Query Driven Simulation Using Active KDL:
A Functional Object-Oriented Database System

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

∗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

Query Driven Simulation Using Active KDL: A Functional Object-Oriented Database System (FOODS)

Abstract

Because of the difficulty of simulating large complex systems with traditional tools, new approaches have and are being developed. One group of interrelated approaches attempts to simultaneously make simulation modeling and analysis easier while at the same time providing enough power to handle more complex problems. This group includes the following important (overlapping) approaches: integrated simulation support environments, object-oriented simulation, and knowledge-based simulation. Query driven simulation fits somewhere in the middle of these three approaches. Its fundamental tenant is that simulationists or even naive users should see a system based upon query driven simulation as a sophisticated information system. A system/environment based upon query driven simulation will be able to store information about or to generate information about the behavior of systems that users wish to study. Active KDL (Knowledge/Data Language), which is a functional object-oriented database system (FOODS), supports query driven simulation by providing access to integrated data, knowledge, and model bases. The three sublanguages of Active KDL provide strong support for simulation modeling and analysis: Complex simulations may be structured using the schema definition language (SDL); results may be retrieved using the query language (QL); and models (which may include active objects) may be implemented using the database programming language (DBPL).

Keywords: object-oriented databases, knowledge bases, object-oriented simulation, knowledge-based simulation, and integrated simulation support environments.

1. Introduction

Because of the difficulty of simulating large complex systems with traditional tools, new approaches have and are being developed. One group of interrelated approaches attempts to simultaneously make simulation modeling and analysis easier while at the same time providing

enough power to handle more complex problems. This group includes the following important (overlapping) approaches: integrated simulation support environments, object-oriented simulation, and knowledge-based simulation. There are two aspects of simulation that can lead to overwhelming complexity. First, simulation modeling and analysis uses and generates a huge amount of information. The management of this information has traditionally been handled by limited ad hoc means. *Integrated simulation support environments* (e.g., TESS [1], [2]) have been of considerable help in this area. Second, the design, implementation, verification, and validation of complex simulation models from scratch is a formidable task. The closely related approaches of *object-oriented simulation* (e.g., SIMULA/DEMOS [3], [4], MODSIM II [5] and *knowledge-based simulation* (e.g., KBS [6], [7] ROSS [8], [9], and SES/MBase [10], [11]) allow new models to be composed from existing models, thereby enhancing the process of model development.

Query driven simulation is a powerful approach to simulation modeling and analysis [12], [13], [14], [15]. It fits somewhere in the middle of the three approaches discussed above. The basic motivation or premise behind query driven simulation is quite simple. Simulationists or even naive users should see a system based upon query driven simulation as a sophisticated information system. They should be able to interact with it at whatever level of detail they desire. A system/environment based upon query driven simulation will be able to store information about or to generate information about the behavior of systems that users wish to study. For the most part, users interact with the system simply by formulating queries.

Our first attempt at developing a query driven simulation system could be described as loosely coupled. This approach coupled a process-interaction based simulation language, called SIMODULA, with a simple single-user relational database system, called OBJECTR [13], [14], [16]. Both were written in Modula-2. To help facilitate the storage of complex objects, OBJECTR provided a simple object-oriented extension. It allowed attributes to be of opaque type. [Opaque types are Modula-2's version of abstract data types.] However, to provide all the sophisticated information management needs involved in simulation (e.g., parameters, statistics,

histograms, graphs, animations, and models), a database management system with greater functionality than OBJECTR is required. Furthermore, to enhance the ease of use and to provide a more uniform environment, a more tightly coupled approach is called for.

We are currently in the process of completing a prototype implementation of a tightly coupled system supporting the query driven simulation approach. The heart of the system is an *object-oriented database system* called Active KDL (Knowledge/Data Language). [Much of the current research and development in the database field involves object-oriented database systems [17], [18], [19], [20], [21], [22] since they provide more powerful constructs for structural and behavioral specification.]

Active KDL is designed to support the complex information needs of *engineering databases* and *expert databases*. The fact that Active KDL is built from solid theoretical foundations (i.e., object-oriented programming, functional programming, and hyper-semantic data modeling) allows Active KDL to meet the needs of demanding applications (e.g., simulation, model management, CAD/CAM, and intelligent database applications such as a university data/knowledge base capable of advising students). In particular, Active KDL provides an integration of *model bases*, *knowledge bases*, and *databases*. Simulation inputs and outputs can be stored by Active KDL since it supports complex objects. More importantly, Active KDL also allows users to specify rules to capture heuristic knowledge and methods to specify complex behaviors or computations. Finally, Active KDL provides a simple mechanism for specifying concurrent execution, namely tasks embedded in active objects. The use of threads (lightweight processes) and coroutines as mechanisms for concurrency allows tasks to run concurrently [23]. These facilities provide a powerful mechanism for building simulation models out of pre-existing model components.

We designed Active KDL as an enhancement of KDL, a hyper-semantic data modeling language developed by Potter [24], [25], [26], [27], [28]. Recently, a prototype implementation of Active KDL in C++ has been completed [29], [30], [31]. This implementation exploits the already present data (or object) modeling capabilities of C++ [32], [33]. Active KDL schema

and queries are translated to C++ (with the help of lex and yacc/bison compiler tools), at which point the C++ code is compiled and executed. Currently, several research projects are either extending C++ to make it a database programming language (e.g., O++ [17], E [34]) or, as we are doing, translating a simpler and higher level language to C++ (e.g., ONTOS (new version of Vbase) [18]).

Users predominantly interact with the system by formulating queries to retrieve stored data, data inferred from knowledge, or data generated from models. The system allows access to not only data, but also knowledge and models. Queries formulated in Active KDL's query language, allow users to retrieve information about the behavior of systems under study. Data from simulation runs is stored by the system. If the information requested by a query is already stored in the database, more or less conventional query processing techniques will be applied. However, if the amount of information retrievable is below a user-settable threshold, model instantiation and execution will be automatically carried out to generate the desired information.

Model instantiation [12] involves the creation of model instances that can then be executed. In its most elementary form, a model instance is created by parameterizing a simulation model with values formed by schema and query analysis. The model is then executed to generate information which is stored in the database and returned to the user. Many queries however, will require that several model instances be created. Finally, complex queries will require the systematic instantiation and execution of multiple models.

2. Language Design Goals and Motivation

A strong interest in object-oriented database systems and a renewed interest in semantic data modeling [35] research began in the mid-1980's with the merger of programming language, database, and artificial intelligence ideas [36], [37]. As a result, research in object-oriented databases has become very intensive [21], [38]. Much of this research has focused on the integration of object-oriented programming with data modeling. In addition, data modeling and artificial intelligence integration has received much attention [39], [40], [41]. Active KDL integrates ideas from all three of these areas.

Active KDL is in the category of Functional Object-Oriented Database Systems (FOODS). The underlying foundation of Active KDL is the hyper-semantic data model KDM (Knowledge/Data Model) [28]. Hyper-semantic data models provide extensive modeling capabilities that subsume those of standard semantic data models and the popular object-oriented data models [24], [26], [42], [43]. The scope of modeling capabilities for hyper-semantic data models includes knowledge, data, and model management [12], [15], [44]. The KDM evolved from the Functional Data Model (FDM) [45], [46]. 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 which is supported via the Active KDL. Active KDL's functional character allows it to handle 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 [51], [52], [53]).

The functional paradigm focuses on data values described by means of functional expressions. Expressions are constructed from function applications. This leads to programs built only from functions, and therefore side-effect free evaluation of programs. Such a programming style has many advantages over traditional imperative programming [54]. The design of Active KDL was based in part on some of the existing functional programming languages such as Hope [55], Standard ML [56], and Miranda [57].

To summarize the desirable characteristics of Active KDL, we will briefly mention some of the important ones (see [58] for more details). Active KDL serves equally well as a knowledge/data modeling language, a query language, and a knowledge/data manipulation/application language. Specifically, it consists of a schema definition language (SDL), a query language (QL), and a database programming language (DBPL) (which is a super-

---

† The functional paradigm is the focal point of many database research efforts (e.g., Vision [47], HiPAC [48], O_2 [49], and Syntel [50]).

set of the query language). The language is functional at the schema design level, the query level, and at the application level. The introduction of side-effects resulting from combining the functional and the object-oriented programming paradigms is precisely identified and kept to a minimum. Active KDL supports hyper-semantic data modeling which is based on the standard abstraction mechanisms found in semantic data models, plus constraints, heuristics/methods, and temporal specifications. Of all the abstraction mechanisms available in Active KDL, the temporal abstraction mechanisms are the most complex. Active KDL supports two types of temporal abstractions. First, it supports active objects, that is, objects performing some task. And second, it allows delayed actions to be performed as the result of a trigger being fired. Both capabilities are very useful for simulation applications.

The fact that Active KDL is object-oriented and includes a database programming language, enables simulation models to be developed in an object-oriented fashion like in object-oriented simulation languages such as MODSIM II. The database programming language is smaller and simpler than a typical programming language. However, we see this as an advantage making it easier to learn and use. The fact that Active KDL is non-procedural (i.e. declarative) coupled with the facts that AI like rules may be expressed in a functional style (see [58] for details) and that it is object-oriented (i.e. frame like) allows it to function as a knowledge base [25]. Finally, since Active KDL is a sophisticated database system, it automatically provides the core part of an integrated simulation support environment.

3. The Active KDL Type System

To support the complex data, knowledge, and model processing needs of query driven simulation, a well defined type system is of paramount importance. Object-types (or classes) form the foundation of the Active KDL type system. Also primitive types and collection types may be used to build up object-types. Active KDL has four primitive data types, namely INTEGER, REAL, CHAR, and BOOLEAN.

Active KDL supports two predefined collection types: set and list. 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 Section 3.2.] In Active KDL, anything that is set-valued (i.e., an object-type or a set type) may not contain duplicates. A *list type* is similar to a set type except that 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 (i.e., STRING is a system defined alias for LIST OF CHAR).

3.1. Object-Types

Object-types are the main building blocks 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). 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.

```
object–type ::=
    OBJECT_TYPE class–name HAS
        [ SUPERTYPES:
            class–name { , class–name } ; ]
        [ SUBTYPES:
            class–name { , class–name } [ HIDING function–list ] ; ]
        [ ATTRIBUTES:
            { attribute–name : type–name
                [ WITH CONSTRAINT: constraint ] ; } ]
        [ MEMBERS:
            { member–name : [ SET OF | LIST OF ] class–name
                [ INVERSE OF member–name [ (class–name ) ] ]
                [ WITH CONSTRAINT: constraint ] ; } ]
        [ CONSTRAINTS:
            { constraint
                [ WITH TRIGGER: activity ] ; } ]
        [ HEURISTICS:
            { rule ; } ]
        [ METHODS:
            { method ; } ]
    END class–name ;

constraint ::= constraint–name (parameter : class–name ): BOOLEAN = predicate
rule ::= rule–name (parameter : class–name ): type = query
method ::= method–name (parameter–list ): type [ = expression ]
activity ::= action [ AFTER expression UNITS action ]
```

Figure 1: Syntax of Object-Type Definition.

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.

3.1.1. Functions

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

1. Attributes are stored functions constituting the data stored within an object.
1.b. Memberwise attributes (or members) are stored functions referencing other objects.
2. Constraints are Boolean functions.
3. Heuristics are functions or rules expressed using the QL.
4. Methods are quasi-functions (may have limited side-effects) expressed using the DBPL.

3.2. Inheritance in Active KDL

Inheritance mechanisms facilitate code reusability and structuring of components of complex software systems. This can be used in simulation systems to derive new simulation models out of existing models (e.g., a bank simulation model that includes loan officers can easily be derived from a simpler model that includes only tellers). Since Active KDL supports multiple inheritance, a set of object-types (classes) declared in schema specifications may form an inheritance lattice. [An inheritance lattice is a generalization of an inheritance hierarchy.]

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 solely for the purpose of code reusability in case of partial inheritance [59]. To make our discussion of inheritance in Active KDL more precise, it is imperative to isolate two aspects of inheritance (an intensional aspect and an extensional aspect).

The intensional state of the database represents the type structure for all the types (or classes) defined for the database. The intensional state is modified both by schema evolution and

by ordinary querying. 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 of a type is a *subset* of the extension of each of its supertypes. Formally, this can be captured as a binary relation on types. More specifically, the relation that exists between types is that of a *family of join lattices*, where join lattice *i*

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

is a partial order (reflexive, antisymmetric, and transitive) in which every pair of types has a least upper bound (lub). The lub property states that for all pairs of types S and T in join lattice i, an lub of S and T must also be a type in join lattice i, i.e., $lub\,(S, T) \in TYPES_i$.

A natural way to represent relations is with a minimal *digraph*‡. 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* (Directed Acyclic Graph). Note, Active KDL does not enforce the lub property once and for all by using a global dummy root (e.g., Object), as is done in other systems. 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 a subtype. Furthermore, any object that is an `instance-of` a type, is a `member-of` the extension of that

---

† 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.

‡ The minimal digraph D_i is the smallest digraph for which $D_i^* = L_i$, where the * operator is the reflexive and transitive closure.

type, and also a `member-of` the extension of all its supertypes.

We will say that 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 which is part of the meta-data schema (see Figure 2).

In Active KDL, a type represents the *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 is stored in the meta-data if the result of the query is saved. The function fam(T), which is a particularly important meta-data heuristic, returns the transitive closure of all subtypes for T, including the type T itself. It represents a sub-digraph of nodes that is rooted at the node representing T in the inheritance lattice.

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 for a given type T, denoted $val(T)$, represents the *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* value. More generally, if T is not a terminal object-type (i.e., it has additional object-types derived from it) then

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

4. Active KDL as a Data Modeling Language: Simulation Model Construction

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

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

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:
                sch:  Schema INVERSE OF Classes (Schema);       // Schema defined in
                dig:  Digraph;                      // Digraph that this node is in
                sup:  SET OF Node;                  // Supertypes in SUPERTYPES clause
                sub:  SET OF Node;                  // Subtypes in SUBTYPES clause
        CONSTRAINTS:
                Full_Inheritance (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 sub (m) = n APPLY m END;
                Sub (n: Node): SET OF Node =            // Immediate subtypes
                        sub (n) +
                        FOR ALL m IN Classes (dig (n)) WHERE sup (m) = n 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 Node =            // 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;
```

Figure 2: Important Meta-Data Object-Types.

section addresses the fundamental abstraction mechanisms provided by Active KDL. Furthermore, it shows how these mechanisms can be used in the construction of simulation models.

The first part of this section discusses 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, aggregation, and association (or membership). 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 inheritance lattice.

The last part of this section discusses how heuristics/methods, and constraints (two abstraction mechanisms represenating special types of computations) can be used as the basic building blocks for simulation models and simulation analysis. Among its other uses, heuristics are very useful in analyzing the results of simulation. Active KDL supports two of the three major *simulation world views*. The three major world views are:

1. Event-Scheduling (ES). The popularity of ES stems from its efficiency in execution and its ease of implementation in conventional languages like Fortran and Pascal.
2. Process-Interaction (PI). In recent years, PI has gained in popularity because it is well supported by many of the simulation languages and even some general-purpose programming languages like Modula-2 and C++ (with its task library). PI provides simulationists with a higher-level view than does ES.
3. Activity-Scanning (AS). The popularity of AS is currently on the increase in part because of its strong similarity and high compatibility with expert system like production rules.

Because Active KDL is designed to operate at a higher level, closer to the conceptualizations of users, than conventional programming languages, it supports only the latter two world views. Methods facilitate the specification of PI simulations, while constraints facilitate the specification of AS simulations. [Note that temporal aspects of Active KDL are coupled with both methods and constraints.]

4.1. Specialization

Subtypes (or subclasses) of an existing type (referred to as the supertype) can be defined. The object instances in the subtype are also members 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. Therefore, 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 *Specialized_Type* HAS
    SUPERTYPES: *Type* ;
    ...
END *Specialized_Type* ;

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

$$fun(Specialized_Type) = fun(Type) \cup \{ f_1, \ldots, f_n \}$$
$$val(Specialized_Type) \subseteq val(Type)$$

Specialization may be used to derive a more particularized simulation model from a currently existing one. For example, a more elaborate bank simulation model could be derived from the simple bank model given in Section 4.6.1.

4.1.1. 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 *Joint_Type* HAS
    SUPERTYPES: *S* , *T* ;
    ...
END *Joint_Type* ;

The functions defined on *Joint_Type* are simply the union of those defined in the supertypes and those added by the type itself. If functions have the same name, then the first one encountered, by following the ordering given in the SUPERTYPES clause, is the one taken.

$$fun(Joint_Type) = fun(S) \cup fun(T) \cup \{ f_1, \ldots, f_n \}$$

From an extensional point of view, the situation is also straightforward. Since *Joint_Type* `is-a` S and *Joint_Type* `is-a` T, objects that are instances of *Joint_Type* are also members of S and members of T. If $top(S) = top(T)$, then the two types have some fundamental compati-

bility, since they belong to the same lattice. Consequently, their combination through multiple inheritance is legal and extensionally speaking results in the following:

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

As an example taken from the Bank_Simulation, consider the Customer and Student classes (see Figures 6 and 4 respectively). Since the classes S = Customer and T = Student are in the same lattice (i.e., $top(S) = top(T)$ = Coroutine), multiple inheritance may be used to define a new type called Customer_Stud. An object that is in *val*(Customer_Stud) must also be in *val*(Customer) and in *val*(Student). 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 relationship between these object-types is better described with `part-of` relationships rather than with `is-a` relationships. In some systems, `part-of` relationships (aggregation) are coupled with inheritance. However, in Active KDL `part-of` relationships are manifested through attributes (see below).

4.2. 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 *Generalized_Type* HAS
    SUBTYPES: S, T HIDING $f_1, \ldots, f_n$;
END *Generalized_Type*;

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

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

4.3. Aggregation

As discussed above, in some languages multiple inheritance can be used to define a new type called Car from Engine and Body. In such a case, a car object is composed of an engine object and a body object, but it is neither. That is, a Car is not an Engine, but rather an Engine is `part-of` a Car. This is a form of aggregation known as object-level aggregation [60]. Active KDL does not permit this form of inheritance, but rather provides aggregation using attributes.

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 a sub-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. Since attributes are stored functions, they constitute the information stored within an object. This information exists only so long as the object itself exists. Consequently, when an object is deleted all of its sub-objects must also be deleted. The example below illustrates the use of attributes for aggregation.

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

Figure 3: Engine and Body Aggregation.

What happens if an inexperienced data modeler using Active KDL attempts to define a Car object-type as a multiple specialization of (rather than an aggregation of) Engine and Body? Active KDL will complain about the fundamental dissimilarity of Engine and Body (they are in separate lattices). The persistent user will observe that he can still achieve the goal simply by generalizing Engine and Body, thereby merging two lattices. Since generalizations may have a fundamental impact on the use and semantics of existing object-types, we believe that in some systems using Active KDL it may be prudent to restrict the use of generalization to experienced

users. Therefore, Active KDL allows users to be assigned different privileges in regards to the capabilities of performing specialization and generalization.

4.4. Association (or Membership)

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 (known as members) enable associations between objects to be formed (an object is `associated-with` some other object(s)). The type of a member must be either an object-type or a collection type whose base type is an object-type. The memberwise attributes (or members) are defined in the MEMBERS clause. For example, a simulation study of the course enrollment process at a university would include Student and Course object-types. Associations between students and courses can be easily established as shown in Figure 4. The association shown in the figure is established via the memberwise attributes (Students and Courses) and is an example of a many-to-many relationship. 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 [61]. Therefore, all forms of relationships or associations provided by popular semantic data models, such as the Entity-Relationship Model (ERM) [62], 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 ERM.

Association looks much like aggregation in that they both use a kind of attribute (memberwise versus ordinary, respectively). However, association is a weak coupling between objects, not nearly so strong as what is suggested by the meaning of `part-of`. Objects in an association enjoy equal, independent status, while the sub-objects in an aggregation do not. The fact

```
OBJECT_TYPE Person HAS
        SUPERTYPES:
                Sim_Object;                     // Active simulation object
        ATTRIBUTES:
                SSN:   INTEGER;
                Name: STRING;
                Sex:    CHAR;
                City:   STRING;
        MEMBERS:
                Parents: SET OF Person;
        HEURISTICS:
                Ancestors (p: Person): SET OF Person =
                        Parents (p) + Ancestors (Parents (p));
        METHODS:
                Create (): Person;              // Constructor
END Person;

OBJECT_TYPE Student HAS
        SUPERTYPES:
                Person;
        ATTRIBUTES:
                GPA:  REAL:
        MEMBERS:
                Courses: SET OF Course;     // An enrollment relationship
        METHODS:
                Create (): Student;             // Active student object registers for courses
END Student;

OBJECT_TYPE Course HAS
        ATTRIBUTES:
                Course_Num: INTEGER;
                Course_Name: STRING;
        MEMBERS:
                Students:       SET OF Student INVERSE OF Courses (Student)
                        WITH CONSTRAINT:
                                Enrollment_Cap (c: Course): BOOLEAN =
                                        COUNT (Students (c)) < 50;
        METHODS:
                Create (num: INTEGER; name: STRING): Course;  // Create a course
END Course;
```

Figure 4: Student and Course Association.

that a course object 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 should also be deleted.

4.5. Heuristics and Simulation Results

Heuristics allow information about an object to be derived or inferred, rather than retrieved using a traditional query against stored data. Thus, heuristics provide an information derivation

mechanism. A particular heuristic for an object-type is expressed as a rule written in a functional notation. Because the rules are expressed functionally (declaratively), 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 [63].

Heuristics give Active KDL fundamentally more expressive power than a relationally complete language like SQL. For example, the Ancestors heuristic, defined as a recursive rule, in the Person class is a case in point. In Active KDL, 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.

Heuristics are very useful in simulation studies, since they can be used to generate a variety of simulation results. The data generated by simulation runs may be somewhat raw and unwieldy. Heuristics can used on this data to create presentation data. For example, in the bank simulation (see the next section) two such heuristics are used, Mean_Wait and Throughput. In general, heuristics can be used to derive a variety of statistical estimates, produce graphs, or even perform qualitative analysis. Although these are the most common uses of heuristics, they can also be used for such things as stopping rules and even model selection (see Section 5.2.3).

4.6. Methods and Process-Interaction Based Simulation

Closely related to heuristics are methods. 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. Although the database programming language is still under development [23], its basic constructs have been designed and will be briefly discussed below†.

† A version of Active KDL is being implemented that allows methods to be written directly in

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.

The bodies of methods in their most common form are simply expressions to be evaluated. All expressions in Active KDL denote values. A *simple expression* is a constant, a function or method application, or an object-type name. A *function application* is composed of a function name followed by one or more arguments (some function applications such as +, -, etc. are written in infix notation for notational convenience). Function applications with more than one argument are method applications, while applications with a single argument are not restricted. Active KDL provides a number of built-in functions (arithmetic functions, list and set manipulation functions, etc.). Following the usual functional style, any function application may be used as one of the arguments of another function application.

FOR expressions are the central language construct provided by Active KDL. [Note, the FOR construct used by the query language is a minor variation of this one (see Section 5).] In principle, the FOR expression is similar to ZF notation (Zermelo-Fraenkel set notation) also used in other functional languages, such as Miranda [57] and ML [56].

C++ (see [12]).

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, each *query*$_i$ may involve free occurrences of the preceding *variable*$_j$ ($1 \leq j < i$). The query may thus depend on values taken by the preceding variables.

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 attri-

bute 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 [64]). 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.

4.6.1. Process-Interaction

In the process-interaction 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 that share a single CPU.

Since a process is defined as a sequence of instructions, the idea is a little incongruent with the functional paradigm, where the ordering of the evaluation of a function is not prescribed. However, the process-interaction approach requires some form of temporal ordering of sub-expression evaluations. Fortunately, continuations [65] allow the equivalent of coroutines to be integrated into a functional style of programming [23]. Therefore, we are using this mechanism in Active KDL as the basis for process-interaction based simulations.

In Active KDL, simulation processes are known as Sim_Object's. The Sim_Object class is derived from the Coroutine class. The coroutine capability is made available in our translation

```
OBJECT_TYPE Sim_Object HAS
        SUPERTYPES:
                Coroutine;
        ATTRIBUTES:
                Activation_Time:        REAL;
                Priority:               INTEGER;
        METHODS:
                Create (priority: INTEGER = 0): Sim_Object;
                        // Create and begin evaluating a Sim_Object.

                Reactivate (queue: LIST OF Sim_Object); time_Delay: REAL = 0):
                                LIST OF Sim_Object;
                        // Reactivate the first Sim_Object in queue after a time_Delay.
                        // Return the new queue.

                Work (time_Delay: REAL; s: Sim_Object): Sim_Object;
                        // Delay the evaluation of the current Sim_Object by time_Delay units.
                        // Return the sub-evaluation represented by the parameter s.

                Suspend (queue: LIST OF Sim_Object; s: Sim_Object): Sim_Object;
                        // Suspend the evaluation of the current Sim_Object in queue.
                        // It remains suspended until explicitly reactivated.
                        // Returns the Sim_Object s.

                Destroy (s: Sim_Object): BOOLEAN;
                        // Destroy the Sim_Object s if acceptable.
END Sim_Object;
```

Figure 5: Sim_Object Object-Type.

target language, C++, in one of two ways. It is provided in the *task class* [32] which is part of the standard library that comes with the *AT&T C++ Language System, Release 2.0*. It has also been implemented by us to run under GNU C++ by using the SunOS 4.x Lightweight Processes Library [66]. In particular, quasi-concurrency comes about since invoking a class constructor causes the creation and execution of a new coroutine. This coroutine will continue to execute until it executes a statement that transfers control away from it. When control is transferred back to this coroutine, it resumes where it left off. Any class derived from the Sim_Object class will have concurrent capabilities, and is said to be an active class (consisting of active objects). [Note, there exist other types of active objects in Active KDL (see [23]).]

To demonstrate the simulation capabilities of Active KDL, we show two implementations of a highly simplified bank simulation. In this section, we give a process-interaction based implementation, while in the next section we give an activity-scanning based implementation. 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).

```
SCHEMA Bank_Simulation;

OBJECT_TYPE Customer HAS
        SUPERTYPES:
                Person;
        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;

END Bank_Simulation;
```

Figure 6: Process-Interaction Based Bank Simulation.

The methods specified in the above schema 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).]

4.7. Constraints and Activity-Scanning Based Simulation

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 allow for various types of constraint enforcement to be carried out (e.g., update-driven, query-driven, and user-initiated).

A particularly interesting feature of Active KDL's constraint mechanism is that a TRIGGER subclause may be specified. If the constraint is violated, a system defined response will be taken, unless a TRIGGER is specified. In this case, a corrective activity will be initiated. An activity has a starting action, and optionally a time duration and finishing action. This facility in Active KDL permits both *expert system like rule processing* [67] to be performed (e.g., forward-chaining), and simulation based on the *activity-scanning world view* [68] to be performed.

4.7.1. Activity-Scanning

In the activity-scanning world view, activities are triggered by constraints that are no longer satisfied. The activities have a beginning action, a finishing action and a time duration. The beginning and finishing actions can effect changes to the simulation. The time duration may be

```
IMPLEMENTATION Bank_Simulation;

Customer.Create (b: Bank): Customer [ Person.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} APPLY
                        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);

END Bank_Simulation;
```

Figure 7: Method Implementation for Bank Simulation.

a given constant or computed as the result of some expression (e.g., an exponentially distributed random variate). The sequencing of action execution is handled by a standard three-phase

scheduler/inference engine [68]. Starting actions are conditionally sequenced, while finishing actions are executed when the simulated time reaches the completion time for the activity.

5. Active KDL as a Query Language: Query Driven Simulation

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. Active KDL's query language is strictly more powerful 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 information intensive applications. Query driven simulation 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, queries triggering an expert system like chain of inference, or even model instantiation.

As mentioned in the introduction, model instantiation [12] 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 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%,

```
OBJECT_TYPE Bank_Model HAS
    ATTRIBUTES:
        Name:              STRING;          // Name of bank
        Num_Customers:     INTEGER;         // Number of customers
        Mean_Arrival:      REAL;            // Mean interarrival time
        Mean_Service:      REAL;            // Mean service time
        Just_Arrived:      BOOLEAN;         // Arrival indicator
        Server_Busy:       BOOLEAN;         // Server status indicator
        Num_In_System:     INTEGER;         // Number in system
        Customer_Num:      INTEGER;         // Customer counter
        Arrival_Times:     LIST OF REAL;    // Arrival times for each customer
        Departure_Times:   LIST OF REAL;    // Departure times for each customer
    MEMBERS:
        Stream:            Ran_Stream;   // Random variate stream
    CONSTRAINTS:
        No_Arrival (b: Bank_Model): BOOLEAN = NOT Just_Arrived (b)
            WITH TRIGGER: Sleep (b)
            AFTER Exponential (Stream (b), Mean_Arrival (b)) UNITS Make_New_Arrival(b);
        No_New_Work (b: Bank_Model): BOOLEAN = Num_In_System (b) = 0 OR Server_Busy (b)
            WITH TRIGGER: Begin_Service (b)
            AFTER Exponential (Stream (b), Mean_Service (b)) UNITS End_Service (b);
    HEURISTICS:
        Mean_System_Time (b: Bank_Model) =
            (SUM (Departure_Times) - SUM (Arrival_Times)) / Num_Customers;
    METHODS:
        Create (stream: INTEGER = 1; num_Customers: INTEGER = 100;
                mean_Arrival: REAL = 8.0; mean_Service: REAL = 7.0): Bank_Model =
            CREATE
                Name            = "Bank";
                Num_Customers   = num_Customers;
                Mean_Arrival    = mean_Arrival;
                Mean_Service    = mean_Service;
                Just_Arrived    = TRUE;
                Server_Busy     = FALSE;
                Num_In_System   = 0;
                Customer_Num    = 1;
                Stream          = Ran_Stream.Create (stream)
            END;
        Sleep (b: Bank_Model): Bank_Model =
            RECREATE Just_Arrived = FALSE END;
        Make_New_Arrival (b: Bank_Model): Bank_Model =
            IF Customer_Num (b) <= Num_Customers (b) THEN
                RECREATE
                    Just_Arrived = TRUE;
                    Num_In_System = Num_In_System (b) + 1;
                    Customer_Num = Customer_Num (b) + 1;
                    Arrival_Times = Arrival_Times (b) + Time (Clock)
                END
            ELSE b;
        Begin_Service (b: Bank_Model): Bank_Model =
            RECREATE Server_Busy = TRUE END;
        End_Service (b: Bank_Model): Bank_Model =
            RECREATE
                Server_Busy = FALSE;
                Num_In_System = Num_In_System (b) - 1;
                Departure_Times = Departure_Times (b) + Time (Clock)
            END;
END Bank_Model;
```

Figure 8: Activity-Scanning Based Bank Simulation.

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 the successively more complex uses of Active KDL by presenting increasingly more complex examples of queries. As we do this we also build up the query language starting with simple queries (see [58] for a complete specification of the query language). To capture the semantics of the constructs in the query language, we present both the intensional and extensional aspects of the queries. 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. The use of the FOR construct in conjunction with set operators allows very general and powerful queries to be formulated.

5.1. 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. The functions to be projected upon are specified in the APPLY clause within the FOR construct.

Query: FOR ALL t IN T APPLY $f_1(t), \ldots, f_n(t)$ END;

$$Intension(Query) = f_1 \cdots f_n_T$$
$$Extension(Query) = \Pi_{f_1 \cdots f_n_T} val(T)$$

Formally, the new object-type $f_1 \cdots f_n_T$ contains n functions, i.e. $f_1, \ldots, f_n$. The extension of this query is a set of objects with function values of $f_1, \ldots, f_n$ 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. An example of a projection query is the following:

FOR ALL b IN Bank_Model APPLY 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 .

5.2. 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 included in the object-type T are extended (using a logical *and*) to contain the condition $p(t)$. The type p_T is a specialization of T since every object `instance-of` p_T will also be an `member-of` T.

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.

5.2.1. Point Queries

Suppose that an analyst wishes to know the customer throughput and the average time that customers spend in 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.

The query driven simulation environment consists of several programs working together under the direction of a UNIX shell script. The following steps will be carried out by the query driven simulation environment for this query. The Active KDL *query processor* will process and execute the query. By supposition, the answer to the query will be empty. Since the threshold is not achieved, the *parameter generator* will be executed. From the WHERE predicate this program will extract the following parameters:

Mean_Arrival = 4.0, Mean_Service = 3.0

All of the input parameters are listed in the constructor of the Bank_Model class. These parameters and their default values are found in the schema. In this case, the missing values not mentioned in the WHERE predicate are the following:

Stream = 1, Num_Customers = 100

Next the parameter generator builds an appropriate small main program whose main purpose is to declare a Bank_Model object. Its constructor will be initiated with the parameters extracted from the previous phase. The execution of the Bank_Model produces results which are appended to the Bank_Model object class. The results are saved in the database since Bank_Model is a persistent class, as are all the object classes defined in an Active KDL schema.

5.2.2. 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.

5.2.3. 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 in parallel.

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 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 must be transformed to disjunctive normal form.

The following query provides information on banks of questionable profitability. It uses the Prob_Bankruptcy heuristic for model selection. If the query driven simulation system is in information generation mode, then models satisfying the two conditions will be instantiated and executed.

```
FOR ALL b IN Bank_Model
    WHERE Mean_Service (b) IN {6.0, 8.0, 10.0, 12.0, 14.0}
        AND Prob_Bankruptcy (b) > 0.2
    APPLY Mean_Service (b), Throughput (b), Mean_Wait (b)
END;
```

Although such heuristics are complex, in its simplest form Prob_Bankruptcy would look something like the following:

```
Prob_Bankruptcy (b: Bank_Model): REAL =
    SUM (FOR ALL f IN Factor WHERE Has_Factor (f, b) APPLY Weight (f) END);
```

where Factor is another object-type which includes an attribute Weight and a BOOLEAN returning method Has_Factor.

5.3. 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;
```

Model instantiation becomes significantly more complex when multiple models are referenced by the 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 were 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.

5.4. Set Theoretic Queries

Results from simulation sub-studies may be combined using 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 join lattice. The other set theoretic operators are defined similarly (see [58]).

6. 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 strongly supports query driven simulation. A single-user uniprocessor prototype C++ implementation of Active KDL has been completed [29], [30], [31]. We 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. In query driven simulation, each parameter set can be executed in parallel [12]. Parallelism within a model can be achieved using for example the TimeWarp algorithm [69]. Parallel database transactions/applications can use high performance concurrency control protocols [69], [70] and can use multiple versions of objects [69]. Finally, parallelism within query/update processing can be improved by using for example a parallel join algorithm and by exploiting the functional nature of the Active KDL sub-languages.

7. References

[1] Pritsker, A., *Introduction to SLAM II, Third Edition*, John Wiley & Sons, N.Y. (1986).

[2] Standridge, C., and A. Pritsker, *TESS: The Extended Simulation Support System*, Halstead Press, N.Y. (1987).

[3] Birtwistle, G., *DEMOS: A System for Discrete Event Modelling on SIMULA*, Springer-Verlag, N.Y. (1987).

[4] Birtwistle, G., O. Dahl, B. Myhaug, and K. Nygaard, *Simula Begin*, Studentlitertur and Auerbach Publishers (1973).

[5] CACI, *MODSIM II: The Language for Object-Oriented Simulation*, Los Angeles, CA (January 1990).

[6] Fox, M.S., N. Husain, M. McRoberts, and Y.V. Reddy, "Knowledge-Based Simulation: An Artificial Intelli-

gence Approach to System Modeling and Automating the Simulation Life Cycle," in *Artificial Intelligence, Simulation and Modeling*, L.E. Widman, K.A. Loparo, and N.R. Nielsen (Eds.), Wiley Interscience, N.Y. (1989).

[7] Reddy, Y.V., M.S. Fox, N. Husain, and M. McRoberts, "The Knowledge-Based Simulation System," *IEEE Software* (March 1986).

[8] McArthur, D., P. Klahr, and S. Narain, *The ROSS Language Manual*, The RAND Corporation, N-1854-1-AF (September 1985).

[9] Rothenberg, J., "The Nature of Modeling," in *Artificial Intelligence, Simulation and Modeling*, L.E. Widman, K.A. Loparo, and N.R. Nielsen (Eds.), Wiley Interscience, N.Y. (1989).

[10] Zeigler, B.P., "Hierarchical, Modular Discrete Event Modelling in an Object Oriented Environment," *Simulation Journal* 49, 5 (November 1987).

[11] Zhang, G., and B.P. Zeigler, "The System Entity Structure: Knowledge Representation for Simulation Modeling and Design," in *Artificial Intelligence, Simulation and Modeling*, L.E. Widman, K.A. Loparo, and N.R. Nielsen (Eds.), Wiley Interscience, N.Y. (1989).

[12] 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).

[13] Miller, J.A., O.R. Weyrich, Jr., W.D. Potter, and V. Kessler, "The SIMODULA/OBJECTR Query Driven Simulation Support Environment," in *Progress in Simulation*, J. Leonard, and G. Zobrist (Eds.) (1990).

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

[15] 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," *Proceedings of the 21st Annual Pittsburgh Conference on Simulation and Modeling*, Pittsburgh, PA (May 1990).

[16] Miller, J.A., O.R. Weyrich, Jr., and D. Suen, "A Software Engineering Oriented Comparison of Simulation Languages," *Proceedings of the 1988 Eastern Simulation Conferences: Tools for the Simulationists*, Orlando, FL (April 1988).

[17] 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).

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

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

[20] Fishman, D., et al., "Iris: An Object-Oriented Database Management System," *ACM Transactions on Office Information Systems* (January 1987).

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

[22] Maier, D., J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," *OOPSLA '86 Confer-*

ence Proceedings, Portland, OR (September 1986).

[23] 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," To appear in the *Proceedings of the 1991 AI and Simulation International Conference*, New Orleans, LA (April 1991).

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

[25] 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).

[26] Potter, W.D., and R.P. Trueblood, "Traditional, Semantic and Hyper-Semantic Approaches to Data Modeling," *IEEE Computer* (June 1988).

[27] 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).

[28] 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).)

[29] Kessler, V., and W.D. Potter, "New Directions for Object-Oriented Database Managements Systems," *Proceedings of the 27th Annual ACM Southeastern Regional Conference*, Atlanta, GA (April 1989).

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

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

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

[33] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, MA (1986).

[34] Richardson, J., and M. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proceedings of the ACM-SIGMOD 1987 International Conference on the Management of Data*, San Francisco, CA (May 1987).

[35] Hull, R., and R. King, "Semantic Database Modeling: Survey, Applications and Research Issues," *ACM Computing Surveys*, Vol. 19:3 (September 1987).

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

[37] Kerschberg, L., (Ed.), *Expert Database Systems: Proceedings From The First International Workshop*, Kiawah Island, South Carolina (1984). (Published by Benjamin Cummings in 1986)

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

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

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

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

[42] 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).

[43] 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).

[44] 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).

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

[46] Shipman, D., "The Functional Data Model and the Data Language DAPLEX," *ACM Transactions on Database Systems* (March 1981).

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

[48] 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).

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

[50] Risch, T., R. Reboh, P. Hart, and R. Duda, "A Functional Approach to Integrating Database and Expert Systems," *Communications of the ACM*, 31, 12 (December 1988).

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

[52] Goodman, N., "Object Oriented Database," *InfoDB*, 4, 3 (Fall 1989).

[53] Stein J., and D. Maier, "Concepts in Object-Oriented Data Management," *Database Programming and Design* 1, 4 (April 1988).

[54] Backus, J., "Can Programming be Liberated from the von Neumann Style," *Communications of the ACM*, 21, 8 (August 1978).

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

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

[57] 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).

[58] Potter, W.D., K.J. Kochut, J.A. Miller, A.A. Keskin, and E. Ucar, "Intensional and Extensional Effects of Data Modeling and Querying in the KDL Object-Oriented Database System," Submitted to *The ACM-SIGMOD 1991 International Conference on the Management of Data*, Denver, CO (May 1991).

[59] 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).

[60] Elmasri, R., and S. Navathe, *Fundamentals of Database Systems*, The Benjamin Cummings Publishing Co., Redwood City, CA (1989).

[61] 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 (January 1981).

[62] Chen, P., "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems* (March 1976).

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

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

[65] Springer, G., and D.P. Friedman, *Scheme and the Art of Programming*, The MIT Press, Cambridge MA (1989).

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

[67] Brownston, L., R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, Inc., Reading, MA (1985).

[68] Kreutzer, W., *System Simulation Programming Styles and Languages*, Addison-Wesley, Inc., Reading, MA (1986).

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

[70] Griffeth, N.D., and J.A. Miller, "Performance Modeling of Database Recovery Protocols," *Transactions on Software Engineering* 11, 6 (June 1985).