# Model Instantiation for Query Driven Simulation in Active KDL

John A. Miller∗, Walter D. Potter∗†, Krys J. Kochut∗† and Orville R. Weyrich, Jr.†
∗Department of Computer Science
†Artificial Intelligence Program
University of Georgia
Athens, Georgia 30602

## ABSTRACT

The focus of this paper is on how Active KDL can be used to provide a very powerful simulation support environment. Active KDL (Knowledge/Data Language) is an object-oriented database programming language, which provides access to integrated *model, knowledge,* and *data bases*. 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 procedural behavior. Finally, Active KDL provides a simple mechanism for specifying concurrent execution, namely tasks embedded in active objects. These facilities provide a powerful mechanism for building simulation models out of pre-existing model components. These capabilities provide a tight coupling between a SIMODULA-like simulation system and a knowledge/database system, supporting query driven simulation, where model instantiation is used for information generation.

## 1. INTRODUCTION

Query driven simulation is a powerful paradigm for simulation modeling and analysis. Simulation modeling 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 [Stan 87, Prit 86]) have been of considerable help in this area. Our ongoing research is an attempt to provide a very simple simulation support environment. Our goal is to have the user view our environment as an information system. Our environment has information about or is able to generate information about the behavior of systems the user wishes to study. For the most part, users interact with the system simply by formulating queries.

Our first approach to providing a query driven simulation support environment could be described as loosely coupled. This approach coupled a process-oriented simulation language, called SIMODULA, with a simple single-user relational database system, called OBJECTR [Mill 88, Mill 89, Mill 90]. 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 paradigm. The heart of the system is an *object-oriented database programming language* called Active KDL (Knowledge/Data Language). [Much of the current research and development in the database field involves object-oriented database systems [Beec 88, Bloo 87, Fish 87, Kim 87, Kim 88, Maie 86, Merr 87, Penn 87, Smit 87], 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*. 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 procedural behavior. Finally, Active KDL provides a simple mechanism for specifying concurrent execution, namely tasks embedded in active objects. 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, which is a hyper-semantic data modeling language developed by Potter [Pott 86, Pott 87, Pott 88a, Pott 88b, Pott 89]. Recently, a partial implementation of KDL in C++ has been completed [Kess 89]. This implementation exploits the already present data (or object) modeling capabilities of C++ [Duwh 89, Lipp 89, Berr 88, Wein 88, Stro 86]. KDL schema and queries are translated (via yacc) to C++, at which point they are compiled and executed. Currently, several research projects are either extending C++ to make it a database programming language (e.g., O++ [Agra 89], E [Rich 87]) or, as we are doing, translating a simpler and higher level language to C++ (e.g., ONTOS (new version of Vbase) [Andr 89, Duhl 88, Andr 87]).

Active KDL includes methods and object activity (in the form of tasks) to the current facilities provided by KDL. An object may have a task embedded in it. These tasks can run concurrently. In our prototype of Active KDL, we are implementing a limited form of concurrency. Because of its importance in *process-interaction* type simulations, we are using *coroutines* as the primitive mechanism for providing concurrency (or more precisely quasi-concurrency). Coroutine primitives are provided by Simula 67 [Birt 73] and Modula-2 [Shar 88, Hugh 87, Wirt 85], both of which are general-purpose programming languages that have been used quite successfully for simulation modeling. Additionally, Simula 67 is credited with the honor of being the first object-oriented programming language. We are using coroutines to facilitate process-interactions that are very similar to those provided in SIMODULA.

In a future enhancement to Active KDL, we plan to provide a more powerful form of concurrency (e.g., Concurrent C's synchronous and asynchronous message passing). With the generality that this provides, we hope to specify the synchronization and concurrency control of tasks in a general fashion using synchronization or temporal constraints [Toml 89].

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 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, and even models involving recursion.

Lastly, we discuss how Active KDL can be used for model derivation. Since modeling elements can be stored in a model base, a query can return modeling elements. This powerful facility, gives users the capability to browse a model base, seeing what is available. If the appropriate modeling elements have already been coded, then the user can very rapidly create his own simulation models, simply by deriving new models from pre-existing model components. This is done using the class derivation capabilities of Active KDL and its translation target language C++ (both support single and multiple inheritance).

## 2. ADVANTAGES OF HYPER-SEMANTIC DATA MODELS

The foundation of Active KDL, the Knowledge/Data Language, is based on the Knowledge/Data Model (KDM). The KDM is a member of a newly defined class of models called *Hyper-Semantic Data Models* [Pott 86, Pott 88, Pott 89]. (These more powerful models have recently been referred to as Fifth Generation Object-Oriented Knowledge/Data Models by Professor Larry Kerschberg in his tutorial on Expert Database Systems at the Sixth International Conference on Data Engineering, 1990.) The KDM subsumes the currently popular features of object-oriented data models by providing features to handle objects, object classes, procedural attachment (methods), and various abstraction mechanisms. In addition, the KDM handles heuristic attachment and basic temporal primitives. Extending and exploiting all of these features are the goals of Active KDL.

In general, hyper-semantic data models focus on capturing the *objects*, *operations*, *relationships*, and *knowledge* associated with an application. Such data models address the notion of capturing *even more of the meaning* (i.e., over and above) of an application than allowed with semantic data models. In the middle 1980's Potter and Kerschberg developed the KDM by combining the modeling capabilities of the Functional Data Model (FDM) [Kers 76, Ship 81] with knowledge representation and knowledge base system capabilities from the field of artificial intelligence. The KDM expands on the ideas found in semantic data models and incorporates knowledge about an application. The KDM allows more information to be associated with an application, than allowed by semantic models, by adding heuristics, more flexible constraints, and temporal relationships to a semantic data model foundation.

In the KDM, objects are the things of interest in the real world application which are to be modeled. Operations are the things that can be done to any object at either the application level or the meta-level. Relationships represent the ways that various objects are associated with one another. Knowledge, in the form of heuristics or rules, is information about objects, operations, and relationships that may be retrieved or inferred when needed. This information is not only maintained for actual instances of objects but also for object classes as well. That is, meta-level knowledge is included in the KDM. Operations on object classes themselves may have knowledge associated with them.

Heuristics are special relationships that allow information about an object to be inferred rather than using a traditional query against stored data. Thus, heuristics provide data derivation mechanisms. They apply to both objects and object classes, and help define the properties of objects and the membership of object classes, respectively.

In addition to knowledge in the form of heuristics, knowledge in the form of constraints may be utilized. Constraints on objects restrict the domain for an object class, so it would not be possible to add an object that violated any of the constraints of its associated object class. Constraints on operations provide a means to control their effects. For example, operations that would create invalid results are not allowed. Constraints on relationships are used to control the number of objects that can be related via some relationship. Also, attribute values are constrained to maintain object integrity.

The ancestoral lineage of the KDM relates back to the FDM, one of the predominant semantic data models. Semantic data models [Chen 76, Hamm 78, Elma 89] are generally characterized as models that support (in an object oriented fashion) the notions of classification, generalization, aggregation, and (sometimes) membership [Hull 87, Smith 77]. The KDM supports these primitive modeling notions as well as three other primitives. Specifically, there are seven constructs associated with hyper-semantic data models (and the KDM) that are used to model real world applications. These include: classification, generalization, aggregation, membership, constraint, heuristic and temporal constructs.

Generalization is when lower level object classes are abstracted into a higher level object class via the *is-a* relationship. The inverse of this relationship is called specialization. The lower level object classes inherit all properties and methods associated with the higher level classes, just as in object-oriented theory. For example, a student class can be created from a more general person class. In this case, the student class *is-a* person class. Classification is a special case of generalization where the objects of a class are database instances. For example, 'Bob' and 'George' are classified as being in the person class.

Aggregation is when an object is composed of other objects that constitute or define the object. These objects are related via the *is-part-of* relationship. All user defined object classes are formed using aggregation of primitive types. For example, name and address are part of any object in the person class.

Membership is where several object classes are considered as a higher level set object class via the *is-a-member-of* relationship. This specifically allows an object class to hold objects of different classes. For example, an object class "hospital committees" could contain objects from the doctor class, the nurse class, and the administrators class. That is, a hospital committee (an object) is made up of three members: a doctor, a nurse, and a hospital administrator.

Constraints are specified by the *is-constraint-on* relationship. Constraints allow the legal values for an attribute (or property) to be limited in some manner. Constraints may also be placed on object classes which limit the objects that could be placed in the class. For example, "the value must be a non-negative value" could be a constraint on salary. Also, "there may be no more than 40 courses offered" could be a constraint on the courses object class.

Heuristics typically allow attribute values of an object to be determined, via an associated inference procedure. These heuristics are related to attributes, objects, and object classes through the *is-heuristic-on* relationship. For example, the courses a student should enroll in next term can be determined using rules relating the courses already taken and the degree requirements associated with the degree program the student is in.

Temporal constructs allow task or procedurally oriented objects to be related to each other by synchronous or asynchronous characteristics. In turn, these objects and relationships may be abstracted into higher level objects or object classes (again, these new objects may be task oriented). The relationships of interest here are: *is-successor-of*, *is-predecessor-of*, and *is-concurrent-to.* For example, a task to calculate the total of an invoice would be declared as being a successor to the task to enter the invoice information. Task objects may be modeled as well as transaction processing. As with heuristics, which depend on an inference process, temporal inferences may be made using event information and some temporal inference process such as Kowalski's Event Calculus [Kowa 86].

## 3. FEATURES OF ACTIVE KDL

Active KDL is the schema specification language and query language for our integrated model, knowledge and data environment. The *schema definition language* is used to define object classes (or object-types) and is declarative in nature. The processing of this declarative information is left to query optimization algorithms. The *query language* is used to formulate queries on the knowledge/database. The query language may also be used to implement the bodies of methods and tasks, as an alternative to implementing them directly in C++.

Active KDL is characterized as a context free language. A partial Extended Backus-Naur Form (EBNF) grammar for the language can be found in the Appendix. The basic notion of Active KDL is similar to the database language DAPLEX which is a semantic data model-based query language for the Functional Data Model. Both languages are strongly functional in character. Active KDL adds object-orientation. In this regard, it is similar to the object-oriented functional query languages: $O^2FQL$ [Mann 89] and OODAPLEX [Daya 89]. Furthermore, Active KDL supports concurrency through the use of active objects.

### 3.1. Semantic Features of Active KDL

The primary construct in Active KDL is the *object class* (or object-type). An object class is a collection of homogeneous objects where objects correspond to real-world concepts or things such as chairs, students, or heuristics (i.e., a heuristic object may be composed of antecedent objects and consequent objects). Object classes encapsulate information about objects in the form of data which may be stored, inferred, or computed.

*Attributes* are used to store data about objects. An attribute corresponds to a particular property or characteristic associated with an object such as the number of legs of a chair, the name of a student, or the condition-set of a heuristic. In keeping with the functional nature of Active KDL, attributes can be viewed as stored functions. Hence as one would expect, attributes are read-only from outside the object. Of course, the methods associated with the object can modify its attributes (read/write access). Active KDL supports a variety of different kinds of attributes. Included among these are the more traditional ones such as the atomic attributes found in the relation data model, multivalued (or set-valued) attributes which are common in semantic data models, and attributes that explicitly reference an object defined in another object class, which are common in object-oriented data models. Heuristics and methods can be used to effectively provide *inferred* and *computed* attributes (more about this later).

Other important features of semantic data model languages supported by Active KDL include: 1) the specification of generalization and specialization hierarchies, 2) the classification and instantiation of object class instances, 3) the aggregation and refinement of object class characteristics, and 4) the membership (or association) specification facility.

Specialization hierarchies are defined by specifying super class (or supertype) relationships within the schema in order to reflect the real world relationships. Object classes inherit the attributes and other capabilities of their supertypes. Classification and instantiation of object class instances provide the primary mechanism for identifying and handling token (or instance) level objects. Also, class level objects are identified and handled using these primitive constructs. The definition of object class attributes and the technique for dealing with attribute abstraction are provided by the aggregation and refinement constructs of Active KDL. Active KDL supports both ordinary aggregation by defining object classes and object-level aggregation by the multiple inheritance mechanism. The membership primitive is provided in order to model information/knowledge system situations dealing with concepts that are composed of elements having different characteristics (such as a committee of diverse members).

Active KDL supports: 1) the modeling primitives defined in the Knowledge/Data Model, 2) a collection of built-in operations such as those found in the programming language Pascal and 3) built-in functions (e.g., aggregation functions in DAPLEX) such as Sum, Count, and Average. An important goal of Active KDL is the specification of knowledge/data system schemas and a query facility for data, meta-data, knowledge, and meta-knowledge. The flexibility of Active KDL is exemplified by its "self-referential" feature which permits a meta-description of the language to be "written" using the language itself. This *meta* feature is essential for an integrated model, knowledge and data base language [Kers 86a, Kers 86b].

The basic Active KDL object class specification structure, called an object-type, is shown in Figure 1. The syntax is specified using Extended Backus-Naur Form (EBNF), where brackets, [ ], represent optional items and braces, { }, identify items that may be repeated zero or more times; a vertical bar, |, indicates an alternative. The upper-case words are reserved words in Active KDL and lower-case words represent information that is defined or specified by the schema designer. Note, as in C++, Active KDL comments are identified by the string // or the string pair /* */.

```
OBJECT_TYPE class-name HAS
    [ SUPERTYPES:
        { class-name { , class-name } ; }
    [ ATTRIBUTES:
        { attribute-name: [ SET OF | LIST OF ] type
            [ INVERSE OF attribute-name [ (class-name) ] ] ;
            [ WITH CONSTRAINTS { constraint; } ] ] }
    [ HEURISTICS:
        { rule; } ]
    [ CONSTRAINTS:
        { constraint; } ]
    [ METHODS:
        { method; } ]
    [ MEMBERS:
        { member-name: class-name; } ]
END class-name;
```

**Figure 1. Active KDL Object-Type Specification.**

Each object class specification begins with the definition of the name of a class in the OBJECT_TYPE clause. The optional characteristics of an object class correspond to the KDM modeling primitives while the class-name uniquely identifies the object class. An object class has zero or more attributes that are defined in the ATTRIBUTES clause. The example below shows a schema for a bank employee database.

```
SCHEMA BankPersonnel;

OBJECT_TYPE Person HAS
        SUPERTYPES:
                Object;
        ATTRIBUTES:
                SSN:            INTEGER;
                Name:           STRING;
                Sex:            CHAR;
                BirthDate:      STRING;
                City:           STRING;
                Parents:        SET OF Person;
                Children:       SET OF Person INVERSE OF Parents;
        HEURISTICS:                                             // Inferred Function or Rule
                Father(p: Person): Person IS
                        FOR SOME q IN Parents(p) WHERE Sex(q) = "M" APPLY q END;
                END Father;
                Siblings(p: Person): SET OF Person IS
                        FOR ALL c IN Children(Parents(p)) WHERE c <> p APPLY c END;
                END Sibling;
                Cousins(p: Person): SET OF Person IS
                        Children(Siblings(Parents(p)));
                        FOR ALL c IN Cousins(Parents(p)) APPLY Children(c) END;
                END Cousins;
                Ancestors(p: Person): SET OF Person IS
                        Parents(p);
                        FOR ALL a IN Ancestors(p) APPLY Parents(a) END;
                END Ancestors;
        METHODS:                                        // Implemented in C++
                Age(): INTEGER;                 // Made a method since it’s complex
                NameChange(NewName: STRING);
END Person;

OBJECT_TYPE Employee HAS
        SUPERTYPES:
                Person;
        ATTRIBUTES:
                Salary:         REAL;
                Office:         STRING;
                WorksIn:        Department;
                Supervisor:     Employee;
        HEURISTICS:
                Superiors(e: Employee): SET OF Employee IS
                        Supervisor(e);
                        FOR ALL s IN Superiors(e) APPLY Supervisor(s) END;
                END Superiors;
        CONSTRAINTS:
                SalaryCheck(e: Employee) IS
                        Salary(e) < Salary(Supervisor(e));
END Employee;

OBJECT_TYPE Department HAS
        SUPERTYPES:
                Object;
        ATTRIBUTES:
                DeptNo:         INTEGER;
                Name:           STRING;
                City:           STRING;
                Staff:          SET OF Employee INVERSE OF WorksIn(Employee);
                Manager:        Employee;
END Department;

END BankPersonnel;
```

**Figure 2. Schema for Bank Personnel.**

### 3.2. Query Language for Active KDL

Queries in Active KDL are simply function applications on either objects or sets of objects. Complex queries are handled by iterating over sets of objects, applying functions to the iteration variable. Indeed, our iteration construct allows the iteration over multiple variables. The syntax for the iteration construct is given below:

```
FOR [ ALL | SOME ] variable IN simple-query { , variable IN simple-query }
    WHERE predicate APPLY { query; } END
```

Some example queries are given below.

```
Name(Employee);

FOR ALL e IN Employee WHERE DeptNo(WorksIn(e)) = 10 APPLY Name(e) END;

FOR ALL e IN Employee, d IN Department WHERE City(e) = City(d) APPLY
     Name(e), Name(d);
END;
```

The first query is simply a function application of the attribute "Name" to the object class "Employee" and would return the names of all employees. The second query is an example of *navigational processing* which is quite common in object-oriented database systems. Finally, the last query is an example of *associative processing* which is the cornerstone of relational query languages. Note, a query with dual iteration variables associated by a condition involves an implicit join. Although some early object-oriented database systems excluded this capability (it is not as fast as navigational processing), more recent systems provide it since it gives users the ability to ask totally unanticipated queries.

### 3.3. Hyper-Semantic Features of Active KDL

Let us at this point elaborate on the heuristic enhancement that KDM provides over semantic data models. *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. A particular heuristic for an object class is manifested as a set of *rules* written in a functional notation. Rules are expressed as parameterized queries and they are typically recursive. The rules are a set rather than a sequence; the rule firing order is determined by our query optimization algorithms, which are currently under development [Kesk 90]. Heuristics give KDL fundamentally more expressive power than a relationally complete language like SQL. The example below illustrates the power of heuristics. Let us suppose our BankSimulation model (see Figure 3) includes information about bank employees. In particular, we would like to know the chain of command above any given employee. The inferred or virtual attribute Superiors, which is defined by a set of rules, gives us this information. In order to determine the chain of command above a given employee, we specify the following query:

```
FOR SOME e IN Employee WHERE Name(e) = "Barney Rubble" APPLY
     FOR ALL s IN Superiors(e) APPLY
          Name(s), SSN(s);
     END;
END;
```

Closely related to rules are methods. The difference is that rules are purely functional (no side effects) and declarative, while methods are basically just general procedural code (although rules can be embedded in methods). Methods are written directly in C++, only their signatures are included in the schema. [Note, the method signatures in the schema will be identical to those of their C++ implementations, except for the way in which individual parameters are declared. For example, NewName: STRING in the schema is translated to STRING _NewName in the C++ implementation.] Furthermore, the rules (which are specified inline) are implemented via sophisticated query optimization/inference algorithms (e.g., variants of forward and backward chaining).

Our rules and methods have direct analogues in POSTQUEL (the query language for POSTGRES [Ston 86]). 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.

In Active KDL, constraints can be imposed in very general ways. Simple constraints can be attached to an attribute to ensure that it takes on only appropriate values. Complex constraints can be used to ensure that new objects inserted into an object class are compatible with current objects in this or any referenced class. Finally, we plan to investigate whether proof-theoretic verification techniques can be applied to method implementations, to ensure that they are compatible with existing constraints.

As mentioned in the introduction, our prototype of Active KDL provides a limited form of concurrency, namely coroutines. The reason for choosing coroutines, is that they are simple, efficient, and the minimal primitive necessary to support process-interaction type simulations. The coroutine capability is made available through the *task class* [Duwh 89] which is provided in the library that comes with the *AT&T C++ Language System, Release 2.0.* Any class derived from the task class will have concurrent capabilities, and is said to be an active class, whose instances will be *active objects*. 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 explicitly executes a statement that transfers control away from it (see Figure 4 in the next section). When control is transferred back to this coroutine is resumes where it left off. Note, a class not derived from the task class is said to be an ordinary passive class, whose instances will be *passive objects*.

## 4. SIMULATION SCHEMA DESIGN USING ACTIVE KDL

The data and models used in simulation analysis are stored in the Active KDL object-oriented database system. To illustrate its capabilities, we define a schema for the ubiquitous bank simulation. In this simple simulation problem, customers arrive (Poisson process) and enter the bank. They proceed to any one of s tellers for service (exponentially distributed). If no teller is free, customers line up in a common queue. After service is complete, customers depart with their waiting time and system time recorded. Note, this model is simply a M/M/s queuing system [Bank 84, Law 82].

An Active KDL schema is used to define the object classes (or types) that make up an information/simulation application. Here, we are using the term object class to denote both a formal data type and the current collection of objects of the type (often referred to as a type extent). The BankSimulation Schema shown in Figure 3 defines Teller, Customer, and BankModel object classes. These object classes inherit from and reference the following existing schemas: the BankPersonnel Schema which is a conventional data processing type schema, and the Simulation Schema which defines facilities for carrying out simulations. The Simulation Schema, although still evolving, is intended to provide the capabilities and style of SIMODULA (and thus to some extent SIMSCRIPT II.5 [CACI 87, Russ 83]). A portion of the Simulation Schema is shown in Figure 4.

```
SCHEMA BankSimulation;

OBJECT_TYPE Tellers HAS
    SUPERTYPES:
        Resource;                   // A Resource is-an Object (passive)
    ATTRIBUTES:
        NumTellers:  INTEGER;
        Crew:        SET OF Employee;
                     WITH CONSTRAINT SizeCheck(t: Tellers) IS
                         Cardinality(Crew(t)) = NumTellers;
END Tellers;

OBJECT_TYPE Customer HAS
    SUPERTYPES:
        SimObject, Person;          // A SimObject is-a task (active)
    ATTRIBUTES:
        AccountNum: INTEGER;
        Balance:     REAL;
        InterestRate: REAL;
        ArrivalTime:  REAL;
        ServiceTime:  REAL;
        WaitingTime:  REAL;
        SystemTime:   REAL;
    HEURISTICS:
        AnnualYield(c: Customer): REAL IS
            Prod(Balance(c), InterestRate(c));
        END AnnualYield;
    METHODS:
        Customer(Bank: BankModel);
            // Constructor -- Script for a typical customer
END Customer;

OBJECT_TYPE BankModel HAS
    SUPERTYPES:
        Model, Tellers;                     // A Model is-an Instantiatable SimObject
    ATTRIBUTES:
        Stream:      INTEGER;               // Stream number
        RanStream:   Variate;               // Random variate stream
        NumCustomers: INTEGER;              // Number of customers
        MeanIArrival: REAL;                 // Mean interarrival time
        MeanService:  REAL;                 // Mean service time
        StopTime:    REAL;                  // Simulation Stop Time
        Customers:   SET OF Customer;       // Customers served by the bank
    HEURISTICS:
        MeanWaitingTime(b: BankModel): REAL IS
            Average(WaitingTime(Customers(b)))
        END MeanWaitingTime;
        Throughput(b: BankModel): REAL IS
            Div(Count(Customers(b)), StopTime(b));
        END Throughput;
    METHODS:
        BankModel(Stream:          INTEGER = 1,
                  NumTellers:      INTEGER = 2,
                  NumCustomers: INTEGER = 100,
                  MeanIArrival: REAL = 8.0,
                  MeanService: REAL = 10.0);
            // Constructor -- Script for a bank model
END BankModel;

END BankSimulation;
```

**Figure 3. Schema for Bank Simulation.**

```
SCHEMA Simulation;

OBJECT_TYPE SimObject HAS
        SUPERTYPES:
                Object, task;                // Object => persistence, task => concurrency
        ATTRIBUTES:                                    // Attrs are stored functions (read-only)
                ActivationTime: REAL;
                Priority:       INTEGER;
                Status:         STRING;
                        WITH CONSTRAINT StatusCheck(s: SimObject) IS Status(s) IN
                                {"PASSIVE", "ACTIVE", "SUPERACTIVE", "INTERRUPTED"};
                FutureActivities: Agenda;
        METHODS:
                SimObject(Priority: INTEGER = 10);
                        // Constructor creates a SimObject & initiates coroutine execution
                Activate(s: SimObject; timeDelay: REAL);
                        // Activates s after a timeDelay
                Work(timeDelay: REAL);
                        // Delay for timeDelay units of time
                Suspend();
                        // Suspend until another object activates you
                Interrupt(s: SimObject);
                        // Interrupts the working SimObject s
                Resume(s: SimObject);
                        // Resumes the previously interrupted SimObject s
                Time(): REAL;
                        // Return the current simulated time
END SimObject;

OBJECT_TYPE Agenda HAS
        SUPERTYPES:
                Object;
        ATTRIBUTES:
                Clock:          REAL;
                ActivityList: LIST OF SimObject;
                        WITH CONSTRAINT Order(s: SimObject) IS
                                ActivationTime(s) < ActivationTime(Succ(s)) OR
                                ActivationTime(s) = ActivationTime(Succ(s)) AND
                                        Priority(s) < Priority(Succ(s));
        METHODS:
                Insert(s: SimObject);
                Remove(): SimObject;
END Agenda;

OBJECT_TYPE Model HAS SUPERTYPES: SimObject; ... END Model;

OBJECT_TYPE Resource HAS SUPERTYPES: Object; ... END Resource;

OBJECT_TYPE Variate HAS SUPERTYPES: Object; ... END Variate;

END Simulation;
```

**Figure 4. Schema for Simulation.**

```
// banksim.c - method/task implementations for BankSimulation

Customer::Customer(BankModel* Bank) :
        /* Initialize Customer object */ {
    /*************************************************************************/
    ArrivalTime = Time();                    // Record arrival time
    Bank->Request(1);                        // Request 1 bank teller
    WaitingTime = Time() - ArrivalTime;      // Record customer waiting time
    ServiceTime = Bank->RanStream.Exponential(Bank->MeanService);
    Work(ServiceTime);                       // Delay this coroutine
    Bank->Relinquish(1);                     // Release the bank teller
    SystemTime = Time() - ArrivalTime;       // Record time in the system
}; // Customer

BankModel::BankModel(int _Stream, int _NumTellers, int _NumCustomers,
            double _MeanIArrival, double _MeanService) :
        /* Initialize BankModel object */ {
    /*************************************************************************/
    for (int i = 1; i <= NumCustomers; i++) {
        Work(RanStream.Exponential(MeanIArrival));
        Customers += new Customer(this);    // Create & initiate new coroutine
    }; // for
    StopTime = Time();
}; // BankModel
```

**Figure 5. Implementation for Bank Simulation.**

## 5. MODEL INSTANTIATION

Query driven simulation can operate in two modes. In the simpliest mode, the query processor will simply return information generated from previous simulation runs. The results of which were stored in the database. The real power of query driven simulation comes into play when the system is in *information generation mode*. In this case, when the answer to a query is judged to be inadequate, the system will automatically generate additional information that is both stored in the database and returned as part of the answer to the query. This process is called model instantiation. Depending on the complexity of the query this can be a simple or a quite complex process. The process centers around the creation of sets of input parameter values which are obtained by schema and query analysis. Until an adequate amount of information is generated, a set of parameter values is established for a model (or models) which is then executed to produce results.

We are currently considering various adequacy criteria. A simple approach is to provide a user settable threshold. If for example, 75% of the full answer is stored in the database, then information generation will not be necessary. However, falling below this threshold will result in simulation runs being executed to meet the 75% requirement. Mechanisms such as this are necessary, since naive users may submit queries that would require an enormous amount of computation to provide the full answer. Let us illustrate how query driven simulation and model instantiation work by giving successively more complex examples.

### 5.1. Point Queries

Suppose that an analyst wishes to know the customer throughput and the average time that customers spend in waiting for a 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 6 minutes, the analyst could simply formulate the following query:

```
FOR ALL b IN BankModel
    WHERE MeanIArrival(b) = 4.0 AND MeanService(b) = 6.0
    APPLY Throughput(b), MeanWaitingTime(b);
END;
```

This is a point query since it 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 the system is in that mode). Information generation for point queries is quite simple. Model input parameters are assigned values extracted from the query, specifically the WHERE predicate (e.g., MeanService = 6.0). Input parameters not mentioned in the WHERE predicate are set to their default values (e.g., NumTellers = 2). 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 query will be empty. Since the environment is in information generation mode, the *parameter generator* will be executed. From the WHERE predicate this program will extract the following parameters:

MeanIArrival = 4.0, MeanService = 6.0

All of the input parameters are listed in the constructor of the BankModel 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, NumTellers = 2, NumCustomers = 100

Then the parameter generator builds an appropriate small main program whose main purpose is to declare a BankModel object. Its constructor, BankModel::BankModel, will be initiated with the parameters extracted from the previous phase. Recall, as discussed earlier, that invoking a constructor for a class derived from the task class, causes that constructor to be executed as a coroutine. The execution of the BankModel produces results which are appended to the BankModel object class. The results are saved in the database since BankModel is a persistent class, as are all the object classes defined in an Active KDL schema.

### 5.2. 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 as more tellers are hired. The following query, in which the NumTellers input parameter is varied from 2 to 6, provides the answer.

```
FOR ALL b IN BankModel
    WHERE MeanIArrival(b) = 4.0 AND MeanService(b) = 6.0
                    AND NumTellers(b) IN {2..6}
    APPLY NumTellers(b), Throughput(b), MeanWaitingTime(b);
END;
```

This is not a point query since multiple input parameter sets will be needed, one parameter set for each value of NumTellers (2 to 6 inclusive). The BankModel will be instantiated (declared, constructed and executed) once for each of the 5 input parameter sets.

### 5.3. 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 whether it is better to hire one fast highly skilled teller, or two unskilled tellers which are half as fast. Assuming the cost of the first alternative is roughly equivalent to the second, the issue could be decided on the basis of which alternative produces the least customer waiting time. The query below, which is in disjunctive normal form, will provide the answer.

```
FOR ALL b IN BankModel
    WHERE MeanIArrival(b) = 4.0 AND MeanService(b) = 6.0 AND NumTellers(b) = 2
        OR MeanIArrival(b) = 4.0 AND MeanService(b) = 3.0 AND NumTellers(b) = 1
    APPLY NumTellers(b), Throughput(b), MeanWaitingTime(b);
END;
```

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

```
PS #1: MeanIArrival = 4.0, MeanService = 6.0, NumTellers = 2, "defaults"
PS #2: MeanIArrival = 4.0, MeanService = 3.0, NumTellers = 1, "defaults"
```

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

### 5.4. Queries Involving Joins

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 BankModel, s IN StoreModel
    WHERE MeanIArrival(b) = MeanIArrival(s) AND
        MeanIArrival(b) IN {3.0, 3.5, 4.0, 5.0} AND
        MeanService(b) = 3.0 AND NumTellers(b) = 2 AND
        ShoppingTime(s) = 2.0 AND CashierService(s) = 1.0
    APPLY MeanIArrival(b), "Bank", MeanWaitingTime(b),
        "Store", MeanWaitingTime(s);
END;
```

In this query the common attribute is MeanIArrival. 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 MeanIArrival to its default value. In the case that each model defined the default value to be different, both default values would be used.

### 5.5. Recursive Queries

Active KDL permits recursive queries to be executed. Suppose an analyst wishes to know the salaries of all the superiors of the bank tellers. This type of query is beyond the capability of relationally complete query languages like SQL in that it requires some form of recursion or iteration. In the query below, recursion comes into play since Superiors is defined by a recursive rule.

```
FOR ALL b IN BankModel WHERE MeanIArrival(b) = 4.0 APPLY
        Salary(Superiors(Crew(b)))
END;
```

In this case the recursion does not involve model instantiation. The information is inferred from existing data, and not generated by a simulation model. Note, data about employees and their supervisors is not generated by our simple BankModel. This data could be in the database due to some other simulation model or indeed conventional database inserts. However, if an analyst needs to simulate the evolution of the hierarchical structure of an enterprise, he/she could enhance the BankModel to do this. This results in a combination of inferential and model-based information generation.

### 5.6. Model Derivation

Since Active KDL and its translation target language are object-oriented (allowing both single and multiple inheritance), deriving new models from previously existing models is relatively easy. For example, suppose that an analyst wishes to construct a new bank model called NewBankModel. This model includes everything in the previous model plus loan officers. Just as Tellers were defined as a Resource, so are LoanOfficers. Additionally, a new type of customer requesting loans, and seeing both a teller and a loan officer will be needed. The class LoanCustomer derived from Customer can be used for this.

## 6. IMPLEMENTATION ISSUES

### 6.1. Model Instantiation Algorithms

As explained in Section 5, in response to a query the query processor will either output information stored in the database (information which was generated during previous simulations), or, if the amount of information is insufficient, it will automatically generate additional simulation results. This additional information is stored in the database for answering future queries. It is also used in answering the query that triggered its generation.

A special program, the *parameter/constructor generator*, implements the model instantiation. The program accepts a given query, and after analyzing it outputs a C++ source program which will run necessary simulations. This generated C++ program contains a number of *object construction calls*. The entire simulation is then triggered by the constructors. The database will also be updated to include the newly generated results. Model instantiation can be applied to any Active KDL object class which is derived from the *Model class*, which is itself derived from the SimObject class. Constructors (e.g., BankModel::BankModel) for classes derived from the Model class are set up to be parameterized by values found in queries.

In general, a query may be a series of nested 'FOR' queries, each of which may include a complex logical combination of simple queries (function calls -- see the formal syntax in Fig. 8). We use a yacc-generated parser in order to translate a query into a simulation triggering program. The translation program consults the database schema in order to determine the calling forms of object constructors. The predicate part of a query is converted into disjunctive normal form, where each conjunction determines a set of model parameter settings for one or more object constructor calls. Assuming only one model is involved in the 'FOR' query, in case the condition in the query being processed is equivalent to a single conjunction of simple queries, there is going to be a single constructor call in the resulting C++ program (it corresponds to a single set of model parameters). On the other hand, if the condition is more complex (a disjunction of conjuncts) the resulting program will contain a number of constructor calls. For example, the query from Section 5.4 results in the following model parameter sets:

(b.MeanArrival = 3.0 AND s.MeanArrival = 3.0 AND b.MeanService = 3.0
AND b.NumTellers = 2 AND s.ShoppingTime = 2.0 AND s.CashierService =
1.0 AND s.Stream = 1 AND s.NumCustomers = 100) OR
(b.MeanArrival = 3.5 AND s.MeanArrival = 3.5 AND b.MeanService = 3.0
AND b.NumTellers = 2 AND s.ShoppingTime = 2.0 AND s.CashierService =
1.0 AND s.Stream = 1 AND s.NumCustomers = 100) OR
(b.MeanArrival = 4.0 AND s.MeanArrival = 4.0 AND b.MeanService = 3.0
AND b.NumTellers = 2 AND s.ShoppingTime = 2.0 AND s.CashierService =
1.0 AND s.Stream = 1 AND s.NumCustomers = 100) OR
(b.MeanArrival = 5.0 AND s.MeanArrival = 5.0 AND b.MeanService = 3.0
AND b.NumTellers = 2 AND s.ShoppingTime = 2.0 AND s.CashierService =
1.0 AND s.Stream = 1 AND s.NumCustomers = 100)

The values for s.Stream and s.NumCustomers were extracted from the default values given in the BankModel schema definition. The parameter generator program generates a random ordering of sets of parameters in order to obtain a "good" coverage. Figure 7 shows the C++ program created in response to the query from Section 5.4. It contains all the necessary constructor calls to achieve the threshold of 100%. In general, however, it may be the case that the initial processing of the query did not produce the desired quality of response because only a small portion of the necessary information was missing. In order to avoid duplication of constructor calls, we keep a history of previous constructor calls, and the resulting C++ program may contain only a subset of the full set of constructor calls (for a given query).

The overall process of answering a simulation query is managed by a simple UNIX shell script, shown in Figure 6, which

1. runs the query processor and examines the results
2. if the results are non-existent, or insufficient, the model triggering C++ program is created, compiled, and executed, after which the query processor is executed again.

```
#!/bin/csh -f
#
#  parameters: $1 - query, $2 - schema, $3 - model
#
RunQuery $1 $2 >! result
CheckResult $1 $2 result >! answer
if (-z answer) then
        GenConstr $1 $2 > NewConstr.c
        CC NewConstr.c $3
        NewConstr
        RunQuery $1 $2 >! result
endif
Display result
```

**Figure 6. Model Instantiation UNIX shell script.**

```
// NewConstr.c -- Information generation program for the query

#include "bank.h"
#include "store.h"

main() {
        BankModel *bm;
        StoreModel *sm;

        // Generate new BankModel and StoreModel results
        // by creating persistent objects.
        // (Note the random ordering of model parameters)

        bm = new BankModel(1, 2, 100, 4.0, 3.0);
        sm = new StoreModel(1, 100, 4.0, 2.0, 1.0);

        bm = new BankModel(1, 2, 100, 3.0, 3.0);
        sm = new StoreModel(1, 100, 3.0, 2.0, 1.0);

        bm = new BankModel(1, 2, 100, 3.5, 3.0);
        sm = new StoreModel(1, 100, 3.5, 2.0, 1.0);

        bm = new BankModel(1, 2, 100, 5.0, 3.0);
        sm = new StoreModel(1, 100, 5.0, 2.0, 1.0);
} // main
```

**Figure 7. Example Generated Driver Program.**

### 6.2. Implementation of Active KDL

Details of the implementation of Active KDL may be found in [Pott 90a, Pott 90b]. In this paper, we give an overview and discuss how queries are processed. The implementation of Active KDL consists of the following modules:

1. Query Processor/Optimizer
2. Schema Processor
3. Inference Engine
4. Constraint Checker
5. Persistent Object Module
6. Concurrency and Recovery Module

Active KDL queries are translated into C++ code, which is then executed. The first step in converting a query to a C++ implementation is the determination of the object classes involved in the query. This is done by checking the FOR constructs, and any function arguments or function calls referencing objects. Information on the attributes each object class uses in the query is also collected. Once the object classes and attributes involved have been determined, the next step is to include C++ class definitions for them. These C++ class definitions are built at schema processing time. They, along with other code, are pre-compiled and only need to be linked to the object code generated by the query. For our example query given below, the object class definition for Employee, "Employee.h", is included.

After the classes have been included, the actual query statements are translated (via yacc) to C++ code. This is a fairly straight-forward procedure. Each referenced object class produces a forall (or forsome) macro statement to iterate over its object instances. If the for construct contains a WHERE predicate, then typically it will be converted to a C++ if statement. Alternatively, if the condition in the WHERE predicate matches some existing index, then the forall can simply iterate over the objects returned by indexing. The decision of exactly how to translate a query is made by our query optimization algorithms. The "function" calls (attribute applications) require some manipulation, however. The way Active KDL specifies function calls to an object is the opposite way C++ does it. So, the nesting of function calls needs to be reversed.

Let us consider how to translate the second query that we presented in an earlier section.

```
FOR ALL e IN Employee WHERE DeptNo(WorksIn(e)) = 10 APPLY Name(e) END;
```

The query above would be converted to the following C++ code if no relevant index exists:

```
#include "Employee.h"

query() {
        forall (Employee e, Employee) {             // iteration macro
                if (e.WorksIn$DeptNo == 10) {
                        cout << e.Name;
                } // if
        } // for
} // query
```

[Note, "$" indicates a logical dereference of a reference to an object in another object class. The reference is an "oid" whose mapping to object location is a physical layer concern.] However, if there exists an index on DeptNo for the Department object class, then a more efficient code segment can be generated.

```
#include "Employee.h"

query() {
        forall (Department d, DeptNoIndex, 10) {
                forall (Employee e, d.Staff) {
                        cout << e.Name;
                } // if
        } // for
} // query
```

The final step of processing a query is the execution of the C++ code. This consists of compiling the C++ code just generated and linking it with the pre-compiled object class and other miscellaneous files. After the executable file is created, it is then run from a shell created by the main program in the system. Any results that may be returned from the query will be stored in a separate storage area.

## 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. Since Active KDL methods are implemented in C++ and the task class is available for implementing process-oriented simulations, Active KDL can be used to implement a powerful simulation support environment. Much work remains to be done. Currently, only partial prototype implementations of Active KDL and SIMODULA-equivalent simulation capabilities have been completed. We hope to eventually have a complete implementation of Active KDL and query driven simulation on a shared-memory multiprocessor. In this implementation, we would attempt to exploit as much parallelism as possible in terms of:

Query Driven Simulation - execute each parameter set in parallel
Parallelism within a Model - e.g., use the TimeWarp algorithm
Parallel Database Transactions - use high performance protocols [Grif 85]
Parallelism within Query Processing - e.g., use a parallel join algorithm

## 8. REFERENCES


[Agra 89] R. Agrawal and N. Gehani, *Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++*, Proc. Second International Workshop on Database Programming Languages, R. Hull, R. Morrison and D. Stemple (Eds.), Gleneden Beach, OR (June 1989).

[Andr 89] T. Andrews, C. Harris and K. Sinkel, *The ONTOS Object Database*, Ontologic Technical Report (1989).

[Andr 87] T. Andrews and C. Harris, *Combining Language and Database Advances in an Object-Oriented Development Environment*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct 1987).

[Bank 84] J. Banks and J. Carson, **Discrete-Event System Simulation**, Prentice-Hall, Inc., Englewood Cliffs, NJ (1984).

[Beec 88] D. Beech, *Intensional Concepts in a an Object Database Model*, OOPSLA ’88 Conference Proceeedings, San Diego, CA (Sept 1988).

[Berr 88] J. Berry, **The Waite Group’s C++ Programming**, Howard W. Sams & Company Indianapolis, IN (1988).

[Birt 73] G. Birtwistle, O. Dahl, B. Myrhaug and K. Nygaard, **Simula Begin**, Studentlitertur and Auerbach Publishers (1973).

[Bloo 87] T. Bloom, S. Zdonik, *Issues in the Design of Object-Oriented Database Programming Languages*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct 1987).

[CACI 87] CACI, **SIMSCRIPT II.5 Programming Language**, CACI, Inc., Los Angeles, Cal. (1987).

[Capl 87] M. Caplinger, *An Information System Based on Distributed Objects*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct 1987).

[Chen 76] P. Chen, *The Entity-Relationship Model - Toward a Unified View of Data*, ACM Transactions on Database Systems (March 1976).

[Daya 89] U. Dayal, *Queries and Views in an Object-Oriented Data Model*, Proc. Second International Workshop on Database Programming Languages, R. Hull, R. Morrison and D. Stemple (Eds.), Gleneden Beach, OR (June 1989).

[Dewh 89] S. Dewhurst and K. Stark, **Programming in C++**, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).

[Duhl 88] J. Duhl and C Damon, *A Comparison of Object and Relational Databases Using the Sun Benchmark*, OOPSLA ’88 Conference Proceeedings, Orlando, FL (Oct 1987).

[Elma 89] R. Elmasri and S. Navathe, **Fundamentals of Database Systems**, The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA (1989).

[Fish 87] D. Fishman et al., *Iris: An Object-Oriented Database Management System*, ACM Transactions on Office Information Systems (Jan 1987).

[Grif 85] N. Griffeth and J. Miller, *Performance Modeling of Database Recovery Protocols*, Transactions on Software Engineering 11, 6 (June 1985).

[Hamm 78] M. Hammer and D. McLeod, *The Semantic Data Model: A Modelling Mechanism for Database Applications*, Proc. 1978 ACM SIGMOD International Conference on Management of Data (June 1978).

[Hugh 87] D. Hughes, *Modula-2 Co-routines and Simulation*, J. of Pascal, Ada, & Modula-2 6, 4 (1987).

[Hull 87] R. Hull and R. King, *Semantic Database Modeling: Survey, Applications and Research Issues*, ACM Computing Surveys, Vol. 19:3 (Sept 1987).

[Kers 76] L. Kerschberg and J. Pacheco, *A Functional Data Base Model*, Monograph Series Technical Report, Portificia Univ. Catolica do Rio De Janeiro, Brazil, (Feb 1976).

[Kers 86a] M. Kersten and F. Schippers, *Towards an Object-Centered Database Language*, Proc. International Workshop on Object-Oriented Database Systems, Pacific Grove, CA (Sept 86).

[Kers 86b] M. Kersten and F. Schippers, *Using the Guardian Programming Paradigm to Support Database Evolution*, Proc. IFIP TC2 Conference on Knowledge and Data, Algarve, Portugal (Nov 1986). (Published by North-Holland as Data and Knowledge DS-2 (1988).)

[Kesk 90] A. Keskin, *Stategies for Deductive Databases*, submitted to the 28th Annual ACM Southeastern Regional Conference, Greenville, SC (1990).

[Kess 89] V. Kessler and W. Potter, *New Directions for Object-Oriented Database Management Sysytems*, Proceedings of the 27th Annual ACM Southeastern Regional Conference, Atlanta, GA (1989).

[Kim 87] W. Kim, J. Banerjee, H. Chou, J. Garza and D. Woelk, *Composite Object Support in an Object-Oriented Database System*, OOPSLA ’87 Conference Proceeedings, Orlando, FL (Oct 1987).

[Kim 88] W. Kim, N. Ballou, J. Banerjee, H. Chou, J. Garza and D. Woelk, *Integrating an Object-Oriented Programming System with a Database System*, OOPSLA ’88 Conference Proceeedings, San Diego, CA (Sept 1988).

[Kowa 86] R. Kowalski and M. Sergot, *A Logic-Based Calculus of Events*, New Generation Computing 4, 1 (1986).

[Law 82] A. Law and W. Kelton, **Simulation Modeling and Analysis**, McGraw-Hill, N.Y. (1982).

[Lipp 89] S. Lippman, **C++ Primer**, Addison-Wesley Publishing Company, Inc. Reading, MA (1989).

[Maie 86] D. Maier, J. Stein, A. Otis, A. Purdy, *Developement of an Object-Oreiented DBMS*, OOPSLA ’86 Conference Proceeedings, Portland, Oregon (Sept 1986).

[Mann 89] M. Mannino, I. Choi and D. Batory, *An Overview of the Object-Oriented Functional Data Language*, Proc. Fifth International Conference on Data Engineering, Los Angles, CA (Feb 1989).

[Merr 87] T. Merrow and J. Laursen, *A Pragmatic System for Shared Persistent Objects*, OOPSLA '87 Conference Proceeedings, Orlando, FL (Oct 1987).

[Mill 90] J. Miller, O. Weyrich, W. Potter and V. Kessler *The SIMODULA/OBJECTR Query Driven Simulation Support Environment*, in **Progress in Simulation**, J. Leonard and G. Zobrist (Eds.) (1990).

[Mill 89] J. Miller and O. Weyrich, *Query Driven Simulation Using SIMODULA*, Proceedings of the 1989 Eastern Multiconference: 22nd Annual Simulation Symposium, Tampa, FL (March 1989).

[Mill 88] J. Miller, O. Weyrich 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).

[Penn 87] D. Penney and J. Stein, *Class Modification in the GemStone Object-Oriented Database System*, OOPSLA '87 Conference Proceeedings, Orlando, FL (Oct 1987).

[Pott 90a] W. Potter, J. Miller and K. Kochut, *Supporting an Intelligent Simulation/Modeling Environment Using the Active KDL Object-Oriented Database Programming Language*, submitted to the 21 Annual Pittsburgh Conference on Simulation and Modeling, Pittsburgh, PA (May 1990).

[Pott 90b] W. Potter, K. Kochut and J. Miller *Implementing Concurrency and Simulation Capabilities in the Active KDL Object-Oriented Database Programming Language*, submitted to the 16th International Conference on Very Large Data Bases (1990).

[Pott 89] W. Potter, R. Trueblood and C. Eastman, *Hyper-Semantic Data Modeling*, Data & Knowledge Engineering (1989).

[Pott 88a] W. Potter, *KDL-ADVISOR: A Knowledge/Data Based System Written in KDL*, Proceedings of the 21st Annual Hawaii International Conference on System Science, Kailua-Kona, Hawaii (Jan 1988).

[Pott 88b] W. Potter and R. Trueblood, *Traditional, Semantic and Hyper-Semantic Approaches to Data Modeling*, IEEE Computer (June 1988).

[Pott 87] W. Potter, R. Trueblood, C. Eastman, and M. Mathews, *KDL: A Hyper-Semantic Data Model Specification Language*, Proceeeding of the 2nd International Symposium on Methodologies for Intelligent Systems, Colloquia Program, Charlotte, NC (Oct 1987).

[Pott 86] W. Potter 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, (Nov 1986).

[Prit 86] A. Pritsker, **Introduction to Simulation and SLAM II**, 3rd Ed., John Wiley & Sons, N.Y. (1986).

[Rich 87] J. Richardson and M. Carey, *Programming Constructs for Database System Implementation in EXODUS*, Proc. ACM-SIGMOD 1987 International Conference on Management of Data, San Francisco, CA (May 1987).

[Russ 83] E. Russell, **Building Simulation Models with SIMSCRIPT II.5**, CACI, Inc., Los Angeles, Cal. (1983).

[Shar 88] R. Sharma and L. Rose, *Modular Design for Simulation*, Software - Practice and Experience (October 1988).

[Ship 81] D. Shipman, *The Functional Data Model and the Data Language DAPLEX*, ACM Transactions on Database Systems (March 1981).

[Smit 77] J. Smith and D. Smith, *Database Abstractions: Aggregation and Generalization*, ACM Transactions on Database Systems (June 1977).

[Smit 87] K. Smith, S. Zdonik, *Intermedia: A Case Study of the Differences Between Relational and Object-Oriented Database Systems*, OOPSLA '87 Conference Proceeedings, Orlando, FL (Oct 1987).

[Stan 87] C. Standridge and A. Pritsker, **TESS: The Extended Simulation Support System**, Halsted Press, NY (1987).

[Ston 86] M. Stonebraker and L. Rowe, *The Design of POSTGRES*, Proc. ACM-SIGMOD International Conference on Management of Data, Washington, DC (June 1986).

[Stro 86] B. Stroustrup, **The C++ Programming Language**, Addison-Wesley Publishing Company, Inc. Reading, MA (1986).

[Toml 89] C. Tomlinson and V. Singh, *Inheritance and Synchronization with Enabled-Stes*, OOPSLA '89 Conference Proceeedings, New Orleans, LA (Oct 1989).

[Wein 88] R. Weiner and L Pinson, **An Introduction to Object-Oriented Programming and C++**, Addison-Wesley Publishing Company, Inc. Reading, MA (1988).

[Wirt 85] N. Wirth, **Programming in Modula-2**, 3rd Ed., Springer-Verlag, NY (1985).

## 9. APPENDIX

```
schema ::= SCHEMA schema-name;
        { object-type }
END schema-name;

type ::= primitive-type | object-type

primitive-type ::= CHAR | INTEGER | REAL | STRING

object-type ::= OBJECT_TYPE class-name HAS
        [ SUPERTYPES:
                { class-name { , class-name } ; }
        [ ATTRIBUTES:
                { attribute-name: [ SET OF | LIST OF ] type
                        [ INVERSE OF attribute-name [ (class-name) ] ] ;
                        [ WITH CONSTRAINTS { constraint; } ] }
        [ HEURISTICS:
                { rule; } ]
        [ CONSTRAINTS:
                { constraint; } ]
        [ METHODS:
                { method; } ]
        [ MEMBERS:
                { member-name: class-name; } ]
END class-name;

rule ::= rule-name (paramters): type IS queries END rule-name

queries ::= query { ; query }

query ::= simple-queries |
        FOR quantifier variable IN simple-query { , variable IN simple-query }
                WHERE predicate APPLY queries END

simple-queries ::= simple-query { , simple-query }

simple-query ::= argument | function-name (arguments) |
                function-name (simple-queries) | constant-set

function-name ::= built-in-function-name | rule-name | attribute-name

quantifier ::= ALL | SOME

constraint ::= constraint-name (parameters) IS predicate

predicate ::= term { OR term }

term ::= atomic-predicate { AND atomic-predicate }

factor ::= simple-query rel-op simple-query | NOT factor | (predicate)

rel-op ::= = | <> | < | > | <= | >= | IN

method ::= method-name (parameters) [ : type ]

argument ::= variable | constant | class-name

arguments ::= [ argument { , argument } ]

constant-set ::= "{" constant .. constant "}" | "{" constant { , constant } "}"

parameters ::= [ parameter-name: type { , parameter-name: type } ]
```

**Figure 8. Partial EBNF Grammar for Active KDL.**