On the Integration of Knowledge, Data, and Models

John A. Miller, Walter D. Potter and Krys J. Kochut

Department of Computer Science
Artificial Intelligence Programs
University of Georgia
Athens, GA 30602

1. Introduction

In this paper, we highlight some of our ongoing research on integrating knowledge, data, and models. Providing crisp definitions is a difficult proposition. After all, everything in a computer boils down to data and instructions. If we take an object-oriented viewpoint, we can clarify the picture. Under the object-oriented paradigm, everything is an object, which is itself an encapsulation of data and methods to manipulate and access the data within the object. Thus we can define knowledge, data, and models each as special kinds of objects. The purpose of a data object is to store facts or raw information, and the methods are relatively simple and so uniform in their behavior that often they can be automatically generated (or even pulled out of objects into a database management system). One could define a knowledge object as one that stores a minimal amount of data and is able derive additional information. Finally, a model object is similar to a knowledge object, except that it can use a general computational procedure to generate additional information.

Knowledge objects and model objects seem similar enough that one could unify the concepts. While this is certainly so, the question is, should one. We feel that one should not for the reasons discussed below.

A knowledge object has a collection of rules that can be viewed as quanta (small packets) of knowledge that can exist relatively independently of their context. A useful and popular way to specify a rule is to use "if-then" notation.

if y is-parent-of x **then** y is-ancestor-of x;
if z is-parent-of y and y is-ancestor-of x **then** z is-ancestor-of x;

Given a question or query, e.g., "Who are the ancestors of Joe Smith", the system can provide a response by processing these rules. One of the reasons for the success of knowledge base systems and expert systems† is that the specification of the knowledge (i.e., the rules) is separated from the algorithm for executing the rules. Such is not the case with traditional software, where the knowledge and execution algorithm are interwoven. This not only makes the knowledge difficult to see, but often makes it so brittle that it can only be applied to a very limited application. Rule processing schemes such as forward and backward chaining have been successfully applied to complex rule (knowledge) bases. These schemes select rules to be fired based upon what is currently known to be true. For example, with forward chaining if the antecedent (if-part) of a rule is true, then the consequence (then-part) is executed.

Deductive databases use rules to derive new information, much like an expert system. If the rule language of a deductive database system is relatively simple (e.g., Datalog [Ullm88] which is one of the more popular rule languages) its semantics can be defined fairly easily. On the other hand, relaxing the restrictions imposed on the language of the deductive database (i.e., enlarging its expressive power) may lead to systems without well-defined semantics, and even with no effective evaluation method (there may be no guarantee that the evaluation process will halt) [Ceri89]. An additional benefit of restricting the expressive power of the language is that it allows sophisticated optimizations to be carried out so that answers can be provided more rapidly [Ullm89].

Restricting the rule language, on the other hand, makes it so that certain computational tasks can no longer be performed. In many application areas such as finance, operations

† Artificial Intelligence (AI) researchers have discovered that many experts (e.g., medical doctors and investment bankers) reason by applying heuristics or rules of thumb. Much of their knowledge consists of a large collection of rules that has been refined through years of experience. During the last two decades AI researchers have developed tools, programming languages and shells, which enable collections of rules to be stored and processed by computers allowing them to function like an expert. Some of the more popular tools for building expert systems are Prolog, OPS5, Nexpert/Object, and KEE [Turb92].

research and simulation special computational tasks are called models. They usually involve general computations, so they are not expressible in a restricted rule language. Consequently, an integrated knowledge, data, and model base needs a language which is equivalent in power to a Turing machine [Hopc79, p. 166] (Turing complete) to express models. Control of the execution of a collection of models, must now be handled by ad hoc means, since no general unifying theoretical foundation can be built (the problems are provably undecidable).

Since models are embedded within objects, just as rules are, we can develop some facility for operating on models. We can retrieve them, instantiate them by assigning values to their parameters, execute them, and even combine them to produce more complex composite models. This process is very difficult to formalize, because of its inherent complexity and generality. Consequently, techniques (algorithms and rule bases) should be developed within certain application domains. The application domain of primary interest to our research group is simulation (i.e., our models are simulation models).

A simple illustration of the distinction between the three (knowledge, data, and models) can be seen by considering birds: (1) Data -- Tweetie is a bird; (2) Knowledge -- Birds can fly (allows us to infer that Tweetie can fly); (3) Model -- An aerodynamic simulation/animation model showing how birds fly. In an integrated system, the interplay of knowledge, data, and models can provide a symbiosis, making such a system greater than the sum of its parts. Figure 1 shows some of the interactions that can occur.

In the rest of this paper, we show how an *object-oriented database system*† can be developed to facilitate the integration of knowledge, data, and models. We indicate how this can be used to support a query driven simulation environment. Finally, we discuss some of the solutions we are developing to manage a complex integrated knowledge, data, and model base.

---

† The evolution of database systems is an interesting subject in its own right [Elma89], from the hierarchical and network databases developed in the mid 1960's to the relational databases developed in the late 1970's to the object-oriented databases currently being developed. Some of the more popular object-oriented database systems currently available include: ONTOS by Ontologic [Andr89], GemStone by Servio-Logic [Maie86], UniSQL/X by UniSQL [Kim91], ObjectStore by Object Design [Obje90], and ITASCA [Itas90], a commercial version of the ORION database system [Kim89].

Figure 1

2. The Active KDL Object-Oriented Database System

Active KDL (Knowledge/Data Language) is a functional object-oriented database system. It evolved [Pott92] from earlier work on KDL which has been ongoing since 1986 [Pott86]. The foundations of Active KDL are threefold: object-oriented programming, functional programming, and hyper-semantic data modeling. These areas strongly influenced the design of Active KDL's three sublanguages: the schema definition language (SDL), the query language (QL), and the database programming language (DBPL). Because of the capabilities and elegance of these sublanguages, Active KDL is capable of supporting demanding applications (e.g., Simulation [Mill90], Model Management [Mill91c], Computer-Aided Design (CAD) [Pola91], Genome Database [Mill92b], and Intelligent Database Applications such as a university knowledge/data base capable of advising students [Pott88a]).

Integrating the three foundation areas via Active KDL is the focus of our work (and has been since the mid-1980's), and represents the primary difference between Active KDL and other object-oriented database systems.

The underlying foundation of KDL [Pott87, Pott88b] as well as Active KDL [Mill90, Mill91a, Mill91c, Pott90] is the hyper-semantic data model KDM (Knowledge/Data Model) [Pott86, Pott88b, Pott89]. Hyper-semantic data models provide extensive modeling capabilities that include those of standard semantic data models and object-oriented data models [Atch89, Lee90, Pott88b, Pott89]. The scope of modeling capabilities for hyper-semantic data models includes knowledge, data, and model management [Mill90, Pott88a, Pott90, Tidr88, Wood90].

The KDM evolved from the Functional Data Model (FDM) [Kers76, Ship81]. This evolution was motivated by the need for knowledge management facilities to be incorporated with advanced data modeling facilities in a tightly coupled manner. Consequently, the KDM has a highly functional nature. In addition to DAPLEX [Ship81], the design of Active KDL was influenced by functional programming languages such as Hope [Burs80], Standard ML [Harp86], and Miranda [Turn85].

The ultimate goal of our research is to continue to improve the accuracy of modeling information (and decision support) systems. Our approach to achieving this goal is based on the integration of concepts and techniques from the areas of artificial intelligence, data modeling, and object-oriented programming languages. It is clear to us that future intelligent information systems will provide users with an integrated knowledge/data/model management environment. The KDM provides the foundation for our integrated approach to developing these new types of systems.

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

1. Active KDL serves equally well as a knowledge/data modeling language, a query language, and a manipulation/application language. Specifically, it consists of a schema definition language (SDL), a query language (QL), and a database programming language (DBPL). The database programming language is a superset of the query language. The power of Active KDL comes from its DBPL which is Turing complete.

2. Active KDL supports hyper-semantic data modeling which is based on the following abstraction mechanisms: classification/instantiation, specialization/generalization, aggregation, association (or membership), knowledge (in the form of constraints and heuristics), and concurrent/temporal capabilities.

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

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

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

6. Limitations are placed on the syntax that can be used to express constraints, making the constraint language equivalent in power to Datalog [Ullm88]. Consequently, constraint analysis, as developed by Delcambre and Urban [Urba90], is applicable to our system.

7. Active KDL supports active objects, that is, objects performing some task. These facilities are designed to support high concurrency [Mill91b].

8. Preliminary work on adding historical extensions to KDL has recently begun [Koch91b].

2.1. The Active KDL Type System

The three KDL sublanguages, i.e. schema specification, query, and database programming sublanguages, are strongly typed. All values are grouped into types. There are three main type groups in KDL: primitive types (*integer*, *real*, *char*, and *boolean*), collection types, and object types.

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** T) is a collection of values from types *conforming* to T (a formal definition of the meaning of type S conforming to type T is given in the next subsection). The element type T on which a set is defined may be any primitive type or object type. In 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. Types of values in a list of type (**list** T) must conform to T. A **string** object is simply a list whose elements are of type **char** (i.e., **string** is an alias for (list char)).

Object types (classes) form the foundation of KDL. They are the main building blocks of the database schema specification language. There are two kinds of object types in KDL. The values of a class which are static in their behavior can be modeled by *passive* object types. Passive objects only react to messages and have no control of their own. On the other hand, certain situations require modeling of active entities representing some specified behavior. Such objects are characterized not only by a set of usual functions but also by a special method called the *script*, which specifies the behavior (scripts play a similar role in Actor-type languages [Hewi77]). An object type for which a script method has been provided is said to be *active*.

Every object instance with at least a single attribute is constructed by the evaluation of a special method called *create*, which creates a new object instance and initializes its attributes. All objects in KDL are *persistent*, that is, they remain in existence (stored in the database) after they are created, and even if they are no longer referenced by names (variables) in a running KDL program, or if the program has terminated. From this point of view there is no difference between active and passive objects (if a KDL program terminates, so do the scripts of all active objects).

A KDL database consists of a collection of objects. As prescribed by the classification abstraction, similar objects are grouped into object types (or classes). Instances of an object type (or class) are called objects. In general, an object (as a value) is an entity composed of other values whose types may be different. The syntax of a KDL schema definition (schema ::=) is shown in Figure 2. Each object type specification begins with the definition of the name of a class in the **object-type** clause.

The optional characteristics of an object type correspond to the KDM modeling primitives while the class-name uniquely identifies the object type. Any object type may be defined as a specialized (derived) form of one or more other object types, called *supertypes*.

The **:supertypes** argument specifies the *base* object types from which a new object type is derived. The arguments may introduce a number of *base* object types. Single or multiple inheritance is determined by the number of supertypes. Specification of supertypes is not without restrictions (see the section on inheritance in KDL). Briefly, any number of object types may be

```
schema ::= (schema schema-name object-type+)

object-type ::=
    (object-type object-type-name
   [:supertypes  ( {object-type-name}+ )   |
     :subtypes  ( {object-type-name  [:hiding  {function-name}+ ] }+ ) ]
   [:attributes  ( {attribute}+ ) ]
   [:members  ( {member}+ ) ]
   [:constraints  ( {constraint}+ ) ]
   [:heuristics  ( {heuristic}+ ) ]
   [:methods  ( {method}+ ) ] )

attribute ::= (attribute-name any-type)
member ::= (member-name any-object-type
                [:inverse-of (member-name object-type-name)])
constraint ::= (constraint-name ((identifier object-type-name) boolean)
               boolean-expression)
heuristic ::= (heuristic-name ((identifier object-type-name) return-type)
               expression)
method ::= (method-name ((identifier object-type-name) parameters return-type)
               method-body)
method-body ::= expression | :clos clos-expression
any-type ::= primitive-type | collection-of-primitives | any-object-type
collection-of-primitives ::= (set primitive-type)  |  (list primitive-type)
primitive-type ::= integer | real | char | boolean
any-object-type ::= object-type-name | collection-of-objects
collection-of-objects ::= (set object-type-name)  |  (list object-type-name)
```

Figure 2: Syntax of KDL Schema Definition Language.

chosen as supertypes of a new object type, provided that all of them belong to the same *inheritance lattice* (they must have a common ancestor in the hierarchy).

The **:subtypes** argument specifies the object types from which the current object type is *generalized*. Optionally, certain functions from object types listed on the :subtypes list may be hidden. This is done by listing the function names after the **:hiding** keyword.

2.1.1. Functions

The functional approach is present in all aspects of KDL. An object type (or class) may be viewed as an encapsulation of functions. In KDL there are five flavors of functions: attributes, members, constraints, heuristics, and methods.

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 another object. KDL also supports multi-valued attributes which map an object into a set or list. An object type (or class) has zero or more attributes that are defined in the **:attributes** clause. Each attribute must have a defined type. This corresponds to *slots* in CLOS, or *data members* in C++. Attributes are defined by the following form:

(*attribute-name type*)

where *type* can be any of the primitive types, collection types, or the name of any of the previously defined object types.

Members. Memberships or associations between objects are fundamentally important for any semantic, hyper-semantic, or object-oriented data model. Objects without connections to other objects are typically insignificant. The **:members** argument specifies the list of members. Members are defined by the following form:

(*member-name type*)

where *type* must be the name of any of the previously defined object types. Members enable associations between objects to be formed. For example, consider the university schema example that includes Student and Course object types. Associations between students and courses can be easily established as shown in Figure 3. This is an example of a many-to-many relationship (or association). The **:inverse-of** clause specifies that Students and Courses are just opposite ends of the same relationship. The **:members** clause in KDL provides a convenient mechanism for grouping together objects, such as students in a course or a thesis committee for a graduate student.

Constraints. In KDL, constraints may be imposed in very general ways. 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 KDL is any Boolean function that is constructible using the query language without method calls and built-in arith-

```
(object-type Student
    :supertypes (Person)
    :attributes ((GPA real)
                 (Major string)
                 (Degree string))
    :members ((Courses (set Course)))
    :constraints ((Limit ((s Student) boolean)
                    (<= (Num_Hours s) 15)))
    :heuristics ((Num_Hours ((s Student) integer)
                   (Sum (for (c :in (Courses s))
                          :eval (Credit_Hours c))))))

(object-type Course
    :attributes ((Title string)
                 (Abbreviation string)
                 (Credit_Hours integer)
                 (Days string))
    :members ((Students (set Student) :inverse-of (Courses Student))
              (Dept (set Academic_Department)))
    :constraints ((Valid_Days ((c Course) boolean)
                    (in (Days c) ("MWF" "TTH" "DAILY")))))
```

Figure 3: Student and Course Association.

metic functions.† The **:constraints** argument introduces a number of constraints, which are one argument, Boolean-valued functions. In the simplest case, the user can impose various restrictions on attribute values. It is also possible to specify more complex constraints involving other objects, or even interactions with other objects, as long as the return value is of type *boolean*. Constraints are defined with the use of the following form:

(*constraint-name* ((*parameter object-type-name*) `boolean`)
    predicate)

where the *predicate* is a Boolean-valued expression. 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, in certain circumstances, user-initiated).

Heuristics. Heuristics, or rules, allow information about an object to be inferred, rather than retrieved using a traditional query against stored data. Thus, heuristics provide an information derivation mechanism that results in greater informational content than is present in the stored data alone. The **:heuristics** argument introduces a number of *heuristic functions*. A

†Such limitations make the constraint language equivalent to Datalog [Ullm88], and, consequently, constraint analysis, as presented in [Urba90], is applicable to our system.

heuristic function is similar to an attribute function. The main difference is that it computes a value (conceptually via an inference), instead of simply retrieving one. The format for a heuristic function definition is as follows:

(*heuristic-name* ((*parameter* *object-type-name*) *return-type*)
expression)

A heuristic function takes a single argument of type *object-type-name* and returns a value of a given *type*. Heuristics give KDL fundamentally more expressive power than a relationally complete language like SQL. Formally, a heuristic is a function that is expressed as a parameterized query. The parameterization is in the form of passing an object of the class as a parameter to the heuristic. (Note that heuristics are overloaded so that they can apply to any subset of objects from the class as well as to a single object.) Meta-rules may be specified to control and manage further results (e.g., providing an abstract query response or an inference explanation).

Methods. 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 in 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. It may be written inline or separately. Methods are used to perform data manipulation, and implement applications. Additionally, methods may be written directly in the host language, i.e. CLOS, thus giving the programmer more freedom in terms of programming constructs and access to scores of CLOS and COMMON LISP functions. An important advantage of using the KDL Database Programming Language (DBPL) in writing the body of a method is, that, aside from their simplicity, such a method will be evaluated in a controlled, well typed environment. The KDL DBPL constructs and CLOS/COMMON LISP expressions may not be mixed. A method can be defined with the use of one of the two following forms:

(*method-name* ((*parameter-name* *type*) ... *return-type*)
expression)

(*method-name* ((*parameter-name* *type*) ... *return-type*)
:clos *CLOS-or-COMMON-LISP-form*)

Both heuristic functions and methods can be defined within an object type definition, or possibly later within the implementation module. The **create** and **script** methods may not be coded as **:clos**-type methods. Creation of an object as well as its behavior must be described in the controlled, strongly typed environment of KDL.

3. Query Driven Simulation

There are two ongoing research projects that address aspects of knowledge, data, and model management in an integrated system. Namely, our own work with Active KDL and that of Delcambre on Object Flow Modeling [Delc90a, Delc90b]. There are also a few projects that are closely related. These systems were developed because of the difficulty of simulating large complex systems with traditional tools. This 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. Included are 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 [Prit86, Stan87]) 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 [Birt73, Birt87], MODSIM II [CACI90] and *knowledge-based simulation* (e.g., KBS [Fox89, Redd86], ROSS [McAr85, Roth89], and SES/MBase [Zeig87, Zeig90, Zhan89]) allow new models to be composed from existing models, thereby enhancing the process of model development.

Our query driven simulation [Mill89, Mill90, Mill91a, Mill91c, Mill92a, Mill92b] is a powerful approach to simulation modeling and analysis. 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 on 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 on query driven simulation is 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.

Active KDL is designed to support the complex information needs of *engineering databases* and *expert databases*. In particular, Active KDL provides an integration of *knowledge bases*, *databases*, and *model 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 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. These facilities provide a powerful mechanism for building simulation models out of pre-existing model components.

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 [Mill90] 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.

Model instantiation occurs when Active KDL does not have sufficient data or knowledge to provide a satisfactory user response. In such a case, Active KDL automatically creates model instances that are executed to generate enough data to give a satisfactory reply 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, control heuristics or a user settable threshold will need to be provided to control the amount of computation. For example, 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 0%, then only data generated from previous simulation runs will be retrieved. Intermediate values for the threshold would typically be the case.

3.1. Example Simulation Model

As a simple example, let us derive a simulation object-type from the object-type Student defined in Figure 3. This object-type (see Figure 4) is made active by including Sim_Object (see [Koch91a, Mill91a] for details) in its list of supertypes. An instance of this new Registering_Student object-type will simply be served by an advisor for a random amount of time. (Note that a more realistic model would have queueing effects.)

```
(object-type Registering_Student
    :supertypes (Student Sim_Object)
    :attributes
        ((Mean_Advising_Time real)
         (Arrival_Time real)
         (System_Time real))
    :methods
        ((create ((mean_Advising_Time real :default 10.0))
                                   (Registering_Student)
            (create Mean_Advising_Time = mean_Advising_Time))
         (script (Registering_Student)   ; only return type
            (let (r = (Work (Exponential 1 Mean_Advising_Time)
                            (recreate (Arrival_Time = (Time Clock))))
            :in (recreate (System_Time = (- (Time Clock)
                                            (Arrival_Time r))))))))
```

Figure 4: Example Sim_Object.

Instances of the object-type (or class) Registering_Student are active simulation objects. When ordinary passive objects are created, at the time of creation, values for their attributes must be assigned and values for their members may be assigned. However, for active objects this is only the beginning. After initial values are assigned, the script processes are automatically initiated, which may during the course of execution (dynamics) cause additional state changes through the use of the recreate construct.

Instances of Registering_Student may be created in several ways. A user may explicitly code another object-type, say Registration_Model of which Registering_Students will be a part. One may also create instances by direct querying. Finally, one can use an automatically produced meta-level object (within the Meta_Sim_Object class) that serves the dual purpose of providing a high-level representation and a simulation driver. The high-level representation will keep track of such things as: (1) the input and output parameters (input parameters appear within the create construct, while output parameters appear within the recreate construct); (2) the definite and indefinite delays (an example of a definite delay is service for a given amount of time and an example of a indefinite delay is the wait in a queue), and (3) all the object-types touched by this object-type. This information is vital to the successful operation of the Strategic Planner (see section 4.1) which is responsible for the overall planning involved in answering queries (e.g, whether to simply retrieve data, infer information, produce simulation replications, or even produce composite simulation models). The meta-level object will contain a reference to the automatically produced class shown in Figure 5.

```
(object-type Registering_Student_Model
    supertypes: (Registering_Student Statistics)
    attributes:
        ((Mean_Inter_Arrival real)
         (Number_Registering_Students integer))
    members:
        ((Registering_Students (set Registering_Student)))
    methods:
        ((create ... )
         (script ... )))
```

Figure 5: Automatically Produced Model Class.

The Registering_Student_Model class is a specialization of Registering_Student and it is also an association of Registering_Student's. Specialization is used to allow a prototypical Registering_Student to be available to the model to facilitate accessing parameter values. The association (which in this case is a one-to-many relationship) indicates the set of Registering_Student's involved in a particular model instance (or scenario). The script within this model class will serve as the driver for periodically creating Registering_Student active objects. The fact that the model is also derived from the Statistics object-type enables standard statistical results (e.g., means and variances) to be produced.

3.2. QDS Queries

In this subsection, we present successively more complex examples of query driven simulation (QDS). In query driven simulation, one is usually interested in certain aspects of a subset of potential objects. 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.

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

Suppose that an analyst wishes to know the average time that students spend registering for courses. In particular, if the analyst is interested in the performance given that the mean interarrival time is 15 minutes and the mean service time is 10 minutes, the analyst could simply formulate the following query:

```
(for (r :in Registering_Student_Model)
    :where (and (= (Mean_Advising_Time r)  10.0)
                (= (Mean_Inter_Arrival r)  15.0))
    :eval (Mean (System_Time (Registering_Students r))))
```

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 may be 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_Advising_Time = 10.0). Input parameters not mentioned in the `:where` predicate are set to their default values (e.g., Number_Registering_Students = 100). These default values are found in the schema as default values for the parameters of the class constructor. Details about model instantiation may be found in [Mill90].

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 changes with the efficiency of the advisement process. The following query, in which the Mean_Advising_Time input parameter is varied, provides the answer.

```
(for (r :in Registering_Student_Model)
    :where (and (in (Mean_Advising_Time r) (6.0 8.0 10.0))
                (= (Mean_Inter_Arrival r)  15.0))
    :eval (Mean_Advising_Time r)
          (Mean (System_Time (Registering_Students r))))
```

This is not a point query since multiple input parameter sets will be needed, one parameter set for each value of Mean_Advising_Time. The Registering_Student_Model will be instantiated and executed once for each of the 3 input parameter sets.

Boolean queries, useful for making comparisons between alternate configurations of a system, allow arbitrarily complex conditions to be given in the `:where` predicate. The `:where` predicate is transformed to its disjunctive normal form, if necessary, and a single parameter set is created for each conjunct. These conjuncts are independent and hence on the appropriate parallel hardware could be executed simultaneously. For example, suppose an analyst wishes to know the impact of making the advisement process twice as efficient. The query below, which is in disjunctive normal form, will provide the answer.

```
(for (r :in Registering_Student_Model)
    :where (or (and (= (Mean_Advising_Time r) 10.0)
                    (= (Mean_Inter_Arrival r) 15.0))
               (and (= (Mean_Advising_Time r)  5.0)
                    (= (Mean_Inter_Arrival r) 15.0)))
    :eval (Mean_Advising_Time r)
          (Mean (System_Time (Registering_Students r))))
```

The above example generates the following parameter sets:

PS #1: Mean_Advising_Time = 10.0, Mean_Inter_Arrival = 15.0, "defaults"
PS #2: Mean_Advising_Time = 5.0, Mean_Inter_Arrival = 15.0, "defaults"

Model instantiation becomes significantly more complex when multiple models are referenced by a query. Not only are individual models instantiated with parameter values, but joint parameterization needs to be done in a systematic and efficient manner. Typically, if systems are being compared they will have something in common, namely, input parameters (or attributes in database terminology). These common attributes are then used as the basis for a join-like operation. This capability can be used to compare different but related systems.

4. Integration Techniques

One aspect of integration is to develop a language suitable for expressing knowledge, data, and model objects. More important than this is to develop the meta-level representations of these objects, and to develop the algorithms and rule bases to be used in planning and decision making related to integrating knowledge, data, and models. The decision making process on how best to answer a query is divided into two phases: Strategic Planning and Tactical Planning.

4.1. Strategic Planning

Based on the existing meta-data and on input from three types of models (cost, adequacy, and preference) a variety of high level decisions will need to be made. Using general cost and adequacy models, and user-specific preference models, global planning will be initiated to determine the appropriate actions to be taken. Because of the scope and complexity of the task of strategic planning, the system will rely primarily on rule based reasoning.

A myriad of possibilities exist for accessing the information: existing data can be retrieved, data can be inferred using knowledge, and/or data can be generated using models. Furthermore, it may suit a user's preference to obtain a rough, incomplete answer in a short amount of time, but also expect a precise, complete answer to be forthcoming. The strategic planner is responsible for making such decisions on how many answers to give, and on what combination of bases (knowledge, data, and model) to access.

For example, suppose a user can tolerate delays of a few minutes to obtain moderately precise information. Once the query is submitted, the system discovers it is fundamentally new. Only a small portion of the information has been stored from previous simulation runs. Simulation models are available to provide precise answers, but will they finish soon enough? The strategic planner decides that it is important enough to find out, so it begins a two-stage batch mean simulation. After the first stage, an estimate is obtained of how many more batches will be needed to obtain a confidence interval of the required relative precision. While this process is executing, to be safe, the strategic planner spawns another process to explore another angle, in case the simulation can not complete in time. This other process takes the existing data, and applies any relevant rules from the knowledge base to try to come up with some reasonable answer (e.g., "Yes, you should abandon ship!").

In even more complex situations, the strategic planner may determine that an appropriate model does not exist in the model base, but that by combining model components using the abstraction mechanisms of Active KDL a new model can be synthesized. Finally, the strategic planner must deal with trade-offs involved in providing answers to queries and also concerning the nature of the base (of knowledge, data, and models) itself: (1) Storage versus Computation (should all or part of the generated and/or inferred data be stored or recreated when needed); (2) Degree of Precision; and (3) Degree of Completeness.

5. Tactical Planning

Tactical planning involves detailed decision making on how to achieve a specific, concrete goal. The techniques used are algorithmic, although in many cases the algorithms are heuristic (e.g., Query Optimization is NP-Hard [Gare78]).

For query optimization while accessing the database, a detailed plan is produced indicating what operations (e.g., join), access paths, and indices are to be used to retrieve the appropriate data. We do not add to or change the techniques used for query optimization, but rather integrate them within tactical planning. However, our approach is extensible so that other query optimization techniques can be added.

For rule selection, given a query that requires new data to be inferred, rules are selected/indexed based upon their relevancy, and inference is carried out by forward and/or backward chaining depending on the situation (a combined forward/backward approach is sometimes appropriate). Again, we do not add to or change the techniques used for rule processing, but rather integrate them within tactical planning.

Lastly, for model instantiation, given a query requiring new data to be generated, models are selected, parameter values are assigned to create model instances, and then these instances are executed to generate new data. Model instantiation is fundamental to our research so we are developing and testing algorithms for it, in addition to integrating it into tactical planning.

6. Summary

A key underlying and evolving theme of computer science is to capture reality ever more closely. Clearly, full reality is the beyond the capabilities of the most capable computers and computer programs. Consequently, approximations or representations of reality are the best one can hope for. As an example, let us suppose that we desire to capture within a computer the reality of a neurosurgeon. Let us consider what elements would be involved. The easiest things to deal with are the basic facts about the person (name, age, height, weight, etc.). Much more complex than this is to capture his/her knowledge (e.g., what to do if a tumor next to the spinal cord shows up on a Magnetic Resonance Imaging (MRI) photograph). Finally, once a decision is made on what to do, what kind of process will be carried out by the neurosurgeon (e.g., laser surgery)? From the point of view suggested by this example, complexity increases from data (e.g., accounting data) to knowledge (e.g., diagnosis) to process model (e.g., surgery).

We believe that object-oriented database systems provide a foundation for developing systems capable of managing knowledge, data, and models in an integrated manner. Our ongoing research with Active KDL and Query Driven Simulation is exploring some of these issues where the domain currently focuses on standard types of discrete-event simulations.

7. References

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

[Andr89] Andrews, T., C. Harris and K. Sinkel, The ONTOS Object Database, Technical Report, Ontologic, 1989.

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

[Birt73] Birtwistle, G., O. Dahl, B. Myhaug and K. Nygaard, *Simula Begin*, Studentlitertur and Auerbach Publishers, 1973.

[Birt87] Birtwistle, G., *DEMOS: A System for Discrete Event Modelling on SIMULA*, Springer-Verlag, New York, 1987.

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

[CACI90] CACI, MODSIM II: The Language for Object-Oriented Simulation, January 1990.

[Ceri89] Ceri, S., G. Gottlob and L. Tanca, What You Always Wanted to Know About Datalog (And Never Dared to Ask), *IEEE Transactions on Knowledge and Data Engineering 1*, 1 March 1989, 146-166.

[Delc90a] Delcambre, L., S. P. Landry, L. Pollacia and J. Waramahaputi, Specifying Object Flow in an Object-Oriented Database for Simulation, *Proceedings of the 1990 Western Multi-Conference, Object-Oriented Simulation Conference*, San Diego, California, January 1990.

[Delc90b] Delcambre, L., J. Waramahaputi, S. P. Landry and L. Pollacia, The Direct Simulation of the Object Flow Model, *Proceedings of Applied Modeling and Simulation Conference, International Association of Science and Technology for Development, International Symposium*, Lugano, Switzerland, June 1990.

[Elma89] Elmasri, R. and S. Navathe, *Fundamentals of Database Systems*, The Benjamin Cummings Publishing Co., Redwood City, CA, 1989.

[Fox89] Fox, M. S., N. Husain, M. McRoberts and Y. V. Reddy, Knowledge-Based Simulation: An Artificial Intelligence Approach to System Modeling and Automating the Simulation Life Cycle, *Artificial Intelligence, Simulation and Modeling*, L. E. Widman, K. A. Loparo and N. R. Nielsen (eds.), New York, 1989.

[Gare78] Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, H. Freeman, San Francisco, 1978.

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

[Hewi77] Hewitt, C. E., Viewing Control Structures as Patterns of Passing Messages, *Journal of Artificial Intelligence 8*, 3 1977, 323-364.

[Hopc79] Hopcroft, J. E. and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass., 1979.

[Itas90] Itasca Systems, Inc., *ITASCA System Overview*, Unisys, Minneapolis, Minn., 1990.

[Kers76] Kerschberg, L. and J. Pacheco, A Functional Data Base Model, *Monograph Series Technical Report*, Brazil, February 1976.

[Kim89] W. Kim and F. H. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications*, Addison-Wesley, Reading, MA, 1989.

[Kim91] Kim, W., Object-Oriented Database Systems: Strengths and Weaknesses, *Journal of Object-Oriented Programming. Special Issue on Databases 4*, 4 July-August 1991.

[Koch91a] Kochut, K. J., J. A. Miller and W. D. Potter, Design of a CLOS Version of Active KDL: A Knowledge/Data Base Systems Capable of Query Driven Simulation, *Proceedings of the 1991 AI and Simulation Conference*, New Orleans, Louisiana, April 1991, 139-145.

[Koch91b] Kochut, K. J., J. A. Miller, W. D. Potter and A. Wright, h-KDL: A Historically Extended Functional Object-Oriented Database System, *Proceedings of the Tools '91 International Conference*, Santa Barbara, California, July 1991, 73-86.

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

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

[McAr85] McArthur, D., P. Klahr and S. Narain, The ROSS Language Manual, Technical Report N-1854-1-AF, The RAND Corporation, September 1985.

[Mill89] Miller, J. A. and O. R. Weyrich, Jr., Query Driven Simulation Using SIMODULA, *Proceedings of the 22nd Annual Simulation Symposium*, Tampa, FL, March 1989, 167-181.

[Mill90] Miller, J. A., W. D. Potter, K. J. Kochut and O. R. Weyrich, Jr., Model Instantiation for Query Driven Simulation in Active KDL, *Proceedings of the 23rd Annual Simulation Symposium*, Nashville, TN, April 1990, 15-32.

[Mill91a] Miller, J. A., K. J. Kochut, W. D. Potter, E. Ucar and A. A. Keskin, Query Driven Simulation Using Active KDL: A Functional Object-Oriented Database System, *International Journal in Computer Simulation 1*, 1 1991, 1-30.

[Mill91b] Miller, J. A. and N. D. Griffeth, Performance Modeling of Database and Simulation Protocols: Design Choices for Query Driven Simulation, *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, Louisiana, April 1991, 205-216.

[Mill91c] Miller, J. A., W. D. Potter, K. J. Kochut, A. A. Keskin and E. Ucar, The Active KDL Object-Oriented Database System and Its Application to Simulation Support, *Journal of Object-Oriented Programming. Special Issue on Databases 4*, 4 1991, 30-45.

[Mill92a] Miller, J. A., O. R. Weyrich, Jr., W. D. Potter and V. Kessler, The SIMODULA/OBJECTR Query Driven Simulation Support Environment, *Progress in Simulation 3 (to appear)* 1992.

[Mill92b] Miller, J. A., J. Arnold, K. J. Kochut, A. J. Cuticchia and W. D. Potter, Query Driven Simulation as a Tool for Genetic Engineers, *Proceedings of the International Conference on Simulation in Engineering Education*, Newport Beach, CA, January 1992, 67-72.

[Obje90] Object Design, *An Introduction to ObjectStore*, Object Design, Inc., Burlington, Mass., 1990.

[Pola91] Polamraju, R. P., *Databases for Engineering Applications*, M.S. Thesis, University of Georgia, August 1991.

[Pott86] Potter, W. D. and L. Kerschberg, A Unified Approach to Modeling Knowledge and Data, *Proceedings of the IFIP TC2 Conference on Knowledge and Data (DS-2)*, Algarve, Portugal, November 1986, V1-V27.

[Pott87] Potter, W. D., R. P. Trueblood, C. M. Eastman and M. M. Mathews, KDL: A Hyper-Semantic Data Model Specification Language, *Proceedings of the 2nd International Symposium on Methodologies for Intelligent Systems, Colloquia Program*, Charlotte, NC, October 1987, 203-214.

[Pott88a] Potter, W. D., KDL-ADVISOR: A Knowledge/Data Based System Written in KDL, *Proceedings of the IEEE Twenty-first Annual Hawaii International Conference on System Science*, Kailua-Kona, Hawaii, January 1988, 319-328.

[Pott88b] Potter, W. D. and R. P. Trueblood, Traditional, Semantic and Hyper-Semantic Approaches to Data Modeling, *IEEE Computer 21*, 6 June 1988, 53-63.

[Pott89] Potter, W. D., R. P. Trueblood and C. M. Eastman, Hyper-Semantic Data Modeling, *Data & Knowledge Engineering 4* 1989, 69-90.

[Pott90] Potter, W. D., J. A. Miller, K. J. Kochut and S. W. Wood, Supporting an Intelligent Simulation/Modeling Environment Using the Active KDL Object-Oriented Database Programming Language, *Proceedings of the 21st Annual Pittsburgh Conference on Simulation and Modeling 21 Part 4* May 1990, 1895-1900.

[Pott92] Potter, W. D., K. J. Kochut, J. A. Miller, V. P. Gandham and R. V. Polamraju, The Evolution of the Knowledge/Data Model, in *Advances in Databases and Artificial Intelligence (to appear)*, Petry and Delcambre (eds.), 1992.

[Prit86] Pritsker, A., *Introduction to SLAM II, Third Edition*, John Wiley & Sons, New York, 1986.

[Redd86] Reddy, Y. V., M. S. Fox, N. Husain and M. McRoberts, The Knowledge-Based Simulation System, *IEEE Software*, March 1986.

[Roth89] 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, New York, 1989.

[Ship81] Shipman, D., The Functional Data Model and the Data Language DAPLEX, *ACM Transactions on Database Systems*, March 1981.

[Stan87] Standridge, C. and A. Pritsker, *TESS: The Extended Simulation Support System*, Halstead Press, New York, 1987.

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

[Turb92] Turban, E., *Expert Systems and Applied Artificial Intelligence*, Macmillan Publishing Company, New York, NY, 1992.

[Turn85] Turner, D. A., MIRANDA: A Non Strict Functional Language with Polymorphic Types, *Proceedings of the IFIP International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Ullm88] Ullman, J. D., *Principles of Database and Knowledge-Base Systems. Vol. I*, Computer Science Press, 1988.

[Ullm89] Ullman, J. D., *Principles of Database and Knowledge-Base Systems. Vol II*, Computer Science Press, 1989.

[Urba90] Urban, S. D. and L. M. L. Delcambre, Constraint Analysis: A Design Process for Specifying Operations on Objects, *IEEE Transactions on Knowledge and Data Engineering 2*, 4 1990.

[Wood90] Wood, S. W., Model Management From An Object-Oriented Perspective, *Unpublished manuscript*, 1990.

[Zeig87] Zeigler, B. P., Hierarchical, Modular Discrete Event Modelling in an Object Oriented Environment, *Simulation Journal 49*, 5 November 1987.

[Zeig90] Zeigler, B. P., *Object-Oriented Simulation with Hierarchical, Modular Models*, Academic Press, Boston, MA, 1990.

[Zhan89] 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, New York, 1989.