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

Kenneth Baclawski

Timothy Mark

Robert Newby

Ramanathan Ramachandran

College of Computer Science
Northeastern University
Boston, Massachusetts 02115

Technical Report NU-CCS-90-17

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 first of a series of reports on the nu& system.

Version 1.3 September, 1989

The authors were 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 first report on the nu& system. As such, it represents the status of the project during the preliminary design phase.

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 [Bancilhon 88], [Banerjee et al. 87], are:

1. Objects are *encapsulated*. This means that the operations performable on an object are handled along with the data structure, that the designer of the structure can control the contexts within which the data structure is used, and that the external user of the object is insulated from the internal implementation details of the structure. The term *data hiding* is used when the internal details of a data structure are hidden from a user.
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*.
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.
4. Objects of different structures may share attributes and methods by inheriting some of their properties from more general objects to which they belong.
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.

Engineering design or CAD database systems present a difficult challenge to traditional database management tools. The most obvious problem is that CAD databases require geometric structures to be represented and manipulated. This problem has been considered extensively in the research literature, and has spurred an interest in database systems that support complex objects. Object-oriented databases are a natural setting for complex objects.

However, there are challenges that are less obvious but no less important. One of these concerns consistency constraints. In traditional databases, consistency constraints are relatively simple and entirely static. By contrast, in an engineering design database the constraints are complex and are also enforced or "triggered" dynamically. Large designs require a long time to create, and early versions can be incomplete and only partially consistent. What is needed is a language for dynamic consistency constraints that allows consistency constraints to be gradually enforced as the design is being built. This is similar to syntax-sensitive editors that perform syntax checking and even compilation of software as it is being written.

Yet another issue concerns concurrent access to engineering designs as they are being produced. For example, if an editing session is regarded as a transaction, then a system failure would result in the entire session being undone. Furthermore, other transactions that require use of the design would have no way of distinguishing whether the design is being used for a long or a short time. In fact, the same problems occur in any editing system. Transaction concurrency control is inherently short-term and volatile. A new kind of lock is needed, one that is persistent, long-term and clearly distinguishable from the ordinary concurrency control mechanism.

Computer-Integrated Manufacturing presents many of the same problems as Engineering Design, but there are still other problems. These systems are updated frequently and demand real-time performance. Furthermore, a CIM system is intrinsically distributed and heterogeneous. This puts a greater burden on the database system to enforce integrity constraints and concurrency control and to provide recovery from failures.

Ever since the first programming language, FORTRAN, was developed, programming languages have been driven by the need for higher-level semantics while at the same time retaining the efficiency of lower-level structures. The current work in object-oriented programming languages can be seen as part of this historical trend. Data models have followed a similar evolution. Early data models offered little very little support for higher-level semantics as well as little independence from the underlying file structures. The relational model, as embodied in the ANSI standard for the SQL language, offers both increased data independence and cleaner (if not higher-level) semantics.

While both programming languages and data models have been evolving toward implementation independence and higher-level semantics, they have been doing so independently. For example, programming languages have type systems that are very different from those in data models. As a result it is very difficult to integrate a programming language with a

data language. This problem has been called the "impedance mismatch" between the two kinds of language [Bancilhon 88]. The impedance mismatch between the relational model and object-oriented programming languages is especially troublesome, since the relational model is a structural rather than an object-oriented model.

3. Objectives of the NU& System.

Given the very different nature of database management and programming languages, it is no surprise that the database management community has evolved a very different culture than the programming language community. Yet the two are fundamentally related, and practitioners in both fields have frequently expressed the need for more compatibility. In the 60's this feeling led to the development of the CODASYL data model which is closely tied with the COBOL programming language. Subsequently, the two communities diverged, and recently the need for compatibility has again been expressed, under the catchy slogan "reduction of the impedance mismatch." *Plus ça change, plus la mèmé chose.*

While developing a data modeling tool with many new "bells and whistles" is seductive, and everyone seems to be doing it, the reduction of the impedance mismatch would have a much greater impact in the long run. For this reason, we chose to make this our most important objective. This means that even if a feature is desirable, it might not be included if it detracts from the impedance mismatch.

The following are some of the other research problems being studied:

1. Integrating complex objects, views, early and late binding with the is-a and instance-of relationships among objects and types.
2. Integrating attribute and behavioral inheritance.
3. Studying views, database restructuring and schema integration in the object-oriented paradigm.
4. Developing a model in which types, methods, transactions and constraints are all objects, allowing a more uniform treatment of data and metadata.
5. Introducing more flexible approaches to concurrency control in which long-duration sessions having user interaction behave like transactions.
6. Implementation issues in distributed, high-performance object-oriented database systems.

We plan to make use of an earlier system, the Network Emulation Tool [Baclawski 87], that has seen extensive, successful use in courses at Northeastern University and elsewhere. This system has been reprogrammed in the object-oriented programming paradigm using C++ [Baclawski, Mark, Ramachandran 89].

Although there are commercial object-oriented database systems now on the market and there are many research prototypes, there is considerable disagreement about how such a system should be designed and what services should be provided. The nu& system will focus on database design and on concurrency control and recovery services.

In contrast to many existing object-oriented database systems which add database system capabilities to existing object-oriented programming languages, the nu& system will be

capable of dealing with a variety of object-oriented programming languages. The two that will be used first are C++ and SmallTalk.

One of the difficulties encountered by anyone teaching data modeling is that most existing models are research prototypes that are not available either as public domain software or as commercial products. Even if the software were available commercially, it is unlikely that the source code would be provided. The nu& system is intended to eliminate this gap. The nu& system will contain parsers, data model translators and class library generators for experimenting with semantic and object-oriented data models as well as for integrating data models with C++ and SmallTalk.

4. The Association Model and Data Definition Language

While the full architecture of the nu& system remains to be designed, some components are already in development. The initial focus was on designing a basic data model which would have interfaces with programming languages in one direction, with the internal storage structures in another, and with various higher-level semantic data models in a third. The translation from basic data model to programming language is the *class library generator*. This report is concerned with a proposed data model called the *association model* and with the class library generator from the association model and C++.

The data definition language for the association model uses standard tools for lexical scanning and parsing. Tokens are either punctuation marks (colon, semicolon, etc.), identifiers (defined as in C) or a number (string of digits). The lexical scanner supports C++ style comments. The scanner understands the output of the C preprocesser.

The grammar is unusual in having no reserved words. There are special words used by the grammar, but any of these may also be used as names in any one of the name spaces. The reason for this unusual feature is to accommodate a graphical interface. The DDL would then be the internal language for storing the schema. It would be unnatural to have reserved words in such an interface, so it was decided not to require them in the textual version.

There are 11 kinds of name space. The context determines the name space of an identifier. A given identifier can appear in any number of name spaces. Although there was no compelling reason for having so many name spaces, neither was there a compelling argument for combining any of the name spaces. The name spaces are as follows:

1. `schema_name`	There is exactly one of these.
2. `category_name`	This is the name of a category, globally unique.
3. `association_name`	These may be overloaded.
4. `attribute_name`	Separate name space for every category.
5. `index_name`	Separate name space for every category.
6. `subclassification_name`	Separate name space for every category.
7. `transaction_name`	Separate name space for every category.
8. `constraint_name`	Separate name space for every category.
9. `enumerator_name`	These must be globally unique.
10. `class_name`	These must be globally unique.
11. `trigger_name`	These must be globally unique.

The grammar of the association model is written in yacc notation and is an unambiguous grammar. Tokens are written in uppercase and non-terminals in lowercase. When an identifier is part of a reduction that defines the identifier within one of the name spaces, it is inserted into the appropriate name space with the procedure `define_name`. When a reduction uses an identifier that should be defined elsewhere, then the identifier is implicitly declared to be within one of the name spaces with the procedure `declare_name`. When parsing is completed, the symbol table is checked to be sure that within each name space every identifier that is declared or defined is defined exactly once. The procedure `make_id_token` changes one of the special tokens into an identifier.

```
schema
:  SCHEMA id LBRACE schema_component_list RBRACE
     { reduction->define_name (2, schema_name); }
;
schema_component_list
:  schema_component_list schema_component_slot
|
;
schema_component_slot
:  schema_component
|  SEMICOLON
;
schema_component
:  CATEGORY id category_def
     { reduction->define_name (2, category_name); }
|  ASSOCIATION id assoc_def
     { reduction->define_name (2, association_name); }
|  ASSOCIATION id CATEGORY id category_def assoc_def
     { reduction->define_name (2, association_name);
       reduction->define_name (4, category_name); }
;
```

A *schema* consists of a sequence of schema components. Each schema component defines either a category or an association. A *category* is also called a *class*, *entity* or *type*. An *association* is also called a *relationship*. An association can also be a category, and its name as an association need not be the same as its name as a category.

```
category_def
:  LBRACE category_component_list RBRACE
|  COLON category_name_list LBRACE category_component_list rbrace_token
;
category_component_list
:  category_component
|  category_component SEMICOLON
|  category_component SEMICOLON category_component_list
```

```
;
category_component
:  attribute
|  index
|  subclassification
|  transaction
|  constraint
;
```

A category can inherit from a list of more general categories (multiple inheritance). In C++ terminology, the inheritance is virtual: if `B` and `C` both inherit from `A` and if `D` inherits from both `B` and `C`, then there will only be one `A` object contained within `D`. The definition of a category consists of a list of components, as in the concept of a *class* in C++.

Each category corresponds to a set of classes generated by the class library generator. There is one class for a collection of objects of the category, one for a single (generic) object in one such collection as well as one for iterating over a collection of objects. Still other classes are generated for various specialized purposes, such as indexing.

```
assoc_def
:  OF id participation WITH LBRACE category_participation_list RBRACE
     { reduction->declare_name (2, category_name); }
|  OF id participation WITH id participation
     { reduction->declare_name (2, category_name);
       reduction->declare_name (5, category_name); }
;
category_participation_list
:  id participation
     { reduction->declare_name (1, category_name); }
|  id participation COMMA
     { reduction->declare_name (1, category_name); }
|  category_participation_list id participation
     { reduction->declare_name (2, category_name); }
|  category_participation_list id participation COMMA
     { reduction->declare_name (2, category_name); }
;
participation
:  NUMBER TO NUMBER
|  NUMBER TO INFINITE
|
;
```

An association is a relationship among two or more categories. The categories need not be distinct. The first category is special and is called the *anchor* of the association. Association names may be overloaded, the only restriction being that two associations having the same name must have either different anchors or a different set of associated categories. The order of the associated categories is not significant (except that the anchor category is

distinguished).

An association is normally value-based: an instance is determined by the objects that participate in it. In other words, an association is normally a relation as in the relational model. However, if an association is declared to be also a category, then the association is object-based: there may be distinct instances having the same participating objects, and the identity of an instance does not change when modifications are made.

A *participation* specifies the least and greatest number of times that an object must participate in the association. The only `NUMBER`s allowed are zero and one, and there are only four meaningful participations: 0 to 1, 0 to infinite, 1 to 1 and 1 to infinite. These correspond to the constraints "at most once," "no constraint," "exactly once," and "at least once," respectively.

From the point of view of the class library generator, each association has a corresponding class as well as a function member in the classes associated with the anchor category. This function member takes values in the class corresponding to the association.

```
attribute
:  attr_type id attr_constraint
     { reduction->declare_name (2, attribute_name); }
|  KEY attr_type id attr_constraint
     { reduction->declare_name (3, attribute_name); }
|  MULTIVALUED attr_type id attr_constraint
     { reduction->declare_name (3, attribute_name); }
|  MANDATORY attr_type id attr_constraint
     { reduction->declare_name (3, attribute_name); }
|  MULTIVALUED MANDATORY attr_type id attr_constraint
     { reduction->declare_name (4, attribute_name); }
|  MANDATORY MULTIVALUED attr_type id attr_constraint
     { reduction->declare_name (4, attribute_name); }
;
index
:  INDEX id index_def
     { reduction->declare_name (2, index_name); }
|  UNIQUE INDEX id index_def
     { reduction->declare_name (3, index_name); }
;
subclassification
:  SUBCLASSIFICATION id LBRACE category_name_list RBRACE
     { reduction->declare_name (2, subclassification_name); }
|  TOTAL SUBCLASSIFICATION id LBRACE category_name_list RBRACE
     { reduction->declare_name (3, subclassification_name); }
|  SUBCLASSIFICATION id DISJOINT LBRACE category_name_list RBRACE
     { reduction->declare_name (2, subclassification_name); }
|  TOTAL SUBCLASSIFICATION id DISJOINT LBRACE category_name_list RBRACE
     { reduction->declare_name (3, subclassification_name); }
```

```
;
transaction
:  TRANSACTION id { reduction->declare_name (2, transaction_name); }
;
constraint
:  create constraint_specifier
|  modify constraint_specifier
|  delete constraint_specifier
;
attr_constraint
:  CONSTRAINED constraint_specifier
|  CONSTRAINED BY constraint_specifier
|
;
constraint_specifier
:  constraint_function
|  constraint_function TRIGGERED id
     { reduction->declare_name (3, trigger_name); }
|  constraint_function TRIGGERED BY id
     { reduction->declare_name (4, trigger_name); }
;
```

A component of a category may be an attribute, index, subclassification, transaction or constraint. An *attribute* is the traditional concept of a field of a record or data member of a class, except that they are allowed to be *multivalued*. An attribute is *mandatory* if it cannot be null. In the multivalued case, this means that the set of values must be nonempty. An attribute can also be declared to be a *key* for the category. A key attribute is single-valued and cannot be null. When a category has a key attribute, the category becomes a value-based category.

An *index* is an access path for the category. There need not be an index file corresponding to an index, although such a file would be a natural way to provide this feature. Having an index means that objects in the category can be selected based on the value of attributes in the index. A unique index is the same as a key when there is only one attribute in the index. As with a key attribute, a category becomes value-based when it has a unique index.

A *subclassification* specifies a set of distinct subcategories which inherit from the given category. Each subcategory in the set must inherit from the given category. Subclassifications allow one to specify disjointness and covering constraints. A set of subcategories *covers* a category if every object of the category is an object of at least one subcategory in the set. When a subclassification is *total*, its set of subcategories covers the category.

A *transaction* is a procedure that is executed atomically. All the elementary operations performed on objects in the database are atomic. It is only when a set of these must be performed atomically that it is necessary to declare them a transaction.

A *constraint* is a predicate (i.e., a boolean function without side-effects) on the database. Constraints can be specified to be tested whenever an object is constructed (a *create*

constraint), whenever an object is deallocated (a *delete constraint*) or whenever any object in the category is changed (a *modify constraint*). Constraints can also be attached to individual attributes, in which case they are tested only when the attributed is modified.

Since constraints may be time-consuming to evaluate, a two-step method can be specified. A relatively easy to evaluate *trigger* is evaluated whenever the constraint should be checked. Only if the trigger evaluates to true is the constraint checked.

```
create
:  CREATE
|  CREATE CONSTRAINT
;
modify
:  MODIFY
|  MODIFY CONSTRAINT
;
delete
:  DELETE
|  DELETE CONSTRAINT
;
constraint_function
:  GLOBAL id { reduction->declare_name (2, constraint_name); }
|  FRIEND id { reduction->declare_name (2, constraint_name); }
|  MEMBER id { reduction->declare_name (2, constraint_name); }
;
```

Constraint functions may be global functions or they may be associated with the category where they are used. The terminology is borrowed from C++.

```
attr_type
:  integral_type
|  UNSIGNED integral_type
|  base_type
|  STRING NUMBER
|  ENUM LBRACE enum_list RBRACE
|  CLASS id { reduction->declare_name (2, class_name); }
;
integral_type
:  CHAR
|  INT
|  SHORT
|  LONG
;
base_type
:  BOOLEAN
|  FLOAT
```

```
|  DOUBLE
|  TEXT
;
enum_list
:  enum_list COMMA enumerator
|  enumerator
;
enumerator
:  id { reduction->declare_name (1, enumerator_name); }
|
;
```

The built-in types are as in C or C++, except that string and text types are considered to be built-in. A *string* is a fixed-length array of characters, while *text* is a variable-length sequence of characters.

```
index_def
:  index_component_list
|  LBRACE index_component_list RBRACE
|  LBRACE index_component_list COMMA RBRACE
;
index_component_list
:  index_component
|  index_component_list COMMA index_component
;
index_component
:  id { reduction->declare_name (1, attribute_name); }
|  DESCENDING id { reduction->declare_name (2, attribute_name); }
;
```

An index is defined by specifying a sequence of attributes. Each attribute can be ordered in ascending or descending order. This applies only to those types for which an order is defined. If a type does not have an order, then the attribute can still appear in the index, but range queries on that attribute are not defined.

```
category_name_list
:  id { reduction->attach_name (1, category_name); }
|  id COMMA { reduction->attach_name (1, category_name); }
|  category_name_list id { reduction->attach_name (2, category_name); }
|  category_name_list id COMMA { reduction->attach_name (2, category_name);}
;
id
: ID                                  | SCHEMA            { goto make_id; }
| CATEGORY          { goto make_id; } | ASSOCIATION       { goto make_id; }
| OF                { goto make_id; } | WITH              { goto make_id; }
| TO                { goto make_id; } | INDEX             { goto make_id; }
```

```
| SUBCLASSIFICATION { goto make_id; } | TRANSACTION       { goto make_id; }
| CREATE            { goto make_id; } | MODIFY            { goto make_id; }
| DELETE            { goto make_id; } | CONSTRAINT        { goto make_id; }
| KEY               { goto make_id; } | MULTIVALUED       { goto make_id; }
| MANDATORY         { goto make_id; } | CONSTRAINED       { goto make_id; }
| BY                { goto make_id; } | UNIQUE            { goto make_id; }
| DESCENDING        { goto make_id; } | DISJOINT          { goto make_id; }
| TOTAL             { goto make_id; } | TRIGGERED         { goto make_id; }
| GLOBAL            { goto make_id; } | FRIEND            { goto make_id; }
| MEMBER            { goto make_id; } | CHAR              { goto make_id; }
| INT               { goto make_id; } | SHORT             { goto make_id; }
| LONG              { goto make_id; } | FLOAT             { goto make_id; }
| DOUBLE            { goto make_id; } | BOOLEAN           { goto make_id; }
| TEXT              { goto make_id; } | UNSIGNED          { goto make_id; }
| ENUM              { goto make_id; } | STRING            { goto make_id; }
| CLASS             { make_id: reduction->make_id_token(); }
;
```

The remaining reductions of the grammar specify that a list of categories is a sequence of category names optionally separated by commas, and that none of the special tokens are reserved. It is more difficult to construct an unambiguous grammar when there are no reserved words, but we felt that compatibility with a graphical interface was important.

5. Evaluation of the Association Model

The association model established that it is possible to develop an object-oriented semantic data model. This is useful in itself as well as furthering the overall objectives of nu&. Although at one time we considered using this model as the basic model of nu&, it soon became clear that it was not suitable for this role, and another one was designed. The association model will serve as the higher-level semantic data model for nu&, and will interact with the basic data model via a translator. The basic model will interact with C++ and SmallTalk via class library generators.

The rest of this section is a discussion of the shortcomings of the association model that disqualify it as a candidate for the basic model. Few of these criticisms apply to the model in its current role as higher-level semantic data model for nu&.

Criticism of any object-oriented data model must proceed systematically since there are so many features to consider. We begin with a discussion of the impedance mismatch. We then criticize the object-oriented aspects of the model. Bancilhon's discussion [Bancilhon 88] of the main features of an object-oriented system will form the basis this discussion. Finally, we mention the database management aspects.

The most important consideration is the impedance mismatch. Although we clearly were making progress on this, there was clearly a one-sidedness about the approach: the data model was translated into C++ and not the other way around. It would be much better to have a data model that would support a reasonably general C++ (or SmallTalk) class. Moreover the model should allow for incremental improvements that would progressively

expand the kinds of class supported. Conversely, nearly any possible schema in the data model should be expressible in C++ or SmallTalk. These are ambitious goals and represent very hard problems, but they are well worth tackling.

Consider next the object-oriented aspects of the model. The first important feature is encapsulation. In this respect, the association model is inadequate. There is no technique in the model for hiding any component of a category. However, this could be added by specifying public and private components as in C++. The fact that categories only have data members and related components (indices, constraints and triggers) is also a failure of encapsulation: no allowance is made for specifying general function members. The concept of transaction was supposed to fulfill this requirement, but it was never developed well enough for this.

The second feature is object identity. This is supported very well, with both object-based and value-based categories and associations provided. The association model also provides a system of types and classes, and multiple inheritance is fully supported.

The last important feature is late binding. This is not provided. In fact, since function members are not supported, the concept of late binding cannot even be expressed in this model.

A further disadvantage of the association model is its complexity. There are too many ways to accomplish the same thing. However, this is not that significant an issue, since the association model is less complex than most other semantic data models despite the many additional features, such as object identity, that it provides.

From the point of view of database management, one criticism of the association model is that it does not include a view mechanism. This is not that difficult to provide. In fact, the subclassification concept was introduced primarily to aid in defining views: only one subclassification of a category would normally be visible in a given view.

Another difficulty with the association model is the fact that categories cannot have named roles within an association. This makes it difficult for the same category to be used more than once as a non-anchor category. This can be remedied easily by adding optional role names to the definition of an association.

6. Conclusion and Future Work on the NU& System

This report describes the current status of the nu& object-oriented semantic data modeling tool. The objectives and initial design decisions have been made, and a detailed discussion of these has been given.

The only component of the system that is complete at this time is a semantic data model called the association model. This model is an appealing model which serves as a bridge between the semantic data models and the object-oriented data models, and will serve as the higher-level semantic data model of the nu& system.

The next report will be made in late 1989 or early 1990. This report will introduce the basic model as well as give more details about the architecture of the system. It is planned to have a working system ready for classroom use in the Spring Quarter of 1990. At that

point, the third report should be ready. The third report will serve as the user manual for the students in the class.

References

[Baclawski 87] K. BACLAWSKI, A network emulation tool, *Proc. 1987 Symp. on the Simulation of Computer Networks*, pages 198–206, August, 1987.

[Baclawski, Mark, Ramachandran 89] K. BACLAWSKI, T. MARK AND R. RAMACHANDRAN, ONET: an object-oriented network emulation tool, Northeastern University, College of Computer Science, 1989.

[Bancilhon 88] F. BANCILHON, Object-oriented database systems, *Proc. of the Sympos. on Principles of Database Systems*, Austin, Texas, pages 152–162, March, 1988.

[Banerjee et al. 87] 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.