DATA MODELING IN DE*LAB*

by

Yannis E. Ioannidis
and
Miron Livny

Computer Sciences Technical Report #761

March 1988

DATA MODELING IN DE*LAB*

Yannis E. Ioannidis [1]
Miron Livny [2]
Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract

DE*LAB* is a simulation laboratory currently under construction, that aims to provide programmers and system analysts with support for the construction of complex simulators and the management of long term simulation studies. The size of the data generated by such studies makes a DBMS an important module of the laboratory. Several unique data modeling issues, which can only be partially addressed by current models, are raised by the special nature of simulation studies. In this paper, we describe the salient features of MOOSE [3], which is the model we have developed to capture the semantics of such databases. MOOSE supports the notion of object in the strong sense, it supports collections of objects in various flavors (sets, multisets, and arrays), and it supports sharing among objects. Objects belong to various classes, some of which may be defined by specialization rules. Structural constraints between classes can be specified to capture the exact semantics of the relationships between classes, especially with respect to sharing. Finally, every MOOSE schema has a straightforward graph representation, thus facilitating a graphics interface to the database.

[1] This research was partially supported by the University of Wisconsin Graduate School Research Foundation and by the National Science Foundation under Grant IRI-8703592.

[2] This research was partially supported by a grant from Microelectronics and Computer technology Corporation (MCC).

[3] MOdeling Objects in a Simulation Environment.

1. INTRODUCTION

As the size and complexity of processing and manufacturing system 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. 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 a *simulation laboratory* that aims to provide such means and tools for simulation studies. In addition to a powerful and modular simulation language, the DE*LAB* laboratory will include a data management system, an experiment manager, and a set of output analysis utilities. In this paper, we focus on the data management system of DE*LAB* and present the MOOSE data model that we have developed to serve the modeling needs of DE*LAB* applications.

Guided by a desire for a better understanding of the needs of simulation studies and the observation that none of the existing simulation languages meets our requirements, we decided to take a 'bottom up' approach to the design and implementation of DE*LAB*. One of the advantages of this approach is that it enables us to experiment with partial implementations. Prototypes of the laboratory that support only a limited set of functions can be tested in real life studies. The experience gained from the usage of lower level elements can be used to guide the design of higher level elements and to improve the implementation of the tested elements. We believe that such experience is crucial to the design of an effective simulation laboratory.

Since the lowest layer of a simulation laboratory is the simulation language, the goal of the first phase of the project was to design and implement a simulation language. That first phase ended in the summer of 1985 when the DE*NET* (Discrete Event NETwork) simulation language became operational. The language is based on the concept of Discrete Event System Specifications (DEVS) [Zeig76]. It views the simulator as a Directed Graph where a node represents a DEVS and a directed arc represents a coupling between two DEVSs. In the past two years the language has been used in a number of real life simulation studies. It was used to simulate distributed processing environments [Chan86], communication protocols

[Qu85], 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. A language for distributed workload specification has been implemented, and interactive debugging tools have been developed. Also, utilities for "post mortem" analysis of traces generated by the reporting facility of DE*NET* have been designed.

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

The target application domain of the effort presented in this paper, i.e., experiment and simulation databases, seems to have certain similarities with statistical and scientific databases [Shos82, Shos84, Shos85]. For example, the issues of multidimensionality, aggregation, and complex data types are common in all of these applications [Shos85]. There are several issues, however, that are unique to one of the applications, e.g., compression and sampling in statistical and scientific databases and specialization rules in experiment databases (Section 4.1). This lead us into the development of a new model that addresses the specific needs of experiment databases. We do believe, however, that some of the concepts included in our model are useful for statistical and scientific databases as well.

The paper is organized as follows. Section 2 contains a brief description of the structure of DE*LAB*, which is adopted from [Livn87], and gives the justification why a DBMS is a necessary component of such a laboratory. In Section 3, we outline the specific data modeling needs of the laboratory and reason why the relational model or the currently existing object-oriented models are not adequate. Section 4 contains a description of the MOOSE model, which is our answer to these modeling needs of DE*LAB*. In Section 5, we give an extensive example of a schema in MOOSE. Finally, Section 6 gives a summary and a brief report on our current and planned future work.

2. DATA MANAGEMENT IN DE*LAB*

2.1. The Structure of DE*LAB*

Four main components can be identified in the current design of DE*LAB*: a *programming environment*, an *experiment manager*, a *DBMS*, and an *output analysis environment*. Each of these components provides support for a different type of activity. DE*LAB* was designed to support the activities of programmers who build simulators and system analysts who use them. Programmers need support in mapping a discrete event model to a program, in debugging it, and in verifying the implementation of the model. Once the program has been debugged and verified, it is transferred to the system analyst who runs the simulator and evaluates the results. At the data gathering stage of a simulation study support is needed for selecting values for input parameters, utilizing available computing resources, and storing the data. The last, and in most cases the most exciting, stage of a simulation study is the results analysis stage. At this stage the system analyst needs tools for data retrieval, statistical analysis, and graphical display.

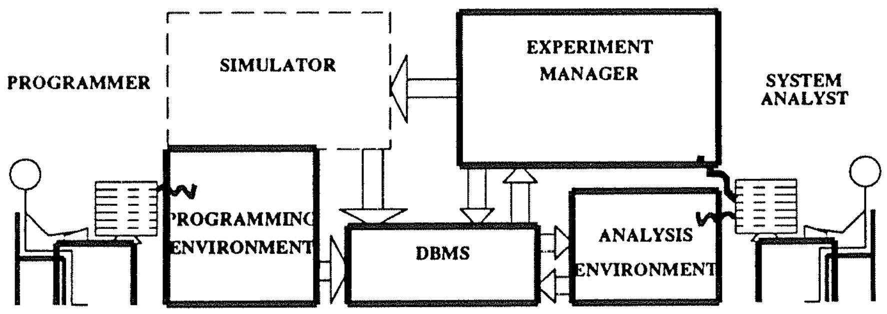

Figure 2.1: Structure of DE*LAB*.

A block diagram of DE*LAB* is presented in Figure 2.1. The programmer interacts with the simulation laboratory via a programming environment, whereas the system analyst interacts with the laboratory via an experiment manager and an output analysis environment. The programming environment, which includes a powerful simulation language, an interactive debugger, and a flexible tracing and reporting facility, supports the programmer in coding the simulator. The simulator is automatically interfaced with the database management system (DBMS) and the experiment manager. Via the experiment manager the system analyst assigns values to simulation parameters. The simulation program is then run by the experiment manager on any available resources using *remote execution* [Mutk87]. Finally, the results come back to the DBMS and are analyzed using the tools of the analysis environment, which include utilities for statistical analysis and graphical display of data.

2.2. The Need for a DBMS

One of the significant lessons we have learned from the simulation studies we have conducted is the vital importance of a DBMS. The amount of data generated throughout the lifetime of a simulation study is immense. Hundreds of attributes are required to define an experiment and hundreds of values are generated by a single run. Keeping the results in a file system is a nightmare, since after a short period of time, there is no way to map the encrypted file names and directories to experiments. At that point, the study becomes highly vulnerable to errors in relating results to experiments and a DBMS becomes essential.

Two users who share the same database do not necessarily share the same view of the data stored in the database. In the course of the different simulation studies we realized that we use two major distinct views to describe an experiment: the *simulation view* and the *system view*. The former is used to define the experiment in terms of the primitives provided by the simulation language (the programmer's view), whereas the latter is used to argue about the different runs (the system analyst's view). Since the system analyst accesses the data via the experiment manager and the analysis environment, the interfaces between these modules and the DBMS are based on the system view. For similar reasons, data generated by the simulator are described in terms of the simulation view. The DBMS of **DE***LAB* is designed to support both views. In this paper, however, we discuss the system view alone.

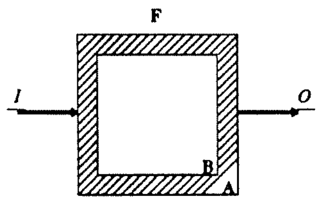

Figure 2.2: Functional view of a simulated system.

The distinction between the simulation view and the system view can be clarified by Figure 2.2. The simulated system implements a function F. The function accepts an input vector $\underline{I}$ and produces an output vector $\underline{O}$. In the system view, only $\underline{I}$, $\underline{O}$, and the interface part of F, denoted as A in Figure 2.2, are described. The system analyst is not at all interested in the complete implementation of the simulated system (F), but only in the parts of it that are directly related to the input and the output of the system (A). To the contrary, in the simulation view, in addition to $\underline{I}$ and $\underline{O}$, all of F needs to be described (both A and B). The programmer needs to be aware of the complete system (F) in order to simulate it.

Each run of the simulation corresponds to a unique $<\underline{I}, \underline{O}>$ pair. The vector $\underline{I}$ represents the values assigned to the simulation parameters, i.e., the input. These parameters control the properties of the simulated system and the environment in which it operates. For example, simulation parameters can be used to control the size of the system, its speed, the load imposed on the system, or the temperature in which it operates. The vector $\underline{O}$ represents a profile of the system behavior, i.e., the output. For example, a profile may consist of a histogram of response times, a report on system utilization, or a count of completed jobs. In most cases, the size of both $\underline{I}$ and $\underline{O}$ is very large. For example, if we simulate a distributed processing system with 32 sites, where each site is characterized by 5 input parameters and generates 10 result values, the input vector consists of at least 160 simulation parameters, whereas the output vector consists of at least 320 results. Due to the size of the input and output vectors and the number of $<\underline{I}, \underline{O}>$ pairs generated in the course of a typical simulation study, the data management in such an environment is a time consuming and error prone process. A need thus arises for a DBMS that will undertake this data management and remove the burden from the user. A DBMS can provide means for storing the data in a database and utilities for efficiently retrieving them.

3. DATA MODELING FOR DE*LAB*

3.1. Special Data Modeling Needs

We now turn our attention to the system view of the data and discuss the requirements it imposes on the modeling capabilities of the DBMS. Several of the data modeling needs of the DE*LAB* environment are encountered in other applications also. There are, however, some specific characteristics of the DE*LAB* environment that create some unique needs not found elsewhere. We will briefly discuss these needs, making the discussion slightly more detailed on the issues that are specific to a simulation laboratory.

A major problem that arises in simulation or experiment databases is the relationship between A, $\underline{I}$, and $\underline{O}$ (see Figure 2.2), i.e., the interface part of the system in test, the input vector, and the output vector. Several runs of an experiment may be performed with the same A but with different $\underline{I}$, and therefore generate different $\underline{O}$. Moreover, since most experiments are subject to statistical peculiarities and/or measurement errors, one may observe changes in the output even for the same system and the same input. Another source of such behavior is the very nature of experiments, real or simulated. Since the system under study is not well understood (after all, our incomplete knowledge of the system imposes the need for the experiment), important input or system parameters affecting the behavior of the system may be initially omited from the model. For example, in a real experiment, the output may be influenced by the temperature. Until the experimenter realizes that and includes temperature in the input vector, he or she may experience enough puzzlement and frustration in observing different output for the same values of the (incomplete) input vector and the system parameters.

The DBMS should be able to accommodate the above situation. In relational terms, $\underline{O}$ cannot be functionally determined by $\underline{I}$ and A, although this *seems* correct. The DBMS should allow sharing, i.e., experiment runs should be explicitly represented and system structure and parameters as well as input should be sharable by runs. Moreover, it should be possible to associate a given system with a given input to different output for different runs. This need of experiment databases advocates the notion of an object. In the value-oriented world of relational databases, "sharing" is expressed by equal values. In experiment databases this will create a large amount of data that are, in some sense, replicated, since they represent the same input or system. The notion of objects together with sharing at the object level overcomes the space problem, achieves a certain degree of clustering, and allows disassociation of the input, output, and the system structure.

The ability to represent and manipulate collections of objects is another important requirement from experiment databases. This is also common for the system analyst to abstract and focus on properties of a collection of objects and not of the objects themselves. For example, one may be interested in knowing that there are 6 CPUs in a system, having a total speed of 15 MIPS, and being organized as a binary tree, without any information about the speed or other characteristics of each individual CPU. Hence, collections of objects should have their own identity in the DBMS and treated independently of their members.

Related to the need for object collections is the need for aggregate computation. Aggregate attributes derived via rules from other attributes, or derived attributes in general, are also needed in a DE*LAB* database. The task of the system analyst is the statistical study of the behavior of the system. The implicit computation of aggregates is an important tool for that. As we mentioned above, one should also have the ability to abstract and deal with aggregates directly, independent of other attributes. Clearly, in that case, the aggregate attributes will not be derived.

Finally, there are cases in experiment databases where certain system parameters are meaningful only if other parameters take specific values. For example, assume that a computer system is modeled, and part of the system is a network. If it is a ring network, i.e., if the network parameter *type* has some appropriate value, then the network parameter *jump*, which is the distance of nodes in the ring that are directly connected to each other (in addition to the connections of the ring), becomes important. If the network is not a ring, then *jump* is meaningless. The ability to classify objects according to special properties they may have, and from that represent special characteristics of them, is very important, since it directly influences the ability to represent the system parameters that affect its behavior.

In the next subsections we will describe the experience we had with using the relational model for DE*NET* and also argue why existing object-oriented data models do not directly satisfy the above mentioned needs.

3.2. The DE*NET* DBMS

We have implemented a relational DBMS that meets the special needs of DE*NET*. The compiler and runtime environment of the language have been instrumented to automatically generate a description of the run. DE*NET* provides the means by which a simulation program can declare the structure of the simulated system and its input and output vectors. The runtime environment of DE*NET* reads the system description and the input values from a file. When the program terminates, the values of its output parameters are put back into the database.

All the information required to plan the relational schema of a simulation is available to the DE*NET* compiler. We have instrumented the compiler to automatically generate such a schema. The schema is passed to the DBMS that creates the relational database. The structure of the input and output vectors, however, leads to a schema that is not in first normal form. Based on a schema generated by the compiler, a separate relational database is established for every simulation program. Each time a module of the program is recompiled, the DBMS checks whether the input/output structure was modified. When a modification is encountered, the structure of the appropriate relations are updated. Depending on whether an attribute was added or removed from a relation, actions to guarantee the integrity of the stored data are performed.

Based on our experience with the above relational DBMS, we have concluded that the relational model is inappropriate for our application. We faced many problems in mapping DE*N*ET programs to relational schemas. A simulation study leads to complex data structures that change from one run to the other. Thus, our DBMS has to deal with non-atomic attributes and variable length tuples. Although the relational model can be used to support the simulation view it is not the optimal one. Moreover, it is clearly inadequate to support the system view of the data. It lacks most of the features mentioned in Section 3.1. This made the incorporation of a different model necessary.

3.3. Why not Existing Object Oriented Models

In recent years, several data models have been developed that are explicitly or implicitly based on the object-oriented paradigm [Ship81, Ship81, Zani83, Cope84, Dada86, Ditt86, Bane87]. The introduction of all these models was motivated by the need to provide database support for non-business applications and the inability of the relational model to capture the semantics of such databases. Based on our own list of desiderata of Section 3.1, one sees that object-oriented models provide functionality that meets (at least partially) the needs of our application. They support the notion of object, aggregate and class hierarchies with inheritance, etc.

We have chosen not to use an existing object-oriented model for the following reasons. First, there is no consensus yet on what an object-oriented model is. There are several approaches taken, none of which seems to be dominating at the moment. Second, object-oriented models usually employ *encapsulation* in their data access facility. This hinders the development and optimization of a query language, which we believe, is a necessary tool for the system analyst. Third, current object-oriented models cannot meet all the requirements of an experiment database, or they can do it in an unnatural way. For example, classifying objects according to special properties they may have can be implemented in object-oriented models using the notion of *subtype*. In our application, however, it is much more natural to represent such a situation using rules than using subtypes (see Section 4.4, Figure 4.9).

For all the above reasons, we have chosen not to use or extend an existing object-oriented model, but to develop our own. Using an existing model would impose some restrictions on us; our thinking of the application would have to be twisted to a certain extent in order to map it into the concepts of the existing model. Our strong desire to avoid that and our expertise on the specific application domain of simulation lead us into the development of MOOSE.

Certain characteristics of MOOSE are shared by many object-oriented models, e.g., the notion of an object, aggregate and class hierarchies, etc. Others, we believe, are unique to MOOSE, e.g., specialization rules (Section 4.4), primary class (Section 4.5), etc. Given these differences, and according to our current understanding of what object-oriented models are, we do *not* call our model object-oriented.

4. THE MOOSE DATA MODEL

In this section we will describe the salient features of the data model we have developed to address the needs of DE*LAB* applications, as were listed in Section 3.1. Although the conception of the model was motivated by the specific needs of a simulation laboratory, we believe that it addresses the representational needs of many other environments. To emphasize the generality of the model, the examples discussed below are taken from a variety of applications. The great majority of the examples, however, is indeed taken from experiment databases.

Unlike most other applications, the notion of an object is relatively vague in the world of DE*LAB*. For example, assume that a CPU is simulated and its performance under various workloads is tested. It is unclear whether specifying the same values for the parameters that characterize the CPU for several runs of the experiment makes it the same CPU object or not, i.e., whether a simulated object has an identity or not, whether the life-span of a simulated object is a simulation run or whether it goes beyond that. These issues imply that, in a simulation environment, the object-oriented and value-oriented approaches to data modeling are much closer to each other than in other environments.

In MOOSE, we have decided to support the notion of an object in the strong sense. Objects may live arbitrarily long, unrestricted by any boundaries imposed by the experiment runs. There are several advantages to this approach. First, the model is applicable in domains other than simulation. Second, experiments are sometimes conducted in a hybrid environment, partly simulated and partly real. Supporting the strong notion of an object allows MOOSE to represent such environments in a uniform way. Third, a proliferation of objects is avoided, thus saving space and allowing stronger physical clustering of similar experiment runs.

Every individual object or object collection is assigned an identifier that is unique for the entire database. In the forthcoming examples, assuming for simplicity that all collections are homogeneous (i.e., all their elements belong to the same class), identifiers for object collections are denoted by a class name in upper case letters followed by a number, e.g., DISK4, and identifiers for individual objects are denoted by a class name in lower case letters followed by a number, e.g., disk13. Also, variables ranging over identifiers of collection or object classes will be denoted by the class name (without a number) in upper or lower case letters respectively.

In the sequel, we will use the terms *class* and *type* interchangeably. A class represents a set of objects having the same structure and the same properties. [2] In a DE*LAB* environment, there is a need to represent not only individual objects, but collections of objects as well. Depending on the specific semantics of the application, collection of objects may appear in different flavors. In addition to supporting the notion of a set in the traditional mathematical sense, there is a need to support duplication and ordering at the set level. Duplication allows us to express implicit sharing of objects by other objects. Ordering allows us to express functional dependencies between members of sets. This plays an important role in disassociating the description of a simulated system and its input and output. Motivated by these needs, we have included four types of classes in MOOSE.

- Classes of individual objects, e.g., **disk**, the class of disks, **emp**, the class of employees, **int**, the class of integers. They are called *object classes*.

[2] Throughout the text, class names appear in bold.

- Classes of object sets, e.g., {**disk**} the class of disk sets. They are called *set classes.*
- Classes of object multisets or bags (sets where an element can appear multiple times), e.g., {{**disk**}}, the class of disk multisets. They are called *multiset classes.* A multiset can be seen as a special case of a set, where the elements of the set are object-count pairs, the count representing the number of times the object appears in the multiset.
- Classes of object arrays, e.g., **disk[{cpu}]**, the class of one dimensional disk arrays indexed by a set of CPUs. They are called *array classes.* An array can also be seen as a special case of a set, where the elements are assumed ordered. Dimensions of arrays can be indexed by any other sets of objects. The index set of an array dimension is restricted to having the same cardinality as the cardinality of the dimension.

Because of their structural similarity, sets, multisets, and arrays share several common characteristics. For ease of reference, we call these three types of classes *collection classes.* The system analyst can define an arbitrary number of any of the above types of classes. There are, however, four system supported object classes, called *base classes*: integers, floats, character strings, and booleans. Classes that are subsets or supersets of the base classes are called *simple classes.*

Except for simple classes, the known members of a class are explicitly stored in the database. This is called the *extent* of the class. For all those classes, objects can be explicitly inserted into and deleted from the extent of the class. Insertions and deletions are governed by various system or user defined constraints (e.g., deletion of an object may imply deletion of other objects referencing it), but they are not discussed in this paper.

Keeping the extent of class is very important for a database in the DE*LAB* environment. For example, one may want to introduce to the system a new CPU. Although, there are no experiments that have been run with this CPU - so, structurally this CPU does not appear in the database - nevertheless it is important that it is stored in the system, so that it is available later when the system analyst decides to run experiments with it. For the same reasons, class extent queries are allowed in the system: "What types of CPUs do I have at my disposal?".

4.1. Graph Representation of a schema

Every database schema developed in MOOSE has a straightforward directed graph representation. Every node in the graph represents a class of objects and is labeled by the class name. According to the kind of the corresponding class, one can define *base, simple, object, set, multiset, array,* and *collection* nodes. Base nodes are represented as ellipses, to be easily distinguishable from the rest. All other object nodes are represented as rectangles. Collection nodes are also represented as rectangles, having the corresponding object node inside them and identification whether they are set, multiset, or array nodes. Array nodes have appropriate arcs pointing to the set node(s) indexing the array. Figure 4.1 gives the specific graphic representations for an array of disks (indexed by a set of cpus), a set of disks, and a multiset of disks.

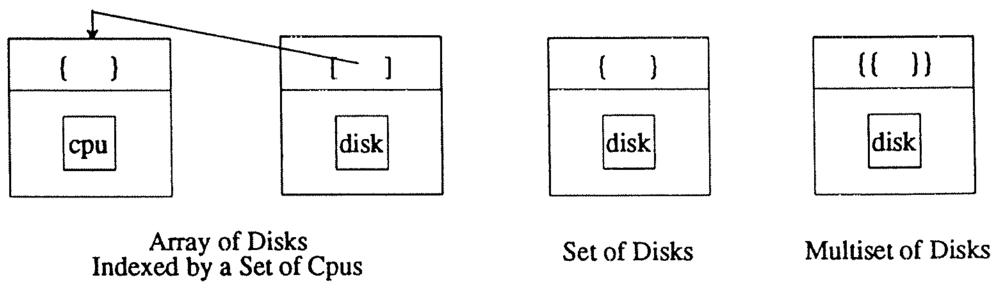

Figure 4.1: Graphic representation of collection nodes.

Relationships between the various classes in the schema are captured by the arcs (directed edges) of the graph connecting the appropriate nodes. Similarly to most object-oriented data models, MOOSE has two major types of arcs: *component arcs* and *inheritance arcs*. Component arcs are needed to represent structural relationships between (classes of) objects. Inheritance arcs are needed to represent semantic relationships between (classes of) objects. Component arcs are shown as solid lines. Inheritance arcs are shown as broken lines. The spanning subgraph of the schema graph (i.e., the complete set of nodes) with the component arcs as its arc set is called the *component subgraph*. Likewise, the spanning subgraph of the schema graph with the inheritance arcs as its arc set is called the *inheritance subgraph*. We will examine the two subgraphs separately.

4.2. The Component (Part-of) Subgraph

Component arcs relate objects of a class to their parts and vice versa. Every object or collection node may have an arbitrary number of children via component arcs. These children are called the *components* of the parent node and can be other object or collection nodes. The direction of a component arc is from a class of objects to the class of their parts. Since a base class cannot have components, we omit the arrow of arcs leading into base nodes.

Although a component arc (A $\rightarrow$ B} has a direction from the parent class to the child class, it nevertheless 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 (see Figure 4.2). For example, as much as one wants to find the disks located in a site of a distributed system, one also wants to find the site where a disk is located. This reflects the common situation in which a system may need to be analyzed in both a top-down and a bottom-up fashion. Clearly, this naturally arises in other applications as well, e.g., in a traditional environment on may want to refer to the employees of a department and the department of an employee.

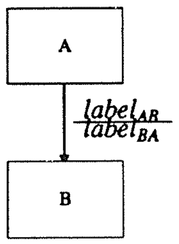

Figure 4.2: Labeled component arc.

Normally, all component arcs are associated with two labels, i.e., two names, one for each direction. For example, the *employees* of a department, the *department* of an employee, the *salary* of an employee. Graphicly, they are shown as in Figure 4.2, the one for the direction of the component edge above the one for the opposite direction. More often than not, however, one (or both) of these labels is equal to the class of the head of the arc traversed in the direction corresponding to the label. For example, in Figure 4.3, the employees of a department is a set of employees (**department** $\rightarrow$ {**employee**}), the department of an employee is a department (**employee** $\rightarrow$ **department**).

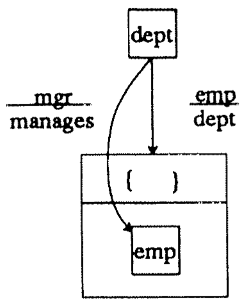

Figure 4.3: Departments and employees.

Clearly, this is not always true; e.g., the manager of a department is an employee not a manager (**department → employee**). In fact, the arc label cannot be equal to the name of the head class when multiple parts of an object belong to the same class. Then, component arcs have to be given specific names. For example, both the father and the mother of a family belong to the class person (**family → person**), so the corresponding arcs have to be given appropriate labels (see Figure 4.4).

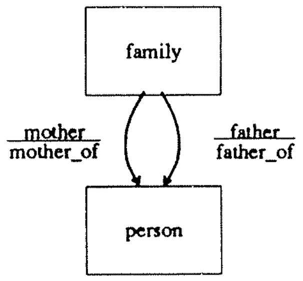

Figure 4.4: Example of labels that cannot be replaced by class names.

In the sequel, unless it is necessary, class names are used as arc labels, which are thus omited from the graphs.

There is no clear need for having heterogeneous collections of objects in MOOSE. Besides, this construct can be represented by the ones that are already available, if the user is willing to name a few more attributes. This is shown in Figure 4.5.

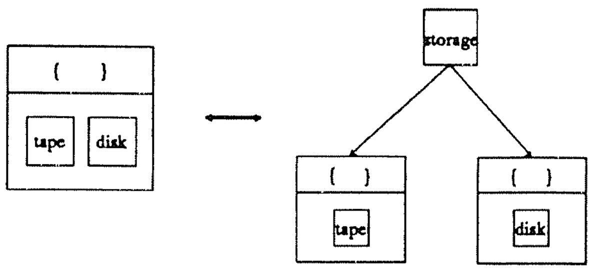

Figure 4.5: Representing heterogeneity.

Nevertheless, for orthogonality's sake, we allow heterogeneous collections of individual objects and object collections.

As we mentioned above, component arcs represent a bidirectional relationship. Following the relational terminology, component arcs represent *attributes* of the participating nodes. The tradition of the relational model is also followed in that the value of a component with label *comp* of an object with identifier *obj* is denoted by *obj.comp*. For example, site2.cpus=CPU5, cpu3.speed=2, dept14.mgr=emp2. Notice that collection nodes may have attributes, independent of the attributes of the objects in the collection. For example, CPU5.number_of_cpus=10, CPU5.fastest_cpu=cpu3, CPU5.name="The Famous CPU Set".

The values of the attributes of an object may be explicitly assigned by the user, but they may also be derived by the system through rules associated with the corresponding component arc. Such *derived attributes* are very common in experiment databases. Although not restricted to those, aggregates of collections are the dominant kind of derived attributes. For example, the following rule determines the number of CPUs in a specific cpu set:

$$CPU.number\_of\_cpus := count(CPU)$$

The values assigned to such *derived attributes* may be explicitly stored in the database (forward application of the rule), or may be inferred on demand (backward application of the rule). Although rarely expected in the DE*LAB* environment, multiple interfering rules may be associated with a component arc. Resolving the conflict is beyond the scope of the paper and is not discussed any further.

Given a component arc, there may be cases where a specific object in the parent class does not have any objects of the child class as parts, or that an object of the child class is not a part of any object of the parent class. For example, for the component arc (**computing_system** $\rightarrow$ {**disk**}), a specific computing system may be diskless, or a specific disk may not have been used in any computing_system yet. In that case, the value of the corresponding attribute will be *null.* This applies to both cases where the child node is an object node or a collection node. *Nulls* are interpreted as "no related object". So, *null* is distinct from any other value. Also, *null* is only a special case of a *default value.* Our model supports defaults in their full generality.

4.3. User-Specified Constraints on the Component Subgraph

Given the constructs mentioned above, the user has the power to represent general and complex component relationships between classes of objects. There are several cases, however, where the user wants to impose structural constraints on the relationship between components, in order to capture the semantics of the data. In particular, object sharing and existence dependence are important features of MOOSE that need to be controlled by the user. Depending on the situation modeled, sharing may be forced, forbidden, or left optional. Also, existence dependence may be forced or left optional. MOOSE supports four types of constraints of this form. Three of them can be used to control sharing and one can be used to control existence dependence.

(a) Sharing of objects among collections. It can either be optional or forbidden; being forced is meaningless. Forbidding sharing is equivalent to saying that the various collections of objects in a class form a partition of the class. E.g., the platters of a disk pack can only participate in a single collection of platters (since they can be parts of only one disk-pack). On the other hand, a disk can be part of multiple collections of disks. In the graphic representation of a MOOSE schema, a double box in the object node within the collection node means that an object can participate in multiple collections; a single box means it cannot (see Figure 4.6).

(b) Sharing of objects among their parents. Again, it can either be optional or forbidden. For example, if the collections of disks assigned to two distinct sites can be the same, then the same collection is shared by both sites. If not, then every collection of disks belongs to a single site. In the graphic representation of a MOOSE schema, a double arrow in the component arc means that two parent objects can share the same child object; a single arrow means they cannot (see Figure 4.6).

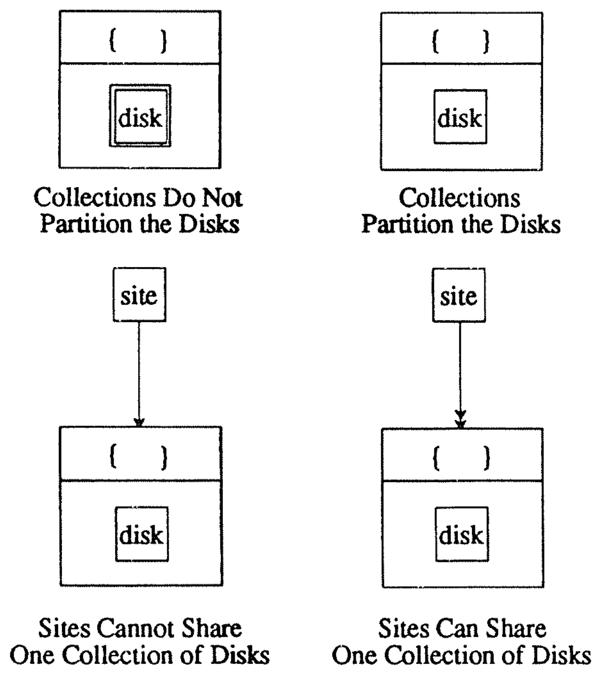
{ }
disk
Collections Do Not Partition the Disks
{ }
disk
Collections Partition the Disks
site
{ }
disk
Sites Cannot Share One Collection of Disks
site
{ }
disk
Sites Can Share One Collection of Disks

Figure 4.6: Graphic representation of structural component constraints.

(c) Sharing of objects along different paths in the component graph. This can be optional, forbidden, or forced. For example, in Figure 4.7, the objects in **D** related to an object in **A** via objects in **B** may be unrelated to (optional), or they may have to be different than (forbidden), or they may have to be the same as (forced) the objects in **D** related to the same object in **A** via objects in **C**.

Since the paths concerned may be arbitrarily long, we do not have a good way to capture such constraints in the graphic representation of a MOOSE schema. They will have to be stated declaratively.

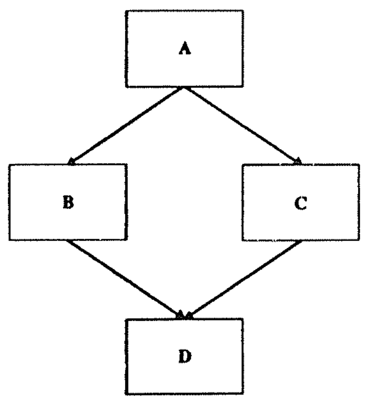

Figure 4.7: Component paths with common ends.

(d) Existence dependence of a child to its parents. E.g., if someone can be employeed without being assigned to a department, then employees exist independent of their parents (the departments). If not, then a department must exist before an employee can be defined. This can be used to express the so-called *referential integrity*.

Notice that constraints (a) and (b) are similar but not identical. Constraint (a) discusses a property of a class, whereas constraint (b) discusses a property of a component arc.

4.4. The Inheritance Subgraph

Inheritance arcs relate classes with each other with respect to containment. Assume that we have the inheritance arc of Figure 4.8 between classes A and B.

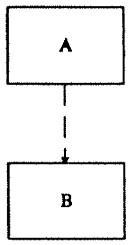

Figure 4.8: Inheritance arc - A is the superclass of B.

The arc (A $\rightarrow$ B) implies that every object in B is in A also:

$$\mathrm{B}(X) \rightarrow \mathrm{A}(X).$$

The above rule may be 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* as well, which specifies which elements of the parent class belong to the child class also. In such a case, the inheritance arc (A $\rightarrow$ B) of Figure 4.8 has a set of two rules associated with it:

$$\begin{gathered} \mathrm{B}(X) \rightarrow \mathrm{A}(X), \\ \mathrm{A}(X) \wedge \mathit{property}(X) \rightarrow \mathrm{B}(X). \end{gathered} \tag{1}$$

The first rule specifies which members of B are members of A also (namely all). The second rule specifies which members of A are members of B also, i.e., the opposite. (Although we permit multiple specialization rules to be associated with the same inheritance arc, the following discussion assumes only one such rule per arc.)

The antecedent of a specialization rule uses the various attributes that may be associated to objects of the parent class of the arc to specify the conditions of the specialization. Consider an inheritance arc from the class **cpu** to the class **fast_cpu**. The following is an example of a specialized rule for this arc:

$$\mathbf{cpu}(cpu) \wedge cpu.speed > 5 \rightarrow \mathbf{fast\_cpu}(cpu). \qquad (2)$$

The rule should be interpreted as follows: "If the speed of a cpu is higher than 5 (MIPS), then the cpu is considered fast". A specialized class can be instantiated at all times, or its objects or object collections can be retrieved on demand, by applying the corresponding rule(s) on an as-needed basis. Direct object classification, i.e., not based on attribute values, is also possible, since it represents a special case of a rule without an antecedent. For example, **optical(disk13)**.

Specialization rules are used in different ways, depending on the way objects are created by the user. Assume we have the inheritance arc (**A** $\rightarrow$ **B**) of Figure 4.8 and a specialization rule r associated with it. If the user inserts an object into the extent of **A**, r is used as a derivation rule. I.e., the object is examined, and if it satisfies the rule's qualification, it is inserted into the extent of **B** as well. If the user assigns an object to class B, r is used as a constraint rule. I.e., the object is examined, and if it satisfies the rule's qualification the assignment is accepted; otherwise it is rejected. Regardless of how the rule is interpreted and processed, if the object belongs to B, the specialized attributes associated with B (if any) are assigned values that are either given by the user or derived from rules associated with the attribute (see Section 4.1). If neither of these applies, a null value is assigned with the interpretation mentioned above, i.e., that no object is associated with the attribute.

Notice that specialization rules are neither integrity constraints nor relational view definitions. Four differences can be identified.

i. Specialization rules can be used both as integrity constraints and as inference rules, depending on whether an object is inserted into the extent of the parent or the child class of the corresponding inheritance arc.

ii. Specialization rules have a very specific form (see (1) and (2)), especially regarding their consequent, which simply declares that an object is a member of a class.

iii. Unlike view definitions, specialization rules do not assign values to the attributes of the child class. The attribute values are either derived by separate rules or supplied by the user.

iv. Specialization rules are not used backwards to translate updates on the attributes of the child class into updates on attributes of the parent or other related classes. Updates on attributes of the child class are either rejected (if the previous attribute value was derived by a rule), or updated in place (if the previous attribute value was manually assigned by the user).

As in object oriented systems, an inheritance arc (**A** $\rightarrow$ **B**) implies that the types of some attributes of **B** are subtypes of the types of the same attributes of **A**, and/or that **B** has more attributes than **A**. Specialization rules serve the same purpose. For example, in characterizing fast CPUs, rule (2) could be avoided by defining **int_gr_5** as a subclass of **int** and declare the attribute *speed* of **fast_cpu** being of type **int_gr_5** (see Figure 4.9).

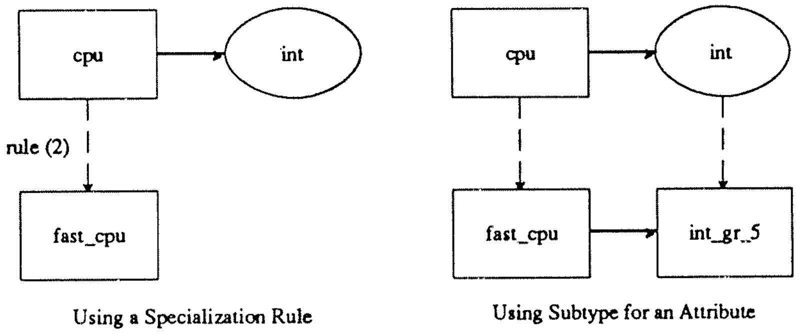

Figure 4.9: Expressing a specialization rule with subtypes of attributes.

As another example, which cannot be captured in a straightforward way by typing information, rule (3) defines a set of cpus as being a **homogeneous_set_of_cpus** if all the members of the set have equal values in the attribute *speed.*

$$\mathbf{cpu}(CPU) \wedge \forall (cpu \in CPU) \forall (cpu' \in CPU)(cpu.speed = cpu'.speed) \qquad (3)$$
$$\rightarrow \mathbf{homogeneous\_set\_of\_cpus}(CPU)$$

We believe that, in the DE*LAB* environment, there are situations where the specialization rule is a more natural representation of the relationship between a class and its subclass. Such situations include cases where there is a well-defined class of objects, objects of that class exist in the database, but there is no attribute of any other class that belongs to that class. The class **fast_cpu** is such an example. At data gathering time, the system analyst may be simply inserting CPUs in the database without worrying which

CPU is fast and which is not. At result analysis time, however, the system analyst can ask questions of the form "retrieve the experiment results for fast cpus", which are now well defined and easy to express. Needless to say that such queries are expected to be very common in the DE*LAB* environment. Every query given to the system can be thought of as a rule that specifies which subset of the database to be given back as an answer to the user. If this rule is difficult and time-consuming to express and defines a semanticly meaningful class of objects, the user may specify the rule to the system once, give a name to the resulting class, and simply use the class name after that.

4.5. Primary Class of an Object

Every object in the database can simultaneously belong to several classes, if these classes have a common ancestor in the inheritance subgraph. Since no global superclass is assumed in the schema, the above condition cannot be satisfied vacuously. One can immediately infer that every object belongs to a sequence of classes, each one of which is a subclass of the previous one. Among all of these classes, one of them is characterized as the *primary* class of the object. The object identifier is a combination of the class identifier of its primary class and an identifier unique within that class. Hence, the resulting identifier is globally unique.

There are two questions to be answered with respect to the primary class:

(a) Why do we need the concept of the primary class?

(b) For each object, how does one choose its primary class?

The need for the primary class arises for physical and "conceptual" clustering of objects. An object plays different roles in different contexts. Usually, in every context one of these roles dominates. In a university context, John Smith is primarily a professor, on top of being the chairman of the curriculum committee of his department and a member of the bridge club of the university. A database can be thought of as capturing our knowledge about a specific environment, a specific context. The dominant role played by an object in that context determines its primary class. This gives us the "conceptual" clustering. The implementation can take advantage of the conceptual clustering to achieve physical clustering also.

Regarding the choice for a primary class, we believe that the straightforward solution is the conceptually best one also. The primary class of an object is the one specified when the object is first inserted into the database.

4.6. Objects in Classes

This section could have been titled "inheritance". We avoid the term, however, because of the presense of specialization rules. Semanticly, we envision that an object is stored in the extent of every class to which it belongs. This, clearly, includes classes connected with unidirectional paths in the inheritance subgraph, i.e., sequences of classes, each one of which is a superclass of the next one. There is no doubt that the implementation of MOOSE will be different. Conceptually, however, it is appealing to think as the object residing in all of the class extents to which it belongs.

Suppose that an object *obj* is inserted into the extent of a class. This may be its first appearance in the database, in which case this will be its primary class, or not. The insertion algorithm is governed by the following rules:

(a) For every class **B** into whose extent *obj* is inserted, it is also inserted in every superclass **A** of **B** if it satisfies any specialization rule(s) associated with (**A** $\rightarrow$ **B**).

(b) For every class **B** into whose extent *obj* is inserted, it is also inserted in every subclass **C** of **B** if there is a specialization rule associated with (**B** $\rightarrow$ **C**), and *obj* satisfies it.

(a) shows how insertion propagates up the inheritance subgraph, whereas (b) shows how insertion propagates down the inheritance subgraph. Specialization rules are used as integrity constraints in (a) and as derivation rules in (b). If any rule encountered in (a), i.e., on the way up, is not satisfied by *obj*, the object is not inserted in the database and the user is notified.

Objects of a class inherit the attributes of all of the class's ancestors in the inheritance subgraph, as well as the attributes of all of their ancestors in the component subgraph. Conflict between attribute names of classes in the same root-to-leaf path are resolved by proximity, i.e., the attribute of the closest object on the way up is inherited. Since the graph is not a strict hierarchy, however, multiple parents, and therefore multiple inheritance, are allowed. We are currently looking into several techniques for resolving conflicts between attribute names due to multiple inheritance in order to identify the appropriate one for our application.

4.7. User-Specified Constraints on the Inheritance Subgraph

We have mentioned above that an object can belong to two classes, none of which is a subclass of the other, if the two classes have a common ancestor. This represents a form of sharing that is different from sharing at the component level. The issue is whether classes can share objects, i.e., whether their extents can intersect. The general ability for classes to share objects is important, but the semantics of specific classes may inhibit that. For example, an object cannot belong to both the **cpu** and the **disk** class simultaneously. Hence, the user should have the ability to control sharing at the class level. There are two options: classes may intersect or may be mutually exclusive. We have chosen sharing to be the default in MOOSE. If two classes with a common ancestor cannot share objects, the user may specify that they are mutually exclusive.

5. AN EXAMPLE

As an example of how our model will be used, we outline the MOOSE schema for a simulation study in which the performance of different Distributed Concurrency Control (DCC) algorithms is evaluated [Care87]. The objective of the study is to evaluate the response time of different DCC algorithms under a variety of workloads and system structures. In our presentation here several details have been excluded for the sake of clarity. For the same reason, we have focused on the component subgraph of the schema.

Each simulation run simulates the behavior of a Distributed Database Management System (DDBMS) with a given DCC algorithm under a given workload. The DDBMS consists of a set of sites and a communication network. A site has a set of CPUs and a set of disks on which a set of partitions are stored. Partitions may be replicated on a number of sites. Transactions are submitted via terminals that are attached to the different sites. A set of transaction types is associated with each site. Each class has an access probability for each partition in the system.

The output of a simulation run consists of three array-sets of real numbers that represent the response time, cpu utilization, and disk utilization of the DDBMS sites, and a single real value that represents the utilization of the network. Figure 5.1 is a graphical presentation of the schema that captures the main elements of the study.

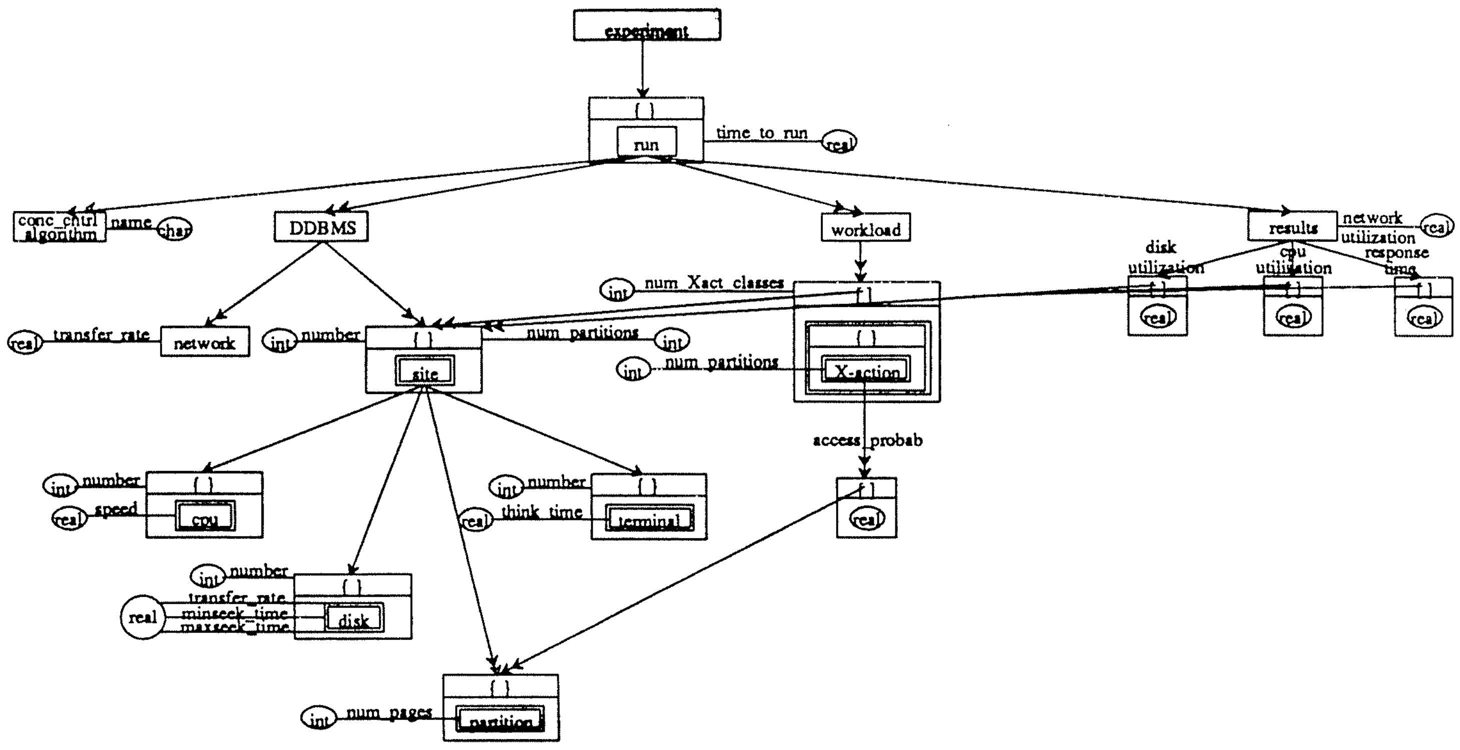

Figure 5.1: The MOOSE schema for a distributed concurrency control algorithm simulation study.

We would like to indicate how the various features of MOOSE have been used to model the semantics of this simulation study. At several points we have expressed structural constraints of type (a) and (b) (see Section 4.3) on the graph. For example, CPUs cannot be shared among sites, so there is only a single arrow in the arc (**site** → {**cpu**}). Arrays have been used at several points in the schema to associate objects of the DDBMS to input and output parameters. For example, response time is an array of reals indexed by a set of sites. We assume that the appropriate constraints of type (c) (Section 4.3) have been declaratively specified forcing the set of sites indexing the array of response times to be the same set of sites of the DDBMS for the same run, i.e., starting from the same run and traversing the two paths that lead to the node {**site**} should give the same result. Similar constraints are assumed for the other arrays of the schema also. Notice that with the use of arrays and type (c) constraints we have achieved the desired separation of the system from its input and output, which was one of the goals of the model (see Section 3.1). Finally, we would like to exemplify the orthogonality between the various types of classes in MOOSE by indicating that the workload is an array of sets of transactions, indexed by a set of sites.

6. SUMMARY AND FUTURE WORK

DE*LAB* is a simulation laboratory currently under construction, that aims to provide programmers and system analysts with support for the construction of complex simulators and the management of long term simulation studies. The size of the data generated by such studies makes a DBMS an important module of the laboratory. Several unique data modeling issues, which can only be partially addressed by current models, are raised by the special nature of simulation studies. In this paper, we have described the salient features of MOOSE, which is the model we have developed to capture the semantics of such databases. MOOSE supports the notion of object in the strong sense, it supports collections of objects in various flavors (sets, multisets, and arrays), and it supports sharing among objects. Objects belong to various classes, and one may define specialized classes based on specialization rules. Every MOOSE schema has a straightforward graph representation. Structural constraints between classes can be specified to capture the exact semantics of the relationships between classes, especially with respect to sharing. In addition, MOOSE supports derived attributes, default values, and null values. Finally, MOOSE have several other features not discussed in this paper, like universal and existential quantification (rule (3), Section 4.4), extensive schema changes, and variable length attributes.

The work presented in this paper is only the first step in providing DBMS functionality in DE*LAB*. We are currently working on the design of 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. We plan to implement a DBMS based on MOOSE and the query language under design on top of the EXODUS extensible DBMS [Care86]. 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 (see discussion in Section 4.3). 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.

7. REFERENCES

[Bane87]
Banerjee, J. et al., "Data Model Issues for Object Oriented Applications", *ACM Trans. on Office Information Systems* 5, 1 (January 1987).

[Care87]
Carey, M. and M. Livny, *"A performance profile of Distributed Concurrency Control"*, in preparation, December 1987.

[Care86]
Carey, M. et al., "The Architecture of the EXODUS Extensible DBMS", in *Proc. of the 1986 International Workshop on Object-Oriented Database Systems"*, edited by K. Dittrich and U. Dayal, Pacific Grove, CA, September 1986, pages 52-65.

[Chan86]
Chang, H. and M. Livny, "Distributed Scheduling Under Deadline Constraints: A Comparison of Sender-initiated and Receiver-initiated Approaches", in *Proc. of the 1986 IEEE Real-Time Systems Symposium*, 1986.

[Cope84]
Copeland, G. and D. Maier, "Making Smalltalk a Database System", in *Proc. of the 1984 ACM-SIGMOD Conference on the Management of Data*, Boston, MA, June 1984, pages 316-325.

[Dada86]
Dadam, P. et al., "A DBMS Prototype to Support Extended NF^2 Relations: An Integrated View on Flat Tables and Hierarchies", in *Proc. of the 1986 ACM-SIGMOD Conference on the Management of Data*, Washington, DC, May 1986, pages 356-367.

[Ditt86]
Dittrich, K. and U. Dayal, *Proc. of the 1986 International Workshop on Object-Oriented Database Systems"*, Pacific Grove, CA, September 1986.

[Livn87]
Livny, M., "DE*LAB* - A Simulation Laboratory", in *Proc. of the 1987 Winter Simulation Conference*, Atlanta, GA, December 1987.

[Mutk87]
Mutka, M. W. and M. Livny, "Scheduling Remote Processing Capacity in A Workstation-Processor Bank Network", in *Proc. 7th International Conference on Distributed Computing Systems*, 1987.

[Ship81]
Mylopoulos, J., P. A. Bernstein, and H. K. T. Wong, "A Language Facility for Designing Database-Intensive Applications", *ACM TODS* 5, 2 (June 1980), pages 185-207.

[Qu85]
Qu, Y., L.H. Landweber, and M. Livny, "PARING: A Token Ring Local Area Network with Concurrency", in *Proc. 10th Conference on Local Computing Networks*, 1985.

[Ship81]
Shipman, D. W., "The Functional Data Model and the Data Language DAPLEX", *ACM TODS* **6**, 1 (March 1981), pages 140-173.

[Shos82]
Shoshani, A., "Statistical Databases: Characteristics, Problems, and some Solutions", in *Proc. 8th International VLDB Conference*, Mexico City, Mexico, September 1982, pages 208-222.

[Shos84]
Shoshani, A., F. Olken, and H. K. T. Wong, "Characteristics of Scientific Databases", in *Proc. 10th International VLDB Conference*, Singapore, August 1984, pages 147-160.

[Shos85]
Shoshani, A. and H. K. T. Wong, "Statistical and Scientific Database Issues", *IEEE Transactions on Software Engineering* **11**, 10 (October 1985), pages 1040-1047.

[Zani83]
Zaniolo, C., "The Database Language GEM", in *Proc. of the 1983 ACM-SIGMOD Conference on the Management of Data*, San Jose, CA, May 1983, pages 207-218.

[Zeig76]
Zeigler, B. P., *Theory of Modelling and Simulation*, John Wiley & Sons, New York, N.Y., 1976.