The NU& Object-Oriented Semantic Data Modeling Tool: Intermediate Report

Kenneth Baclawski
College of Computer Science
Northeastern University
Boston, Massachusetts 02115

March 1, 1990

Abstract

The nu& system is a semantic object-oriented data modeling tool. It is a research vehicle for studying ways to enhance the modeling power of a transaction management system while maintaining its high-performance characteristics. This is the second of a series of reports on the nu& system.

The author was supported by NSF Grant # CCR-8716485

1 Introduction

The `nu&` system is a semantic object-oriented data modeling tool. It is intended to be a vehicle for both research and education on such tools, and for this reason is a public domain software system. It is concerned with studying ways to enhance the modeling power of a transaction management system while maintaining its high-performance characteristics. The name is a reference to Northeastern University where most of the work is being carried out.

This is the second report on the `nu&` system. The first report [4] described the objectives of the `nu&` system and gave some details about one component of the system called the association model. This model is now called the `alpha` component of `nu&`. It is one of six planned components of the system. In this report the overall architecture of `nu&` is described in general, the components are described in some detail, and the various software modules that link the components together are introduced.

2 Background on Database Management and Object-Oriented Systems

Database management is concerned with the management of large amounts of reliable, shared data. The specificity of this objective contrasts with the situation in programming languages where there are many competing concerns, and progress is more difficult to ascertain since there is no goal to be achieved.

Object-oriented programming is a programming paradigm which is popular especially among those who engineer data-intensive systems. In recent years, these areas needed to handle large amounts of persistent and shared data (something which used to occur only for business-oriented applications) and this need, together with the changes in the cost ratio between core memory and hard-disk memory provided the impetus for the development of object-oriented database systems.

The main characteristics of an object-oriented system, as identified in [6, 7], are:

1. Objects are *encapsulated*. There are three aspects to this: *control*, *insulation* and *data hiding*. The control aspect requires that

both the definition of the structure of an object and the definition of the operations allowed on the object are handled at the same time. This gives the designer control over the use of the data structure. Insulation means that the user of an object need not be concerned with the implementation of the object. In particular, the implementation of the object can be altered without affecting use of the object via the external interface. Finally, data hiding is the requirement that the internal implementation details be hidden from an external user of the object. Note that data hiding does not imply insulation. For example, C++ supports data hiding and control, but not insulation. See [2] for an example. Database systems usually provide a subschema or view mechanism. This can used to support insulation and data hiding, but not control.

2. Objects exist independently of their value; new attributes and values can be attached or removed from objects without affecting their existence. This feature of an object-oriented system is known as *object identity*. There are various levels of object identity. If the identity of an object persists only for the duration of the execution of a program, then this is the weakest form of object identity. Many database systems, especially older ones such as those based on the CODASYL model, offer the strongest form of object identity: the identity of the object persists forever (in principle).
3. The existence of a system of types and classes. While types are used for grouping together objects that have the same characteristics (as far as data structures and operational methods are concerned), classes are designed to assist the user at run-time by providing mechanisms for creating and storing objects. While both databases and programming languages provide a system of types, they are seldom compatible. This is one of the main sources of the "impedance mismatch."
4. Objects of different structures may share attributes and methods by inheriting some of their properties from more general objects to which they belong. Both semantic data models and object-oriented programming languages provide an inheritance mechanism. Unfortunately the semantics of inheritance differs markedly from one programming language to another and differs from that used in semantic data models.

5. Methods, to be applied to objects, can be defined early before their actual content is defined. The actual method to be applied to an object is determined at run-time through a mechanism called *late binding*. This approach simplifies programming activity by allowing the programmer to use diverse objects in a uniform manner. For example, a print statement that prints a single object would display the object in a manner that is appropriate for the object. A non-object-oriented language would require a much more complex construction to accomplish the same result, if it could be done at all.

3 Features

The following is a list of some of the features of the `nu&` system along with the education and research areas that each addresses.

3.1 The `mu` data model

The `mu` model was developed as general framework for object-oriented database concepts. Numerous features of database systems and object-oriented systems have already been incorporated into the model, and work continues on incorporating or expressing additional features into the model.

- A strong form of object identity is supported in which an object maintains its identity through changes of its attributes and even of its type. Objects are persistent and remain in existence forever (in principle).
- Encapsulation is supported through a view mechanism. The same mechanism can be used to provide user views (subschemas), privacy, security and data hiding.
- Both types (object specifications) and classes (object factories) are supported either separately or as a combined concept (as in C++).
- A great diversity of inheritance concepts can be expressed in the model. This includes single inheritance, multiple inheritance, multiple independent inheritance, prototype inheritance and private inheritance Prototype inheritance can also be used to support incremental versions, although this is not as efficient nor as

flexible as the direct support for versions that is also provided.

- Both overloading and late binding are supported. Late binding allows one to write procedures that exhibit a limited but very useful form of *polymorphism*. A procedure is *polymorphic* if it can act on a variety of types without being rewritten or even recompiled.
- Both ordinary attributes and function attributes or *methods* can be overloaded or late bound. This allows one to define late bound derived data, and permits more flexibility for storage strategies.
- The model allows inheritance to be either *covariant* or *contravariant*: attributes, functions and parameters can be defined to be more specific in a more specific type (covariance) or more general in a more specific type (contravariance), although the easiest one to express is covariance. Each attribute, function or parameter can be separately constrained to satisfy one of these two forms of variation or both (in which case it is said to satisfy the *exact match* condition).
- A full range of fundamental built-in data types is supported, including all the C or C++ built-in types, along with many others. Additional built-in types can be added as needed.
- Objects can examine their own type structure.
- Types can be used as attribute values of objects, and types can have objects as values of their attributes.
- Generic or parametrized types can be defined. Types are instantiated using much the same mechanism as the instantiation of objects, namely, by using a constructor. Even higher levels of generic types can be expressed in the model.
- Constraints are objects in the database. Efficiency of enforcement depends on the constraint. For example, functionality constraints are supported within the model and are implemented at a low level. For more details, see section 3.4 below.
- Several concepts of versioning can be expressed in the model. There will be direct support for at least one of these.

The `mu` model is intended to assist in research on the foundations of object-oriented systems. The fact that the model has been so successful in expressing every concept of inheritance that has ever been proposed is an indication that it is the correct model for fundamental research in this area.

The `mu` model will form the basis for the textbook on data modeling that is currently being prepared by the author and Dan Simovici.

3.2 Object-oriented semantic data models

The association model is an object-oriented semantic data model with these features:

- Support for an object-oriented entity concept called a *category*.
- Support for a relationship concept called an *association*. Names of associations can be overloaded.
- an association can also be a category. This is sometimes called *reification*. The name as an association need not be the same as its name as a category.
- The data definition language has no reserved words. This is important for developing a graphical interface.
- Inheritance can be specified using either generalization or specialization. *Specialization* is the usual method for specifying inheritance, also called *derivation*, in which a more specific type is defined using more general supertypes. *Generalization* is a technique whereby a more general type is defined from a collection of more specific types.
- Transactions, constraints and triggers can be declared and attached to categories.
- Arbitrary functionalities can be declared for each component of an association using participation numbers.
- Both value-based and object-based categories and associations can be defined.
- An attribute of a category can be single- or multi-valued, can be mandatory or optional, and can be a key for the category.
- Covering and disjointness constraints can be placed on the inheritance hierarchy.
- A category can be subclassified in one or more ways into other categories.
- Access paths to categories can be declared. This allows one to make range queries on categories having the appropriate access path.
- All the built-in fundamental types of C or C++ are supported.

In addition, fixed-length strings and variable-length text types are provided.

The association model is suitable for research on and teaching of data modeling concepts. A typical assignment in a data modeling course is to design a database for a particular task in several data models and compare them. The `nu&` system provides support for two models: the association and `mu` models. It also supports a translator from the association model to the `mu` model.

A more advanced project in the course would involve designing a new data model. The availability of source code makes it possible for a student or group of students to modify or extend the association model. Such a project could eventually lead to a research project on data modeling.

3.3 The class library generators

Several class library generators will be provided. Both C++ and Smalltalk class library generators are currently being designed.

- The data definition language is a subset of the host language. In particular, method bodies are written in a subset of the host language. The semantics of the data definition language is designed to mimic the host language as much as possible.
- Functionalities are expressed using data type operations in the host language rather than by introducing an entirely new syntax.
- Although the data model permits covariance, the data definition language permits only exact match semantics for better compatibility with the host language.
- Mimic data types are used extensively as a tool for integrating the data language in a manner that is almost transparent to the application programmer.
- Set operations are provided by using a lazy evaluation technique to implement generic objects at run-time. This permits one to do set-at-a-time operations directly in the host language.
- Structural constraints can be compiled into the class library to improve performance at the cost of reduced flexibility.

  The class library generator is amenable to a great deal of manipulation. It is expected that the various features will form

the basis for assignments and class projects in courses on data modeling as well as on object-oriented databases.

3.4 Constraint management

Constraint management involves so many aspects that a separate component is provided for studying this important research area. Functionality and other "structural" constraints are the easiest constraints to understand, and are especially important during the early design stages. Such constraints have more efficient implementations since they represent a relatively limited class of constraints. However, because structural constraints are limited, more general constraints are also necessary. The following is a partial list of the kinds of constraint that will be supported in the `nu&` system.

- *Functionality constraints* are supported directly within the data model and are implemented at a relatively low level.
- Tuple-structured, set-structured and union types are defined using *type structure constraints*. These constraints permit one to build complex objects recursively by employing powerful data structuring operations.
- *Database restructurings* are supported. A restructured database can be viewed either in its original structure or in the new structure. For example, one can "reify" an attribute into an object. The equivalence of the two views is maintained by a *restructuring constraint*. Other restructurings are associativity of union, associativity of product, distributivity of product over union and the exponential law.
- Long-duration transaction-like sessions can be defined. Such sessions are called *superactions*. The concurrency control mechanism used to implement superactions is called the *transaction option*. Transaction options are a form of dynamic constraint on the database.
- *Triggered constraints* are supported. These allow one to define complex constraints that are enforced only when triggered by an event.

Each of these classes of constraint represents an entire research area in it own right. The `nu&` system provides an environment in

which these concepts can be explored.

3.5 Implementation features

Since the `nu&` system is primarily a data modeling tool, it is expected that there will be less emphasis on the internal file structures used for low-level implementation. There are, however, some interesting features of the interface with the internal level that will be developed.

- *Name server* A service is provided for managing the names of type, attributes, functions and so on.
- *Disambiguation* A flexible mechanism for dispatching requests for data and for execution of function members is provided. The mechanism is based on the many-to-many relationship between attributes and their names which is managed by the name server.
- *File structures* File structures can be specified in the schema, using the `operator new` and `operator delete` operators.
- *Transaction management* The invocations of function members in the schema are transactions by default. To obtain transaction-like sessions that are larger than an invocation of a function member, one can define superactions [1].

4 Architecture

The architecture of `nu&` consists of the following six components. Each software module will either act upon one of these components in isolation or will serve as part of the interface between two of the components.

- **The alpha component** This component is the association model described in the preliminary report [4] on the `nu&` system. This model is an object-oriented semantic data model. The parser for this component is complete.
- **The mu component** The central data model of the `nu&` system is called the `mu` model. The model is defined in section 5 below. This model is very close to the model presented in [5] where one can find more details about its theoretical properties.
- **The chi component** The primary programming language of `nu&` is C++. The `chi` component consists of the application

programs and their class library. The language used here is C++ with no alterations or extensions other than the class library. The principal software module concerned with this component is the C++ class library generator. A working module is expected to be ready by the Spring Quarter of 1990.

- **The sigma component** A second language interface is planned which will allow application programs to be written in Smalltalk. The details of this interface will be discussed in a later report.
- **The iota component** The internal level of `nu&` is called the `iota` component. Since the `nu&` system is primarily concerned with data modeling and not file structures, we will use software already available for this component. Several object-oriented storage managers are being considered, but it is also possible to use a standard SQL-based relational database management system for this component. There are several interface modules that must be provided, one for each application language and storage manager supported. These modules are discussed in section 6.
- **The eta component** An important aspect of `nu&` is its flexible constraint capability. The `eta` component is devoted to defining and enforcing constraints.

5 The `MU` Data Model

In this section the `nu&mu` data model is introduced. We first give the theoretical foundations in section 5.1. We then specialize the model for the `nu&` system in section 5.2. This specialization involves choosing objects that serve as "built-in" types, functionality constraints and so on. The schema language is defined in section 5.3. This language is a subset of C++ with semantics chosen to match the semantics of C++ as closely as possible. The design of a data model and data definition language involves many design choices. We discuss these choices in section 5.4 and compare our approach with some of the other models that have been proposed.

5.1 Theoretical foundations

The starting point for the `mu` model is the observation that most data relationships are binary. Accordingly, a *database* is defined to be a set D of triples (o, a, p), where o and p are *objects* and a is an *attribute*

or *component*. The universe of all possible objects is denoted $\mathbf{O}$, and the universe of all possible attributes is denoted $\mathbf{A}$. If $(o, a, p) \in D$, then we say that the object o has value p for the attribute a, or more succinctly, that o has a-value p. Notice that attributes are *multivalued*: an object o can have many a-values. Constraints can be introduced that can restrict attributes to be single-valued.

The first step in database design is to identify the "entities." Consequently, the most fundamental fact about any database is that objects are somehow organized into collections or *types*. As this relationship is so basic, we distinguish it from all other attributes and gave it special features in the `mu` model. This attribute is called the *instance-of* relationship, and it will be denoted by `instance_of` or simply by $a_1 \in \mathbf{A}$. An object o is an instance of a type t in the database D when $(o, a_1, t) \in D$.

The `mu` model does not distinguish "object" from "type," so that every object is, in principle, also a type. However, to avoid logical inconsistencies (such as the "lier's paradox") that arise when objects are allowed to be instances of themselves, one must limit the pairs of objects that can be related by `instance_of`. Simply assuming that a_1 is an acyclic attribute is not enough since that does not permit one to give a satisfactory concept of a "schema" from database management or the analogous concept of "compile-time" from programming languages.

Accordingly, in the `mu` model it is assumed that there is a function called the *level function*,

$$\texttt{level} : \mathbf{O} \longrightarrow \{0, 1, 2, \ldots\}$$

that stratifies the universe of all objects into "ordinary objects" (i.e., those for which `level` has value 0), "types" (i.e., those for which `level` has value 1), "meta-types" (i.e., those for which `level` has value 2), and so on. The attribute a_1 relates objects on adjacent levels:

$$\text{if } (o, a_1, p) \in D \text{ then } \texttt{level}(p) = \texttt{level}(o) + 1.$$

Note that we refrain from formally defining the term *type* to mean an object on `level` 1, since objects on this level are just ordinary objects to the objects on `level` 2.

Like the `instance_of` attribute, every attribute in $\mathbf{A}$ relates objects that are a fixed number of levels apart: for every attribute $a \in \mathbf{A}$, there is a number $\texttt{degree}(a)$ such that

$$\text{if } (o, a, p) \in D \text{ then } \texttt{degree}(a) = \texttt{level}(o) - \texttt{level}(p).$$

In particular, `degree(instance_of)` $= -1$. Most attributes will have `degree` 0, i.e., they relate objects on the same level. The restriction of an attribute a to `level` n will be denoted by a^n.

To express the concept of a schema or type specification, instances of a type must be constrained by the attributes of the type. For example, suppose that `person` and `string` are two types. We would like to specify that an instance of `person` has a `name` which must be of type `string`. This is done by inserting the triple $(\texttt{person}, \texttt{name}, \texttt{string})$ into the database D. We call this a *type constraint*: it *constrains* any `name` value of an instance of `person` to be an instance of type `string`. Type constraints are important enough to make them an axiom of the `mu` model called the *type constraint axiom*: if $(t, a, u) \in D$ and o is an instance of t, then every a-value of o must be an instance of u.

The type constraint axiom does not quite fully capture the concept of a type specification, since it allows instances of a type to have attributes other than those attached to the type. Moreover such "extra" attributes are unconstrained. There are situations where such attributes are useful, for example, when types are considered instances of a meta-type. However, for ordinary objects one generally assumes the *type specification condition*: if $(o, a, p) \in D$, then there are types t and u such that o is an instance of t, p is an instance of u and $(t, a, u) \in D$.

The type constraint axiom and the type specification condition apply only to "ordinary" attributes. This means that they do not apply to the `instance_of` attribute, nor to the `is_a` attribute to be introduced below.

The concept of inheritance is represented with a distinguished attribute that has special features in the `mu` model. This attribute is called `is_a` or $a_0 \in \mathbf{A}$. When $(s, a_0, t) \in D$, we say that s is *derived* from t. When s and t are types, we say that s is a *subtype* of t and that t is a *supertype* of s.

The first question concerning the `is_a` attribute is whether it should be allowed to have cycles. The objects in an `is_a` cycle are equivalent to one another with respect to the data model. It is easier to allow an object to have several names than to deal with equivalence classes in a data model. It is especially easy in the `mu` model, since attributes in this model are multivalued by default. For this reason, it was decided to require acyclicity of a_0 as an axiom.

The last axiom of the `mu` model is called *object inheritance* to distinguish it from an alternative form of inheritance called *instance*

inheritance to be discussed later. In this concept of inheritance, the inheritance graph on one `level` is reproduced for each object instantiated on the next lower `level`: if $(o, a_1, s) \in D$ and $(s, a_0, t) \in D$ then there exists a unique object o' such that $(o, a_0, o') \in D$ and $(o', a_1, t) \in D$. If one borrows the terminology of C++, this can be explained as follows: if an object (o) is an instance of a derived type (s), then it is derived from a unique "base" object (o') which is an instance of the base type (s). Object inheritance is assumed as an axiom of the `mu` model.

Instance inheritance is a very different concept of inheritance. A database satisfies instance inheritance if instances of a subtype are required to be instances of any supertype: if $(o, a_1, s) \in D$ and $(s, a_0, t) \in D$ then $(o, a_1, t) \in D$. If one thinks of types as collections of objects, then this condition is purely set-theoretic: supertypes contain subtypes. Instance inheritance is the intuitive picture one has in mind when one is thinking about any inheritance concept. However, as discussed in [3], it is object inheritance that is actually provided by programming languages and databases. For this reason the `mu` model uses object inheritance rather than instance inheritance as its inheritance concept.

So far nothing in the axioms for the `mu` model prevents an object from being an instance of an arbitrary set of types (or no type at all). This is reasonable on higher levels, but on `level` 0 and possibly also `level` 1, it is desirable to be more restrictive. A database is *uniquely typed* on `level` n if every object on this `level` is an instance of exactly one type. In other words, the attribute $\texttt{instance\_of}^n$ is single-valued. We will assume this condition on `level`s 0 and 1.

The inheritance mechanism of the `mu` model supports a number of important properties. The first property is *substitutability*: an instance of a subtype can be regarded as (or substituted for) an instance of a supertype. *Late binding* on the other hand states that a derived object still "remembers" that it is an instance of a subtype even when it is being regarded as a base object. A database D is *transitive* if for any objects or types o, p and q, $(o, a_0, p) \in D$ and $(p, a_0, q) \in D$ implies $(o, a_0, q) \in D$. Far from being a trivial property, transitivity is the sole difference between "virtual" inheritance and "multiple independent" inheritance in C++. For a more complete discussion of these properties and others, see [3].

If a type t has an attribute a, but an instance o of t does not have this attribute, then we say that o has *null value* for attribute a. This

is the only concept of null directly supported within the model.

5.2 Special objects

To complete the description of the `mu` model one must distinguish a number of *built-in* objects. For example, one must specify built-in types such as `int`, `string` and so on. The built-in types represent the starting point for building more complex types. The choice of which types should be built-in is an important design decision of any data model, and the ones that are provided match the built-in types of C or C++, with some additional ones such as `string` that have been recognized as generally useful in a wide range of application areas.

Dual to the concept of built-in type is the concept of a built-in constraint. Some of these are important enough to deserve being distinguished as axioms, and are imposed universally. Others are imposed more selectively. To allow constraints to be manipulated like any data in the database, they are given object status. Like objects in general, complex constraints can be built from more elementary constraints. In particular, this means that some constraints will be built-in.

The specialization satisfies the following conditions:

SDB1 (level limit) All objects have level 0, 1 or 2; all proper attributes have degree 0 or 1.

SDB2 (meta-types) There are four objects on level 2: `TYPE`, `CONSTRAINT`, `FUNCT` and `VIEW`. Both `FUNCT` and `VIEW` are subtypes of `CONSTRAINT`.

SDB3 (constituencies) The database is uniquely typed on level 1 and uniquely typed and type specified on level 0.

SDB4 (constraints) A instance of `CONSTRAINT` has no instances.

SDB5 (functionalities) There are five distinguished objects in `FUNCT`: `one-to-one`, `one-to-many`, `many-to-one`, `mandatory-many-to-one` and `many-to-many`.

Both views and functionalities are specified the same way: an attribute a of a type t is in a view v (or has functionality f) if $(v, a, t) \in D$ (or $(f, a, t) \in D$, respectively). Since the various functionalities have syntactic consequences in C++, it is currently not feasible to allow a subtype to override the functionality of an attribute.

5.3 The data definition language

The data definition language for `nu&mu` is a subset of C++. As we acquire more experience, we plan to increase the number of C++

constructs that can be accommodated. However, it is unlikely that the subset will be much larger than that currently allowed, since `nu&mu` must be compatible with other languages as well.

A `mu` schema consists of a collection of units (called *chapters*), each of which is one of the following: a class definition, using the reserved word `struct`; a member function definition; or a global variable. A class definition, in turn, consists of a collection of components, each of which is one of the following: a data member; a function member; a constructor; a destructor; a conversion; or an operator (except `new` and `delete`). One may specify visibility of members, using `public`, `protected`, `private` and `friend class`. Other constructions for specifying visibility are not yet supported.

The only kind of inheritance currently supported is virtual public inheritance. If the reserved word `virtual` is omitted, it is assumed, but a warning is given. If the reserved word `private` is used, a syntax error occurs.

The type of a data member, function member, conversion, operator, function parameter or global variable must be one of the following:

1. One of the fundamental built-in data types.
2. One of the user-defined data types in the schema.
3. A type obtained from one of the types in (1) or (2) above by using one of the following type constructors: ordinary type (`<type>`), pointer (`<type>*`), reference (`<type>&`), unbounded array (`<type>[]`) or unbounded array of pointers (`<type>(*)[]`). In the fifth type constructor, the parentheses may be omitted, but a warning is given in this case.

The following is a partial list of the most prominent features of C++ that are currently not included. Any of these is an excellent research topic. Some are suitable for a course project, while others are material sufficient for a Ph.D. thesis.

- Private derivation.
- Non-virtual derivation.
- Named and unnamed unions.
- Static, const and volatile data and function members.
- Typedefs.
- Pointers to members, especially member functions.
- Register, static and volatile variables in function bodies.

- Friend functions.
- Allowing type names to be hidden.
- Enum types.
- Lazy prototyping.
- Initializer blocks.
- External data or functions accessed from the schema.
- Bounded arrays as type constructors.

5.4 Design choices

The `mu` model differs considerably from other models that have been proposed. In this section we discuss some of the design choices that led to the model as well as compare the model to other models. The most important predecessors to the `mu` model are the Format model of Hull and Yap [8], the LDM model of Kuper and Vardi [9] and the O_2 model of the Altaïr group [10].

One of the more dramatic ways in which `mu` differs from other models is the use of explicit typing and inheritance. In O_2, for example, types are abstractions determined by the structure of objects. Objects have the same type if they have compatible structures in a suitable sense. The type of an object is *inferred*. Inheritance is also determined by inference. By contrast, the `mu` model requires that objects be explicitly specified as instances of a type. Two objects could have exactly the same attributes and even attribute values without being instances of the same type. Similarly, type inheritance is explicit. We noted that declarative programming languages like C++ and Smalltalk use explicit typing and inheritance. For the sake of reducing the impedance mismatch, we decided to be explicit.

One of the features of O_2 is that objects can have attributes not specified by their type. This feature is allowed only on positive levels in the `mu` model because it clashes with declarative object-oriented programming languages which do not support such flexibility. The feature is convenient on level 1, for otherwise the meta-type `TYPE` would have to have every attribute of any type. The feature is essential on level 2, for without it the model would force one to have an "infinite tower" of meta-types on higher and higher levels.

Another curious feature of O_2 is that the inheritance graph can have cycles. As this has no analog in declarative programming languages or in semantic data models, we chose not to allow it.

Another significant difference between the `mu` model and other models is the lack of an explicit null object or value. The disadvantage of null objects or values is that the axioms of the model must make constant reference to these special cases. This leads to a nonuniform treatment that is much harder to understand. Except for the two distinguished attributes, the axioms of the `mu` model do not recognize any other distinguished attributes and do not recognized any distinguished objects at all. In general, it is much better to have a uniform treatment (i.e., without special cases) than to introduce distinguished cases. This is what makes late binding such a powerful programming technique, and, by analogy, we felt that it would also help strengthen the `mu` model which serves as the foundation of the entire `nu&` system.

Another problem with null objects or values is that the semantics can be confusing and ambiguous. For example, in O_2 there is a NIL value. This value is distinguishable from the empty set and the empty tuple (having no attributes). Experience with data modeling suggests that such distinctions are artificial and confusing. Even in O_2 it is not clear what these distinctions are supposed to be. For example, there are four distinguishable possibilities for an object to have a "nil" value for an attribute: it might not have the attribute at all, the attribute might be an object with NIL value, the attribute might be an object with an empty set-structured value and finally the attribute might be an object with an empty tuple-structured value.

We decided to avoid such semantic subtleties by having just one concept of null value. An object has a null value for an attribute if it does not have that attribute. That the object "should" have the attribute is determined by the type of the object (which is specified explicitly rather than being inferred by the structure of the object). Thus the empty set of values for an attribute is the same as not having a value which, in turn, is the same as having the null value.

Of course, there is nothing to stop a designer from introducing "null" objects and incorporating their semantics into methods. Within the `mu` model, however, these would be objects like any other with no special privileges.

Another interesting design decision was that every attribute has the potential to be multi-valued, and it will be so by default unless constrained to be otherwise. This was done to allow one to perform schema updates in the same manner as other updates. In fact, the axioms do not distinguish any level. Of course, the usual functionality constraints (such as single-valuedness) are provided as special

objects to give a designer the option to improve performance of the implementation at the cost of making it more difficult to update the schema. This performance/flexibility trade-off is a well-known one. Most database systems, such as SQL-based relational systems build this choice into the database management system rather than deferring the choice until final implementation.

Each type t in a `mu` database is automatically a "tuple-structured" type, since it can have any number of attributes. Each of these attributes, in turn, can have a set of types as its "value." Say, for example, that attribute a has values u and v. An instance of the type is also tuple-structured, and each attribute value is constrained to be an instance of an anonymous union type. For example, a value of attribute a must be an instance of either u or v (or both). Finally, each attribute of an object can have a set of values thereby making each attribute "set-structured." In other words, the `mu` model provides both "tuple-structured" and "set-structured" data as in the Format, LDM and O_2 models. It also provides for unions which are part of the Format model but not the LDM or O_2 models.

One significant mismatch between declarative programming languages and data languages is the lack of true set structures within programming languages. Declarative programming languages generally have facilities for handling tuple-structures and unions, but they provide arrays rather than set-structures. Note that the lack of a set structure occurs only at the lowest object level. At the type level, unions are set structured as is multiple inheritance. The array concept is also an incomplete one, since arrays cannot in general be passed as a (value) parameter. The decision to provide set structures in the data model in lieu of an array or list structure was a trade-off. On the one hand, it leads to a mismatch problem with programming languages. On the other hand, incorporating an array or list structure in the data model leads to a more complex data model which is less powerful and whose semantics differs from level to level. Fortunately nearly every class library provides some form of set structure, so that if sets are not explicitly part of an object-oriented programming language at its lowest object level, then they are easily available within the class library. Accordingly, we chose to use set structures at every object level. Other collective structures, such as lists and arrays will have to be synthesized. Only time will tell if this is the right decision.

The decision to have a uniform treatment of objects, types and meta-types led to the concept of a parametrized type as simply a

type on a higher level. The parametrized type instantiates ordinary types by using a type constructor exactly as ordinary objects are instantiated using a constructor. One big difference between types and objects is that one may assume that every object is an instance of some type, but one normally does not assume that every type is an instantiation of some parametrized type. This is one of the reasons why the axioms are somewhat loose about type constraints, and why it is necessary to have axiom SDB3. We wanted the data model to deal with all levels uniformly, but since most languages and database systems are not uniform in this respect, this nonuniformity had to be stated somewhere.

We could provide parametrized types within `nu&` immediately, but we would like to be compatible with the notation being developed for parametrized types (templates) in C++ which are still in development.

6 The Internal Level

The most interesting aspect of the interface to the internal level of an object-oriented system are the mechanisms for overloading, overriding, disambiguation and late binding (or dispatching). The key to these mechanisms is a flexible name facility, called the `name server`. Instead of regarding attributes as being ambiguous, we take the point of view that attributes in themselves are never ambiguous. Ambiguity and hence the need for the concepts of overloading, overriding and disambiguation arises from the fact that the relationship between components and their names is many-to-many: each component has many names and different components can have the same name. Moreover every context has a different view of this many-to-many relationship. Furthermore the disambiguation mechanism is both determined by this view and helps to determine it.

References

[1] K. Baclawski. Database transaction options. Technical Report NU-CCS-86-5, Northeastern University, College of Computer Science, 1986.

[2] K. Baclawski. Mimicking data types in an object-oriented programming language. Technical Report NU-CCS-89-33, Northeastern University, College of Computer Science, 1989.

[3] K. Baclawski. The structural semantics of inheritance. Technical Report NU-CCS-90-19, Northeastern University, College of Computer Science, 1990.

[4] K. Baclawski, T. Mark, R. Newby, and R. Ramachandran. The `nu&` object-oriented semantic data modeling tool: preliminary report. Technical Report NU-CCS-90-17, Northeastern University, College of Computer Science, 1989.

[5] K. Baclawski and D. Simovici. An algebraic approach to databases with complex objects. *Information Systems*, 17(1):33–47, 1992.

[6] F. Bancilhon. Object-oriented database systems. In *Proc. of the Sympos. on Principles of Database Systems*, pages 152–162, Austin, Texas, March 1988.

[7] J. Banerjee, H. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H. Kim. Data model issues for object-oriented applications. *ACM Trans. on Office Information Systems*, 5(1):3–26, 1987.

[8] R. Hull and C. Yap. The format model: a theory of database organization. *J. ACM*, 31:518–537, 1984.

[9] G. Kuper and M. Vardi. A new approach to database logic. In *Proc. 3^{rd} ACM Sympos. on Principles of Database Systems*, pages 86–96, 1984.

[10] C. Lécluse, P. Richard, and F. Velez. O_2, an object-oriented data model. In *Proc. SIGMOD*, pages 424–433, 1988.