DATA MODELING IN DE*LAB*

Yannis E. Ioannidis - Miron Livny
Computer Sciences Department, University of Wisconsin - Madison

As the size and complexity of processing and manufacturing systems increases, the need for Database Management Systems (DBMS) that meet the special needs of studies that experiment with such systems becomes more current. System analysts who study the performance of modern processing systems have to manipulate large amounts of data in order to profile the behavior of the system. They have to identify the relationship between the properties of a compound system and a wide spectrum of performance metrics. In a recent study in which we have analyzed a set of distributed concurrency control algorithms, we performed more than 1400 simulation experiments. Each experiment was characterized by more than 6000 input parameters and generated more than 400 output values. It is thus clear that powerful means for defining the structure and properties of complex systems are needed, as well as efficient tools to retrieve the data accumulated in the course of the study. We are currently engaged in an effort to develop and implement the **DE*LAB*** ***simulation laboratory*** that aims to provide such means and tools for simulation studies.

The goal of the first phase of this effort was to design and implement a simulation language. It ended in 1986 when the **DE*NET*** (Discrete Event NETwork) simulation language became operational. The language is based on the concept of Discrete Event System Specifications (DEVS). It views the simulator as a collection of self contained objects that communicate via Discrete Event Connectors that provide a unified synchronization protocol. In the past two years the language has been used in a number of real life studies. It was used to simulate distributed processing environments, communication protocols, and production lines. Several tools have been developed around the language. All tools adhere to the same modeling methodology and thus create a cohesive simulation environment.

In the second phase of the **DE*LAB*** project we have been addressing the data management problem. **DE*NET*** has been interfaced to a special purpose relational DBMS that can store descriptions of simulation runs and provides access to the stored data. Based on our experience with this DBMS, we have reached the conclusion that system analysts need to be provided with a view of the data that differs from the way the **DE*NET*** program views the data, and thus decided to develop a data model that meets their needs. The **MOOSE** data model, which is the result of this effort, has an *object oriented* flavor. It was developed with the guidance of potential users and was tested on a number of real life simulation studies.

Although the conception of **MOOSE** was motivated by the specific needs of a simulation laboratory, we believe that it addresses the representational needs of many other environments. We have decided to support the notion of an *object*. Every object is assigned a unique identifier. Depending on their properties (attributes), objects can simultaneously belong to several *classes*, inheriting properties from all of them. Among these classes, one is characterized as the *primary* class of the object. The notion of a primary class helps achieving a "conceptual" as well as a physical clustering among similar objects. Collections of objects are supported as regular objects in **MOOSE** in the form of sets, multisets (bags), and arrays. The *extent* of a class, i.e., the objects that are known members of the class, is explicitly stored in the database. Every **MOOSE** database schema has a straightforward directed graph representation. Each node represents a class of objects and is labeled by the class name. Relationships between the classes in the schema are captured by the arcs of the graph. Similarly to most object-oriented data models, **MOOSE** has two major types of arcs: *component arcs* and *inheritance arcs*. The former capture structural relationships (*part-of*), whereas the latter capture semantic relationships (*is-a*) between classes of objects.

Although a component arc ($\mathbf{A} \rightarrow \mathbf{B}$) has a direction from the parent to the child class, it relates the two classes in both directions. An object in **A** has a part that is in **B**; an object in **B** is part of an object that is in **A**. The values of the attributes of an object may be explicitly assigned by the user or derived by the system through rules associated with the attribute. Although not restricted to those, aggregates of collections are the dominant kind of such *derived attributes*. The values assigned to derived attributes may be explicitly stored in the database (forward application of rules), or may be inferred on demand (backward application of rules). *Null* values are supported in **MOOSE** and are interpreted as "no related object". Thus, *null* is distinct from any other value. Also, *default values* are supported in their full generality. **MOOSE** supports four types of user-defined structural constraints that can be used to control sharing or existence dependence between objects. These are (a) sharing of objects among collections, (b) sharing of objects among their parents, (c) sharing of objects along different paths in the component graph, and (d) existence dependence of a child to its parents.

An inheritance arc ($\mathbf{A} \rightarrow \mathbf{B}$) implies that every object in **B** is in **A** also: $\mathbf{B}(X) \rightarrow \mathbf{A}(X)$. This rule is called a *generalization rule* and is implicitly assumed for every inheritance arc. In addition to an implicit generalization rule, an inheritance arc may be explicitly associated with a *specialization rule*, which specifies which elements of the parent class belong to the child class. A specialized class can be instantiated at all times, or its objects can be retrieved on demand, by applying the corresponding rule(s) on an as-needed basis. Although the semantics of simple specialization rules could be captured by typing information and inheritance, specialization rules can be arbitrarily complex, thus allowing the necessary generality to capture the level of abstraction needed for experiment management.

We are currently designing a query language for **MOOSE** that will take advantage of its features and will give the system analyst the capability to express complex queries easily. Multiple inheritance will be a significant feature of the language, and one of the key sources of its power and ease of use. Work planned for the future includes building a graphics user-interface to the DBMS that will take advantage of the graphic representation of a **MOOSE** schema and will allow the user to express queries as paths on the graph. The ability to express all types of constraints in a graphic form will then become very important. Some preliminary work on the subject has shown that focusing on different parts of the database establishes different contexts, within which the same constraint can be expressed in different ways. We are currently studying this relationship between constraints and contexts and plan to incorporate it in the query language as well as in the graphics interface.