructor causes the creation and execution of a new thread. Any class derived from the Sim_Object class will have concurrent capabilities, and is said to be an active class (consisting of active objects).

To demonstrate the simulation capabilities of Active KDL, we show an implementation of a highly simplified bank simulation. The bank consists of one teller with customers arriving according to a Poisson process, and with service times being exponentially distributed (an *M/M/*1 queue). Customers who find the teller busy will wait in line in order of arrival (FCFS).

The methods specified in the schema shown in Figure 6 are implemented separately using the database programming language (DBPL). Only the signatures of the methods are given in the schema. A method exists for each of the object-types (classes) given in the schema to serve as a constructor. A method used as a constructor creates a new instance of the object-type (i.e., a new object). The two constructors (methods named Create) for the Customer and Bank classes, are used to create active objects. Because their object-types are specializations of Sim_Object, these constructors are able to delay and suspend their evaluations. Finally, the last two methods are provided to allow the actions of a customer to modify the state of the bank they are in. [Note that to resolve ambiguity, a method name may be prefixed by a class name (e.g., Customer.Create).]

6. Query Driven Simulation Using Active KDL

A very nice feature of database systems is that they provide users with a concise and easy-to-use interface to large amounts of data. This interface is in the form of a query language (e.g., SQL). The query language for Active KDL has the high level, easy to use character of SQL. However, Active KDL is strictly more powerful than SQL since it supports recursive queries (through heuristics) and general computations (through methods). Furthermore, its object-orientation allows not only data, but knowledge and models to be accessed.

The power of the Active KDL database system allows for the development of sophisticated

```
OBJECT_TYPE Customer HAS
        SUPERTYPES:
                Sim_Object;
        ATTRIBUTES:
                Account_Num: INTEGER;
                Balance:        REAL;
                Interest_Rate:  REAL;
                Arrival_Time:   REAL;
                Start_Service:  REAL;
                Waiting_Time:   REAL;
                System_Time:    REAL;
        HEURISTICS:
                Annual_Yield (c: Customer): REAL =
                        Balance (c) * Interest_Rate (c);
        METHODS:
                Create (b: Bank_Model): Customer;
                        // Constructor -- Script for a typical customer
END Customer;


OBJECT_TYPE Bank_Model HAS
        SUPERTYPES:
                Sim_Object;
        ATTRIBUTES:
                Name:           STRING:                 // Name of bank
                Num_Customers: INTEGER;                 // Number of customers
                Mean_Arrival: REAL;                     // Mean interarrival time
                Mean_Service: REAL;                     // Mean service time
                Teller_Idle:    BOOLEAN;                // Teller Status
        MEMBERS:
                Stream:         Ran_Stream;             // Random variate stream
                Bank_Queue:  LIST OF Sim_Object;// Queue of waiting customers
                Customers:      SET OF Customer;        // Customers served by the bank
        HEURISTICS:
                Mean_Wait (b: Bank_Model): REAL =
                        AVERAGE (Waiting_Time (Customers (b)));
                Throughput (b: Bank_Model): REAL =
                        Num_Customers (b) / Time (Clock);
        METHODS:
                Create (stream: INTEGER = 1; num_Customers: INTEGER = 100;
                        mean_Arrival: REAL = 8.0; mean_Service: REAL = 7.0): Bank_Model;
                                // Constructor -- Script for a bank model
                Begin_Service (b: Bank_Model): REAL;
                End_Service (b: Bank_Model): REAL;
END Bank_Model;
```

Figure 6: Process-Interaction Based Bank Simulation.

information intensive applications. Query driven simulation (QDS) [Mill89, Mill90, Mill91b, Mill91c] is an example of such an application. Information about the structure, performance, and reliability of systems under consideration is captured by Active KDL. This information allows Active KDL to answer questions posed by users. The questions may be answered by simple data retrieval, complex query processing, querying requiring heuristic knowledge, or even model instantiation.

Model instantiation [Mill90] occurs when Active KDL does not have sufficient data or knowledge to provide a satisfactory answer. In such a case, Active KDL automatically creates model instances that are executed to generate enough data to give a satisfactory answer to the

```
Customer.Create (b: Bank): Customer [ Sim_Object.Create () ] =
        LET c = Work (Exponential (Stream (b), Mean_Service (b)),
                CREATE
                        Arrival_Time = Time (Clock);
                        Start_Service = Begin_Service (IF Teller_Idle (b)
                                THEN b
                                ELSE Suspend (Bank_Queue (b), b))
                END)
        IN
                LET end_Time = End_Service (b)
                IN
                        RECREATE
                                Waiting_Time = Start_Service (c) - Arrival_Time (c);
                                System_Time = end_Time - Arrival_Time (c)
                        END;


Bank.Create (stream: INTEGER; num_Customers: INTEGER; mean_Arrival: REAL;
                mean_Service: REAL): Bank_Model [ Sim_Object.Create () ] =
        LET b =
                CREATE
                        Name            = "Bank";
                        Num_Customers = num_Customers;
                        Mean_Arrival  = mean_Arrival;
                        Mean_Service  = mean_Service;
                        Teller_Idle     = TRUE;
                        Stream          = Ran_Stream.Create (stream)
                END
        IN
                FOR ALL i IN {1 .. Num_Customers} EVAL
                        Work (Exponential (Stream (b), Mean_Arrival (b)),
                                RECREATE Customers = Customers (b) + Customer.Create (b) END)


Begin_Service (b: Bank_Model): Bank_Model =
        LET b =
                RECREATE Teller_Idle = FALSE END
        IN
                Time (Clock);


End_Service (b: Bank_Model): REAL =
        LET b =
                RECREATE
                        Teller_Idle = TRUE;
                        Bank_Queue = IF COUNT (Bank_Queue (b)) > 0
                                THEN Reactivate (Bank_Queue (b))
                                ELSE Bank_Queue (b)
                END
        IN
                Time (Clock);
```

Figure 7: Method Implementation for Bank Simulation.

user. Depending on the complexity of the query, model instantiation may be a simple or quite complex process. The process centers around the creation of sets of input parameter values which are obtained by schema and query analysis. Model instantiation has the potential to require an enormous amount of computation in response to a query. Therefore, a user settable threshold is provided to control the amount of computation. If the threshold is at 100%, then all parameter sets implied by the query, whose results are not already stored in the database, are used to instantiate models. At the opposite extreme if the threshold is at 0%, then only data

generated from previous simulation runs will be retrieved. Intermediate values for the threshold would typically be the case.

In the rest of this section, we demonstrate successively more complex examples of query driven simulation. In query driven simulation, one is usually interested in certain aspects of a subset of potential objects. The Active KDL FOR construct, which is the principle feature of the query language, facilitates this type of retrieval.

6.1. QDS Projection Queries

Projection queries are used to extract information from a collection of objects. In particular, any subset of functions defined for the object-type may be applied. If the function is an attribute (or a member), then its stored value is returned. If it is a heuristic, then its derived value is returned. Projection queries allow multiple views of a simulation experiment to be easily displayed. An example of a projection query is the following:

FOR ALL b IN Bank_Model APPLY Name (b), Throughput (b), Mean_Wait (b) END;

This query extracts information from the collection of objects *val* (Bank_Model) by applying the Throughput and Mean_Wait heuristics.

6.2. QDS Selection Queries

Selection queries are intended to extract object instances of type T satisfying some additional constraint p. Most commonly, query driven simulation will be carried out on a single model. In this case, queries will be a combination of selection and projection capabilities. For this type of querying, let us illustrate how query driven simulation works by giving successively more complex examples.

6.2.1. QDS Point Queries

Suppose that an analyst wishes to know the customer throughput and the average time that customers spend waiting for the teller. In particular, if the analyst is interested in the performance of the bank given that the mean interarrival time is 4 minutes and the mean teller service time is 3 minutes, the analyst could simply formulate the following query:

```
FOR ALL b IN Bank_Model
        WHERE Mean_Arrival (b) = 4.0 AND Mean_Service (b) = 3.0
        APPLY Throughput (b), Mean_Wait (b)
END;
```

This is a point query since the WHERE predicate is a conjunction of equality comparisons. The query is processed as a selection. Objects satisfying the WHERE predicate will be returned and the results of the function/attribute applications will be displayed. If no objects satisfy the predicate, then the answer is clearly insufficient, so information generation will be performed (assuming a non-zero threshold). Information generation for point queries is quite simple. Model input parameters are assigned values extracted from the query, specifically the WHERE predicate (e.g., Mean_Service = 3.0). Input parameters not mentioned in the WHERE predicate are set to their default values (e.g., Num_Customers = 100). These default values are found in the schema as default values for the parameters of the class constructor. Details about model instantiation may be found in [Mill90].

6.2.2. QDS Range Queries

It is frequently the case that an analyst would like to know how performance changes as a parameter is changed. Range queries provide this capability. Suppose an analyst wishes to know how the performance of the bank changes with the efficiency of the teller. The following query, in which the Mean_Service input parameter is varied, provides the answer.

```
FOR ALL b IN Bank_Model
        WHERE Mean_Arrival (b) = 9.0 AND Mean_Service (b) IN {4.0, 6.0, 8.0}
        APPLY Mean_Service (b), Throughput (b), Mean_Wait (b)
END;
```

This is not a point query since multiple input parameter sets will be needed, one parameter set for each value of Mean_Service. The Bank_Model will be instantiated (declared, constructed and executed) once for each of the 3 input parameter sets.

6.2.3. QDS Boolean Queries

Boolean queries allow arbitrarily complex conditions to be given in the WHERE predicate. Model instantiation for such queries is relatively straightforward if the WHERE predicate is expressed in disjunctive normal form. A logical expression is said to be in disjunctive normal form if it consists of conjuncts (AND'ed terms) that are OR'ed together. Each of the conjuncts

is used to create a single model instance. These disjuncts are independent and hence on the appropriate parallel hardware could be executed simultaneously.

Boolean queries are quite useful for making comparisons between alternate configurations of a system. For example, suppose an analyst wishes to know how significant an effect having a faster teller would have. The query below, which is in disjunctive normal form, will provide the answer.

```
FOR ALL b IN Bank_Model
        WHERE Mean_Arrival (b) = 7.0 AND Mean_Service (b) = 6.0
                OR  Mean_Arrival (b) = 7.0 AND Mean_Service (b) = 4.0
        APPLY Mean_Service (b), Throughput (b), Mean_Wait (b)
END;
```

The ith parameter set is formed from the ith conjunct together with left over default values.

```
PS #1: Mean_Arrival = 7.0, Mean_Service = 6.0, "defaults"
PS #2: Mean_Arrival = 7.0, Mean_Service = 4.0, "defaults"
```

If the WHERE predicate is not in disjunctive normal form, it is transformed to this form.

6.3. QDS Join Queries

Model instantiation becomes significantly more complex when multiple models are referenced by a query. Not only are individual models instantiated with parameter values, but joint parameterization needs to be done in a systematic and efficient manner. Typically, if systems are being compared they will have something in common, namely, input parameters (or attributes in database terminology). These common attributes are then used as the basis for a join-like operation. This capability can be used to compare different but related systems. For example, if an analyst needs to know whether the customer waiting time is greater in a client's bank or convenience store, the following query could be formulated:

```
FOR ALL b IN Bank_Model, s IN Store_Model
        WHERE Mean_Arrival (b) = Mean_Arrival (s) AND
                Mean_Arrival (b) IN {3.0, 4.0, 5.0} AND Mean_Service (b) = 3.0 AND
                Shopping_Time (s) = 2.0 AND Cashier_Service (s) = 1.0
        APPLY Mean_Arrival(b), Name (b), Mean_Wait (b), Name (s), Mean_Wait (s)
END;
```

In this query, the common attribute is Mean_Arrival. The two models will be compared for each value of this attribute. Note that if the second term in the WHERE predicate was absent, the rules for model instantiation would have set Mean_Arrival to its default value. In the case that

each model defined the default value to be different, both default values would be used.

7. Summary and Future Work

To summarize, Active KDL incorporates *data*, *knowledge*, and *model* semantics through an object-oriented view of data, knowledge, and models. Consequently, Active KDL is able to support demanding applications such as query driven simulation. The storage and retrieval of large quantities of simulation data is made easy by the query language. The execution of simulation models is no more difficult since the user need simply enter a query. Finally, the building of models is simplified since they are integrated into the database and can be composed from existing models or submodels (e.g., through the use of inheritance).

A single-user uniprocessor prototype C++ implementation of Active KDL has been completed [Kesk90b, Ucar90]. We next plan to implement a high performance parallel/distributed multi-user version of Active KDL capable of handling truly demanding applications, such as large simulation problems, CAD/CAM, and intelligent decision support systems. In this implementation, we would attempt to exploit as much parallelism as possible [Mill91a]. Another version of Active KDL, written in CLOS (Common Lisp Object System), is being used to test various design features in a more "rapid prototyping" fashion [Koch91a]. This version is also providing historical extensions [Koch91b].

The ultimate goal of our research is to continue to improve the accuracy of modeling knowledge, data, and model management applications. Our approach to achieving this goal is based on the integration of concepts and techniques from the areas of artificial intelligence, data modeling, and programming languages. We feel Active KDL is the appropriate vehicle for reaching our major goal. In addition, the use of Active KDL has been proposed in various practical settings: a University Database/Academic Advisor [Pott88a], a Decision Support System [Tidr88], a Model Manager [Wood90], and a Simulation Support System [Mill90, Pott90, Mill91b, Mill91c].

8. References

[Agra89] Agrawal, R., and N. Gehani, "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++," *Proceedings of the Second International Workshop on Database Programming Languages*, R. Hull, R. Morrison, and D. Stemple (Eds.), Gleneden Beach, OR (June 1989).

[Alas89] Alashqur, A.M., S. Su, and H. Lam, "OQL: A Query Language for Manipulating Object-Oriented Databases," *Proceedings of the Fifteenth International Conference on Very Large Databases*, Amsterdam (August 1989).

[Andr89] Andrews, T., C. Harris, and K. Sinkel, "The ONTOS Object Database," *Ontologic Technical Report*, Burlington, MA (1989).

[Atch89] Atchan, H.M., R. Bell, and B. Thuraisingham, "Knowledge-Based Support for the Development of Database-Centered Applications," *Proceedings of the Fifth International Conference on Data Engineering*, (February 1989).

[Back78] Backus, J., "Can Programming be Liberated from the von Neumann Style," *Communications of the ACM*, Vol. 21, No. 8 (August 1978).

[Banc90] Bancilhon, F., and P. Buneman, (Eds.), *Advances in Database Programming Languages*, Addison-Wesley, Reading, MA (1990).

[Beec88] Beech, D., "Intensional Concepts in an Object Database Model," *OOPSLA '88 Conference Proceedings*, San Diego, CA (September 1988).

[Burs80] Burstall, R.M., D.B. MacQueen, and D.T. Sanella, "HOPE: An Experimental Applicative Language," Technical Report, Department of Computer Science, Edinburgh University (1980).

[Caru87] Caruso, M., "The VISION Object-Oriented DBMS," *Proceedings of the Workshop on Database Programming Languages*, Roscoff, France (September 1987).

[Chan73] Chang, C., and R. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, N.Y. (1973).

[Chen76] Chen, P., "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems*, Vol. 1, No. 1 (March 1976).

[Cope84] Copeland, G., and D. Maier, "Making Smalltalk a Database System," *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (June 1984).

[Date90] Date, C.J., *An Introduction to Database Systems, Volume I, 5th Edition*, Addison-Wesley, Reading, MA (1990).

[Daya89] Dayal, U., "Queries and Views in an Object-Oriented Data Model," *Proceedings of the Second International Workshop on Database Programming Languages*, R. Hull, R. Morrison and D. Stemple (Eds.), Gleneden Beach, OR (June 1989).

[Daya88] Dayal, U., A.P. Buchmann, and D.R. McCarthy, "Rules are Objects Too: A Knowledge Model for an Active Object-Oriented Database System," *Advances in Object-Oriented Database Systems* (September 1988).

[Dewh89] Dewhurst, S., and K. Stark, *Programming in C++*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).

[Ditt86] Dittrich, K., and U. Dayal, (Eds.), *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA (September 1986).

[Fish87] Fishman, D., et al., "Iris: An Object-Oriented Database Management System," *ACM Transactions on Office Information Systems*, Vol. 5, No. 1 (January 1987).

[Good89] Goodman, N., "Object Oriented Database," *InfoDB*, Vol. 4, No. 3 (Fall 1989).

[Harp86] Harper, R., R. Milner, and M. Tofte, "Standard ML," Technical Report ECS-LFCS-86-2, Computer Science Department, Edinburgh University (1986).

[Hech81] Hecht, M.S., and L. Kerschberg, *Update Semantics for the Functional Data Model*, Database Research Technical Report, AT&T Bell Labs: DR-TR No. 4, Murray-Hill, NJ (January 1981).

[Hull87] Hull, R., and R. King, "Semantic Database Modeling: Survey, Applications and Research Issues," *ACM Computing Surveys*, Vol. 19, No. 3 (September 1987).

[Kers76] Kerschberg, L., and J. Pacheco, "A Functional Data Base Model," *Monograph Series Technical Report*, Portificia Univ. Catolica do Rio De Janeiro, Brazil (February 1976).

[Kers84] Kerschberg, L., ed., *Expert Database Systems: Proceedings From The First International Workshop*, Kiawah Island, South Carolina (1984), Benjamin Cummings, Menlo Park, CA (1986).

[Kers87] Kerschberg, L., (Ed.), *Expert Database Systems: Proceedings from the First International Conference*, Benjamin Cummings Publishing Co., Menlo Park, CA (1987).

[Kers88] Kerschberg, L., (Ed.), *Expert Database Systems: Proceedings from the Second International Conference*, Benjamin Cummings Publishing Co., Menlo Park, CA (1988).

[Kesk90a] Keskin, A.A., "Strategies for Deductive Databases," *Proceedings of the 28th Annual ACM Southeastern Regional Conference*, Greenville, SC (April 1990).

[Kesk90b] Keskin, A.A., "A Query and Rule Processing System for the KDL Object-Oriented Database System," Masters Thesis, University of Georgia (August 1990).

[Kim89a] Kim, W., "A Model of Queries for Object-Oriented Databases," *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, Amsterdam (1989).

[Kim89b] Kim, W., and F.H. Lochovsky (Eds.) *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, Reading, MA (1989).

[Koch91a] Kochut, K.J., J.A. Miller, and W.D. Potter, "Design of a CLOS Version of Active KDL: A Knowledge/Data Base System Capable of Query Driven Simulation," *Proceedings of the 1991 AI and Simulation Conference*, New Orleans, LA (April 1991).

[Koch91b] Kochut, K.J., J.A. Miller, W.D. Potter, and A.D. Wright, "h-KDL: A Historically Extended Functional Object-Oriented Database System," *Proceedings of the Tools '91 International Conference*, Santa Barbara, CA (July 1991). (to appear)

[Kris88] Krishnamurthy, R., R. Ramakrishnan, and O. Shmueli, A Framework for Testing Safety and Effective Computability of Extended Datalog," *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (June 1988).

[Lee90] Lee, K., and S. Lee, "An Object-Oriented Approach to Data/Knowledge Modeling Based on Logic," *Proceedings of the Sixth International Conference on Data Engineering* (February 1990).

[Lecl88] Lecluse, C., P. Richard, and F. Velea, "O_2, an Object-Oriented Data Model," *Proceedings of the ACM-SIGMOD International Conference on Management of Data* (June 1988).

[Maie86] Maier, D., J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," *OOPSLA '86 Conference Proceedings*, Portland, OR (September 1986).

[Mill91a] Miller, J.A., and N.D. Griffeth, "Performance Modeling of Database and Simulation Protocols: Design Choices for Query Driven Simulation," *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, LA (April 1991).

[Mill91b] Miller, J.A., K.J. Kochut, W.D. Potter, E. Ucar, and A.A. Keskin, "Query Driven Simulation Using Active KDL: A Functional Object-Oriented Database System," *International Journal in Computer Simulation*, Vol. 1, No. 1 (1991). (to appear)

[Mill91c] Miller, J.A., O.R. Weyrich, Jr., W.D. Potter, and V.C. Kessler, "The SIMODULA/OBJECTR Query Driven Simulation Support Environment," *Progress In Simulation*, Vol. 3, Leonard-Zobrist Editors (1991). (to appear)

[Mill90] Miller, J.A., W.D. Potter, K.J. Kochut, and O.R. Weyrich, Jr., "Model Instantiation for Query Driven Simulation in Active KDL," *Proceedings of the 23rd Annual Simulation Symposium*, Nashville, TN (April 1990).

[Mill89] Miller, J.A., and Orville R. Weyrich, Jr., "Query Driven Simulation Using SIMODULA," *Proceedings of the 22nd Annual Simulation Symposium*, Tampa, FL (March 1989).

[Mylo89] Mylopoulos, J., and M.L. Brodie, (Eds.), *Readings in Artificial Intelligence and Databases*, Morgan Kaufmann Publishers, Inc., San Mateo, CA (1989).

[Nier89] Nierstrasz, O., "A Survey of Object-Oriented Concepts," *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F.H. Lochovsky (Eds.), Addison-Wesley, Reading, MA (1989).

[Pott91] Potter, W.D., K.J. Kochut, J.A. Miller, V.P. Gandham, and R.V. Polamraju, "The Evolution of the Knowledge/Data Model," *Advances in Databases and Artificial Intelligence*, Petry-Delcambre Editors (1991). (to appear)

[Pott90] Potter, W.D., J.A. Miller, K.J. Kochut, and S.W. Wood, "Supporting an Intelligent Simulation/Modeling Environment Using the Active KDL Object-Oriented Database Programming Language," *21st Annual Pittsburgh Conference on Simulation and Modeling*, Pittsburgh, PA (May 1990).

[Pott89] Potter, W.D., R.P. Trueblood, and C.M. Eastman, "Hyper-Semantic Data Modeling," *Data & Knowledge Engineering*, Vol. 4, No. 1 (July 1989).

[Pott88a] Potter, W.D., "KDL-ADVISOR: A Knowledge/Data Based System Written in KDL," *Proceedings of the 21st Annual Hawaii International Conference on System Science*, Kailua-Kona, Hawaii (January 1988).

[Pott88b] Potter, W.D., and R.P. Trueblood, "Traditional, Semantic and Hyper-Semantic Approaches to Data Modeling," *IEEE Computer*, Vol. 21 , No. 6 (June 1988).

[Pott87] Potter, W.D., R.P. Trueblood, C.M. Eastman, and M.M. Mathews, "KDL: A Hyper-Semantic Data Model Specification Language," *Proceedings of the 2nd International Symposium on Methodologies for Intelligent Systems, Colloquia Program*, Charlotte, NC (October 1987).

[Pott86] Potter, W.D., and L. Kerschberg, "A Unified Approach to Modeling Knowledge and Data," *Proceedings of the IFIP TC2 Conference on Knowledge and Data (DS-2)*, Algarve, Portugal (November 1986). (Published by North-Holland as *Data and Knowledge DS-2* (1988).)

[Risc88] Risch, T., R. Reboh, P. Hart, and R. Duda, "A Functional Approach to Integrating Database and Expert Systems," *Communications of the ACM*, Vol. 31, No. 12 (December 1988).

[Shaw89] Shaw, G.M., and B. Zdonik, "An Object-Oriented Query Algebra,", *Proceedings of the Second International Workshop on Database Programming Languages*, R. Hull, R. Morrison, and D. Stemple (Eds.), Gleneden Beach, OR (June 1989).

[Ship81] Shipman, D., "The Functional Data Model and the Data Language DAPLEX," *ACM Transactions on Database Systems*, Vol. 6, No. 1 (March 1981).

[Stei88] Stein J., and D. Maier, "Concepts in Object-Oriented Data Management," *Database Programming and Design*, Vol. 1, No. 4 (April 1988).

[Ston86] Stonebraker, M., and L. Rowe, "The Design of POSTGRES," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, Washington, DC (June 1986).

[Sun90] *SunOS 4.1, System Services Overview*, Sun Microsystems (1990).

[Tidr88] Tidrick, T.H., W.D. Potter, and R.I. Mann, "New Directions for DSS: Integrating Data, Knowledge, and Model Management," *Proceedings of the 26th Annual ACM Southeastern Regional Conference* (1988).

[Turn85] Turner, D.A., "MIRANDA: A Non Strict Functional Language with Polymorphic Types," *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture*, (Lecture Notes in Computer Science, Vol. 201) Nancy, France (1985).

[Ucar90] Ucar, E., "Schema Processing in the KDL Object-Oriented Database System," Masters Thesis, University of Georgia (September 1990).

[Urba90] Urban, S., and L. Delcambre, "Constraint Analysis: A Design Process for Specifying Operations on Objects," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 4 (December 1990).

[Wood90] Wood, S.W., "Model Management From An Object-Oriented Perspective," unpublished manuscript, Artificial Intelligence Programs, University of Georgia (1990).

[Zdon90] Zdonik, S.B., and D. Maier, (Eds.), *Readings in Object-Oriented Databases*, Morgan Kaufmann Publishers, Inc., San Mateo, CA (1990).