ZOO: A Desktop Experiment Management Environment*

Yannis E. Ioannidis[†] **Miron Livny** **Shivani Gupta** **Nagavamsi Ponnekanti**
Department of Computer Sciences, University of Wisconsin, Madison, WI 53706
{yannis,miron,shivani,vamsi}@cs.wisc.edu

Abstract

Over the last decade, a dramatic increase has been observed in the ability of individual experimental scientists to generate and store data, which has not been matched by an equivalent development of adequate data management tools. In this paper, we present the results of our efforts to develop a *Desktop Experiment Management Environment* that many experimental scientists would like to have on their desk. The environment is called *ZOO* and is developed in collaboration with domain scientists from Soil Sciences and Biochemistry. We first describe the overall architecture of ZOO, and then focus on key features of its various components. We specifically emphasize aspects of the object-oriented database server that is at the core of the system, the experimentation manager that initiates the execution of experiments as a result of scientists' requests, and the mechanisms that the modules of the system use to communicate between them. Finally, we briefly discuss our experiences with the use of the current ZOO prototype in the context of plant-growth simulation experiments and NMR spectroscopy experiments.

1 Introduction

In the past few years, several scientific communities have initiated very ambitious and broad-ranged projects in their disciplines. The NASA Eos effort and the NIH Human Genome project are two examples of such national and international scientific endeavors. A major part of these projects is the collection of huge amounts of data (sometimes measured in petabytes) on complex phenomena. Managing this surge of scientific data poses significant challenges, many of which cannot be effectively addressed by existing database technology. This has resulted in much research activity in the area of *Scientific Database Systems* [FJP90, SOW84].

Despite the renewed interest in the area, still little attention has been devoted to the needs of small teams of scientists who perform individual experimental studies in their laboratories. In particular, a major problem that many experimental scientists are facing is that there are no adequate experiment management tools that are powerful enough to capture the complexity of the experiments and at the same time are natural and intuitive to the non-expert. A small laboratory that can easily generate and store several megabytes of data per day is still dependent on the good old paper notebook when it comes to keeping track of the data.

Over the past three years, in collaboration with several domain scientists, we have studied the needs of a wide range of experimental disciplines, developed solutions to some of the basic problems in experiment management, and have made significant progress towards implementing a simple *Desktop Experiment Management Environment (DEME)* called *Zoo*. Our work has proceeded in a tight loop between developing *generic* experiment management technology that is implemented in a *generic* tool, Zoo, and installing *customized* enhancements of the tool that constitute full systems (complete *Customized Desktop Experiment Management Systems (CDEMSs)*) in laboratories[1] of interest. New technology has been continuously transferred to these laboratories, while feedback from installed software as tested and evaluated in real-life settings has affected our research directions and decisions. Because of our emphasis on the genericity of the basic technology developed, we believe that the main research results and software tools are applicable to a wide range of experimental disciplines.

* Work supported in part by the National Science Foundation ("Scientific Databases Initiative") under Grant IRI-9224741.

† Additionally supported in part by the National Science Foundation under Grant IRI-9157368 (PYI Award) and by grants from DEC, IBM, HP, AT&T, Oracle, and Informix.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996**

[1] We use the term 'laboratory' to indicate any scientific environment where experiments are conducted, be it a physical laboratory in the traditional sense, or a virtual laboratory involving scientists collaborating across the network, simulation-based modeling, etc.

In this paper, we first describe the overall philosophy and architecture of Zoo, thus making it a defining document of the entire project. We then focus on key features of its core components, i.e., the object-oriented database server, the experimentation manager that initiates the execution of experiments as a result of scientists' requests, and the mechanisms that the modules of the system use to communicate between them. Finally, we briefly discuss our experiences with the use of the current Zoo prototype in the context of plant-growth simulation experiments and NMR spectroscopy experiments. Some aspects of the project and some of the Zoo modules emphasizing user interfaces have already been discussed elsewhere: the role of schemas in Zoo [IL92], the theoretical framework used for schema visualization [HIL94] and the resulting prototype schema manager [HIL95, ILBH96], the data model and query language of the system [WI93], and the object-to-file translator [AIL96]. This paper concentrates on the internal engines of the system, which have not been presented earlier. It briefly touches upon those of the above issues that are necessary for a comprehensive understanding of the system, but focuses on the overall system architecture and implementation, and on the technical contributions of the work in the areas of data management and experimentation.

2 Life-Cycle of Experimental Studies

We have been in an on-going dialog with experimental scientists who represent many experimental disciplines: primarily groups in soil sciences and biochemistry, but also physics, genetics, biotechnology, molecular biology, earth sciences, and manufacturing. Although these sciences have very little in common, typical experimental studies in any of them (and even in experimental computer science, as we have experienced it in our own work on DeNet [Liv88] and Condor [LLM88]) seem to go through very similar *life-cycles*. In particular, we have identified the following stages in the typical experiment life-cycle (we have given more details elsewhere [IL92]):

- *Design of Experiment:* The *experimental frame* (i.e., the experiment structure) of a study is laid out [Zei76], specifying which variables will be controlled and what will be measured as output. This is typically done on paper.
- *Data Collection:* Experiments are actually conducted. The scientist specifies the experiment set-up and the values of the input parameters, and the relevant output data is then collected. This is done using some experimentation tool, e.g., a simulator or some laboratory equipment.
- *Data Exploration:* The collected data is studied so that conclusions about the subject of the experiment may be drawn. This is typically done using a variety of systems for retrieving, analyzing, or visualizing the data.

Note that the life-cycle described only captures the activities involved in conducting the experiments and not those involved in preparing the appropriate experimentation tools, e.g., implementing the necessary simulators or setting up the necessary laboratory equipment.

To illustrate the above life-cycle, we discuss experiments in the area of soil sciences, conducted by a group of domain scientists with whom we have been collaborating the longest. They have developed the *Cupid model* [NC83, NC89], which represents an attempt to define collective plant-environment interactions by combining knowledge from the disciplines of meteorology, soil physics, plant physiology, microbiology, entomology, and plant pathology into a single manageable package. Cupid is quite complex (more than 10K lines of Fortran) and is used in about a dozen laboratories in the U.S. and abroad. Typically about a hundred parameters are input to Cupid and over three hundred are received as output for any specific application.

Traditionally, an experimental study using Cupid goes through the following stages. *Experiment Design:* The input and output variables that are important to the study are chosen among all those dealt with by the model. This is done with pencil and paper and the final outcome is kept in notebooks. *Data Collection:* Input files are constructed in the format required by Cupid, containing the combinations of input variables to be tested. Cupid is called on each one of these files, generating each time an output file in a specific format. *Data Exploration:* Unix scripts are written to extract the required data for every different research question that the scientists may have in the course of their study.

A major impediment to exploiting the full power of Cupid has been keeping track of the numerous input and output files that are associated with a study. Over time literally thousands of files are generated, making the task of data exploration a nightmare. Another major problem has been that scientists are forced to use very different tools during each of the three life-cycle stages, making the whole process difficult to manage.

A key objective of our effort has been for Zoo to be an integrated software package with a *uniform* user interface that (a) supports the entire life-cycle of an experimental study allowing smooth transitions between its stages, (b) transparently manages all the data generated by the study, and (c) hides the details of any underlying software used. Another key objective is to blur the separating line between the data collection and data exploration stages, in the sense that data exploration may implicitly involve some data collection. When a scientist is studying a phenomenon, whether a specific piece of information has already been collected or needs to be collected via an experiment is irrelevant. Thus, some requests in the data exploration stage may generate orders for data collection. The following section describes the architecture of Zoo, which has been influenced significantly by the above objectives.

3 Architecture of Zoo

Zoo is designed to be a generic Desktop Experiment Management Environment (DEME). To become a complete Customized Desktop Experiment Management System (CDEMS) and be installed in a specific laboratory, e.g., the Cupid laboratory, it must be enhanced with some custom-made pieces, which can be generated usually with little effort. The overall architecture of Zoo and a resulting CDEMS is shown in Figure 1. Blocks with white background are generic Zoo modules and files; for ease of reference, a short description of these modules is shown in Table 1. Blocks with gray background must be generated separately for each complete Zoo-based CDEMS. Blocks with striped background are external systems with which a given CDEMS needs to communicate. Among them there is *at least* one experimentation system[2], where the experiments are conducted during the data collection stage. In addition, there may be other external systems that are useful in the data exploration stage, e.g., for statistical analysis or visualization.

3.1 Zoo Module Functionality

At the core of the system is *Horse* (*H*eavy-duty *O*bject *R*epository for *S*cientific *E*xperiments), its database server. It is based on the *Moose* (*M*odeling *O*bjects *O*f *S*cientific *E*nvironments) object-oriented data model and the *Fox* (*F*inding *O*bjects of e*X*periments) query language [IL89b, WI93], which we have designed for Zoo. Understanding the rest of the paper requires some familiarity with the Moose data model, so its salient features are briefly described below.

There are various *kinds* of object classes in Moose. The *primitive* classes are *integer*, *real*, *boolean*, and *character-string*. *Tuple* classes have objects consisting of a prespecified number of other objects, called *parts*, identified by labeled relationships. *Collection* classes have objects consisting of an arbitrary number of other objects, all from a single *elements* class. Collection classes are distinguished into *set*, *multiset* (bag), *sequenced-set* (list or array), and *indexed-set* classes. An indexed-set is essentially an array indexed by (the elements of) another arbitrary collection object; the latter is called the *keyset* of the indexed-set and its elements are called *keys*.

There are five *kinds* of binary object relationships in Moose. The structure of a tuple class is defined by an arbitrary number of *has-part* relationships, each pointing to a single object. The structure of a collection class is defined by a single *set-of* relationship, unless it is an indexed-set class in which case its structure is defined by a single set-of and a number of *indexed-by* relationships equal to the dimensionality of the indexed-set. *Association* relationships do not define any structure but simply connect individual objects in two arbitrary classes (of any kind). Finally, an *is-a* relationship between two classes has the usual meaning.

[2]We use the term 'system' in a general sense, to include both software systems and physical systems possibly involving humans in their operation.

Module	Description
EMU	Experimentation manager
FOX	Declarative object-oriented query language
FROG	Visual tool for specifying mappings between Moose objects and Ascii files
HORSE	Object-oriented database server based on Moose and Fox
MOOSE	Object-oriented data model
OPOSSUM	Visual schema manager
SQUID	Visual query manager
TURTLE	Translator between Moose objects and Ascii files

Table 1: Alphabetical list of Zoo modules with short descriptions

Any relationship from a class A to a class B may be specified as *derived*, meaning that for each A object, the related B object is constructed or identified based on other objects that are (indirectly) connected to the A object via other relationships. The construction or identification may be through a Fox query, or may require processing by an external system that receives as input a file containing (parts of) these other objects. Likewise, any subclass may be specified as *derived*, meaning that the members of its superclass that belong to it are identified through some query or other computation.

Figure 2 shows a simple Moose schema in graph form that is used as an example throughout the rest of the paper. It represents a (simplified) soil-science study to determine the total yield and quality of a crop depending on the weather and on how various types of plants are distributed in a large piece of land divided into zones. Each *Experiment* is modeled as a complex object, with sub-objects representing its *Input* and its *Output*. Its output is a pair of the total *yield* and *quality* of the harvest. Its input consists of the *Weather* and a *Plant_community*, which is an indexed-set of *Plants* indexed by the set of land *Zones* so that the zone where each plant is grown is recorded independently for each Plant_community in which the plant participates. The weather is captured by *rainfall, temperature*, and *wind-speed* values, and may be *windy* (derived as wind-speed $>$ 30 mph), in which case *wind-direction* becomes important as well, *dry* (derived as rainfall $<$ 2 in), in which case air *humidity* becomes important as well, or *disaster*, which combines the two. (The derivation conditions are not shown.)

Note that the output of an experiment is indicated as a derived relationship (label (D)). Although not shown in the figure, the derivation is based on the input part of an experiment (in particular, the values in the primitive leaf classes of that complex object class) and is realized by the execution of an external program (e.g., Cupid, although Cupid deals with much more complex experiments).

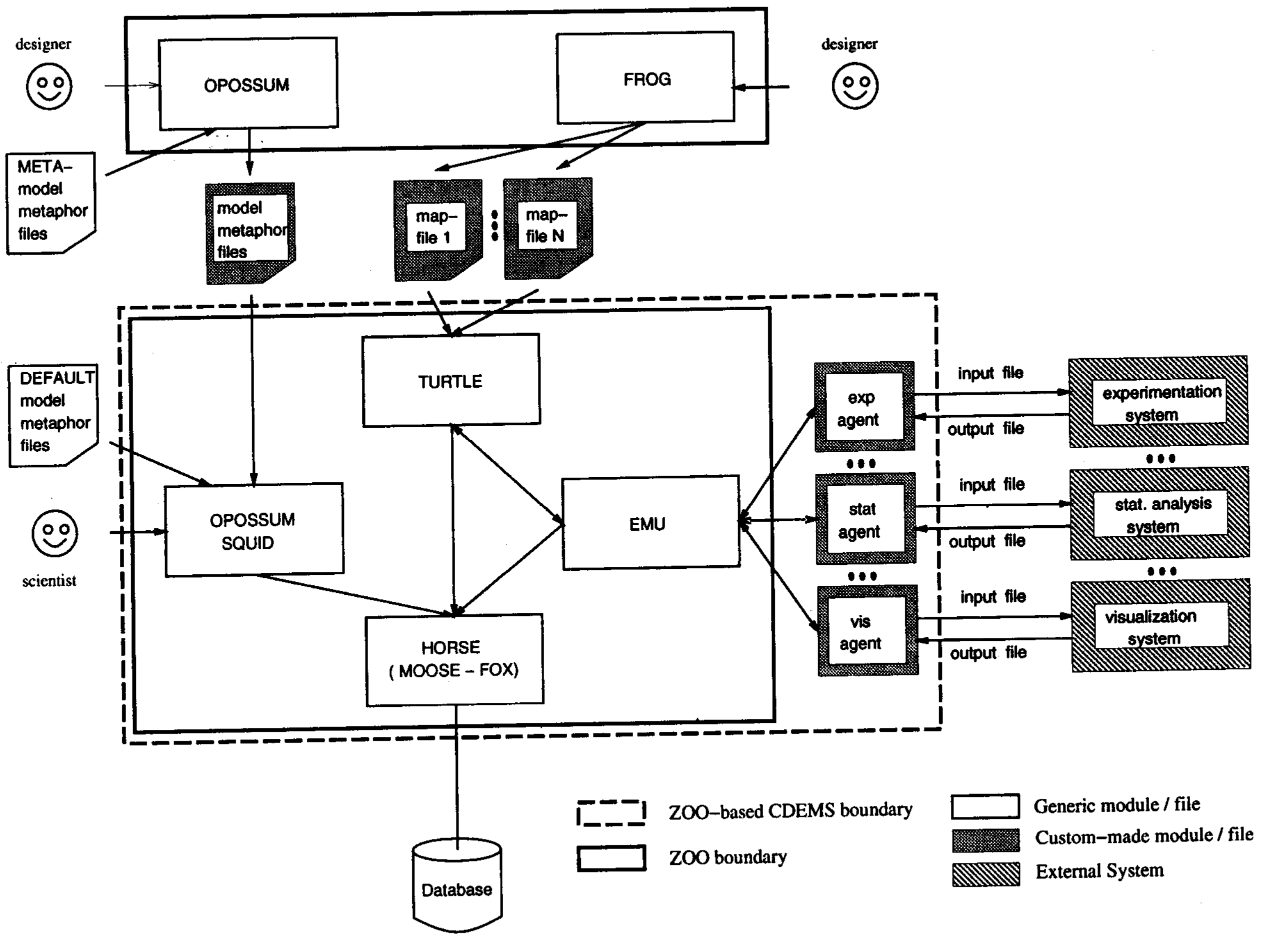


Figure 1: Overall architecture of Zoo

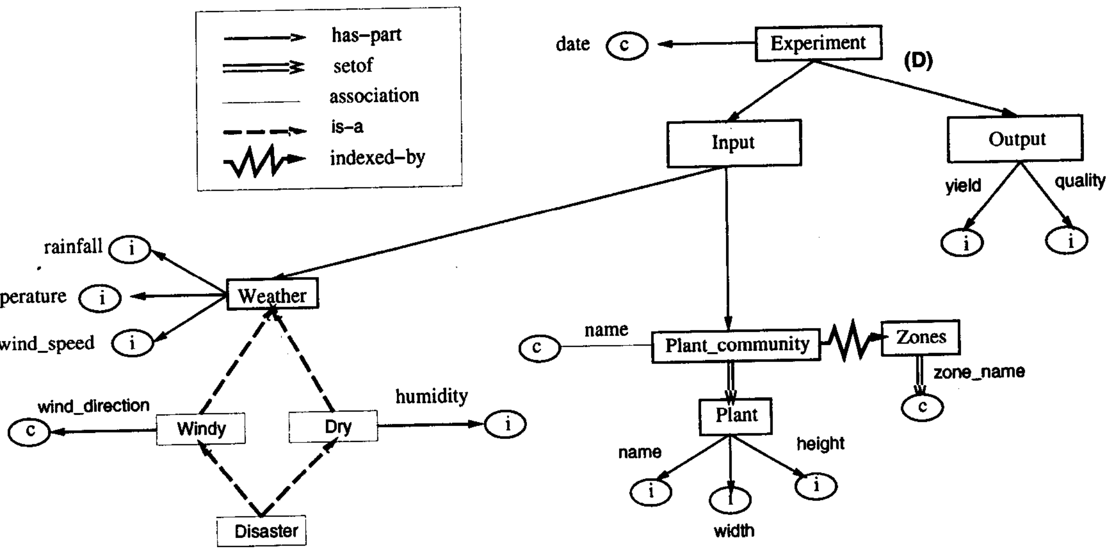


Figure 2: Sample Moose schema of Soil Sciences experiment

Opossum (*O*btaining *P*resentations *O*f *S*emantic *S*chemas *U*sing *M*etaphors) and *Squid* (*S*ystem for *Q*ueries *U*pdates *I*nsertions *D*eletions) make up the user interface of Zoo. Opossum is a schema manager [HIL95] and Squid is a query/update manager. Opossum and Squid have been built following a visualization framework that we have developed that separates the data domain from its visualization [HIL94] for maximum flexibility. In particular, they are *generic* visual systems whose inputs are files with specifications of a data model or query language (which are always Moose and Fox for Zoo, respectively), a visual model, and a metaphor that indicates the correspondence between visualizations and underlying schemas or queries. For example, the most useful visual model for Moose schemas is that of graphs and the most useful corresponding metaphor maps graph nodes to Moose classes, graph edges to Moose relationships, etc. (Figure 2 presupposes such a metaphor.) Receiving these, Opossum is customized to operate for the specific data model and visualization style. (Likewise for Squid.) Although for Zoo the data model and query language are fixed, Opossum and Squid still offer much flexibility in defining different ways in which schemas or queries may be visualized.

Emu (*E*xperimentation *M*anagement *U*nit) is responsible for transforming user requests into actions at external systems and preparing everything necessary for these actions. It interacts with Horse for retrieving the necessary user requests and with custom-built *agents*, one for each external system that the specific Zoo-based CDEMS needs to communicate with. It also interacts with Turtle, to which it delegates the necessary object-to-file translations.

Turtle (*T*ranslation *U*nit of *R*un *T*ime of *L*arge *E*xperiments) is the system's translator from Moose objects to Ascii files and vice versa. It is also a generic module; it receives as input a *map-file* that contains specifications of how the various parts of a complex object correspond to the various areas of an external file, and based on that, it performs the actual translations.

Frog (*F*iles *R*elated to *O*bjects *G*raphically) is a visual tool for generating the map-files required by Turtle. In one window, it has a *sample* (input or output) file of the external system, and in another, it has the Moose schema for the experiment concerned, managed by Opossum. By highlighting a specific area in the file and clicking on the appropriate part of the schema, the designer specifies what objects correspond to what area of the file. Printing details, e.g., precision, are guessed by Frog and can be overwritten by the user.

3.2 Installation of Zoo-based CDEMS

Each external system with which communication is desired may have specialized usage requirements that are impossible to include in a generic system. Thus, installation of a Zoo-based CDEMS in a specific laboratory or for a specific study entails a customized enhancement of the system:

- For each external system of interest, a customized *agent* is built incorporating all the specific details required for interacting with and monitoring the system. The agent is registered with Zoo and information about it is stored in a system-defined class (Section 6).

For example, our soil-science collaborators want to execute the Cupid simulator under Condor [LLM88], so the corresponding agent takes care of all the Condor communication. On the other hand, our biochemistry collaborators run experiments on spectrometers that are operated by humans, so the corresponding agent uses electronic mail to send the appropriate messages to designated technicians and to collect the output from designated files. Agents for new external systems can be built and registered dynamically, even after experiments have been run.

Both modules in the user-interface of Zoo, Opossum and Squid, are generic and need some customized input to become operational. Graph-based visualizations of schemas and queries are universally useful and intuitive, so the appropriate model and metaphor files have been constructed and are provided for scientists to use directly. If, however, the scientists desire different schema or query visualizations, then the following activity becomes necessary at installation time as well:

- Opossum is used as a meta-schema manager to create the files containing the data and visual models and metaphors required by the scientists. Meta-model and metaphor files that are "manually" constructed for this bootstrapping process are used as input to Opossum. The resulting meta-schemas are then stored in files and used as Opossum and Squid inputs [HIL95].

New visual models and metaphors can also be designed dynamically.

Note that, unless new visual models and metaphors are desired, installation only requires some programming in a regular programming language to build the agents but no database expertise, which is one of the goals of our effort.

3.3 Experiment Design

During the design stage of an experiment's life-cycle,

- Opossum is used to specify the schema of the experiment, which is then used to generate a database under Horse. This schema contains derived relationships corresponding to external systems associated with the given Zoo-based CDEMS.

Since Turtle is also a generic module, to be able to perform the appropriate translations between objects and files, it needs some customized input. Therefore, experiment design includes the following activity as well:

- Frog is used to specify mappings between the designed Moose schema and the input and output files required by each external system. The resulting map-files are then stored and used as Turtle input [AIL96].

Again, no real database expertise is required for experiment design, as both Opossum and Frog are visual tools offering a high-level intuitive interface.

3.4 Data Collection and Exploration

As mentioned in Section 2, an important feature of Zoo is that it blurs the distinction between the data collection and data exploration stages if the scientist so desires. In particular, the scientist may use Squid to request results of experiments without any knowledge of whether they have been run yet or not. When they have been run, Horse retrieves the necessary information from its database and returns it to Squid for display. When not, Horse invokes the mechanism for dealing with derived relationships (recall that the output of an experiment is derived), which eventually triggers the necessary actions at the appropriate external system.

4 Information Storage and Shipping

A Zoo-based CDEMS deals with a wide variety of information types, i.e., the contents of the database associated with each experimental study, results of queries that have been posed to a database and are important enough for the scientific study to store away, the queries themselves, visual representations of the database schemas, and eventually maybe even the models and metaphors required by Opossum or the map-files required by Turtle. For uniformity, genericity, and extensibility of storage by Horse and of communication between the various Zoo modules, Zoo views *any* piece of information of *any* of the above flavors as an object in a Moose database [IL89a]. This is done recursively, in the sense that the schema of each one of these databases is also an object in a different, higher-level meta-database, until some *root* databases are reached, whose schemas are known to all modules (Horse in particular) in a hardwired fashion. There are three root databases: one storing all user-defined Moose schemas in the Zoo-based CDEMS, one storing all saved Fox queries, and one storing all visual models that are used for visualization. Figure 3 shows the schema of the Moose-schema database. We do not present the other two, since we have not described Fox or the visualization methodology in any detail for these schemas to be comprehensible. Figure 3 is self-explanatory (the attributes of Relationship capture its kind, its forward and reverse labels (fname and rname), and its forward and reverse properties represented as a bitmap (fprop and rprop), e.g., cardinality, mutability, etc.). Note that derivation rules are part of the schema and each one is associated with one of the relationships. Currently, derivation rules are represented as plain text, but in principle, one could develop a schema rooted at the DerivationRule class capturing the rules' structure.

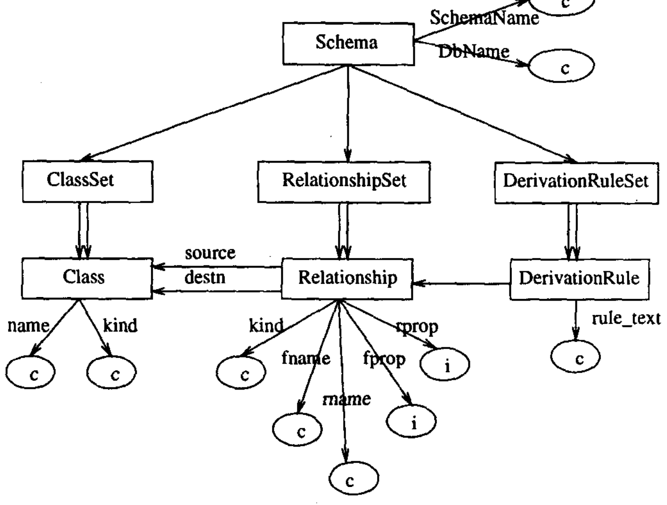


Figure 3: Meta-schema of user-defined Moose schemas

There are several important implications of the overall approach above with respect to information storage:

1. Schemas are a commodity whose usage goes way beyond the typical one in the context of database systems. Scientists can define them, examine them, or modify them, thus affecting the experimental frame of their studies, without thinking about them as expressing database structures. In fact, a schema does not necessarily have to be associated with a database, but may exist uninstantiated.

   A schema of one database can be simply copied and be used as part of another schema. It is common for experimental studies in the same field to share pieces of their overall setup, so during experiment design, it is very convenient for scientists to simply take and reuse an existing schema. In fact, even subschemas can be reused this way. For example, several Cupid studies may need the exact same modeling of weather as in the schema of Figure 2; scientists can simply copy that subschema to their own and avoid the trouble of redesigning it.

2. Whole databases can be made part of larger databases, as subobjects. In fact, databases can be shared by multiple larger databases that need to use the contents of these subdatabases without replicating them. Similarly to point 1 above, experimental studies in the same field often share not just their design but even some of the data that they use as input. This is often the case in modeling studies (simulation studies), where part of the input is actual measurements or observations. Then, specific datasets that have been collected at some point and have properties that are well-known tend to be used continuously, so database sharing comes very handy and saves much space. Continuing on with the example in point 1, a study may need

not only the schema of the Weather class, but even the actual data with which this class and its subclasses have been populated (because, say, these represent actual weather characteristics observed in a specific geographic area of interest). This can be seen as a simple special case of schema/database merging in heterogeneous databases [KLK91] where there are no semantic inconsistencies in the merged schemas, all of them are in the same data model, and they are reused without any transformation.

Expanding this notion of database sharing, Horse permits scientists to pose requests that span multiple databases. Scientists can bring up through Opossum or Squid (visualizations of) any number of schemas and then pose queries on them. In some sense, this is like generating an ad hoc new database that has the actual databases as parts. This capability is very important, as scientists often want to access the results of multiple separate studies to correlate or combine their results. For example assume that two different studies have the exact same schema of Figure 2. One may be a study that has used the Cupid simulator, while the other may be a study actually done on the fields containing actual measurements. Scientists may then request pairs of yield values for every experiment that has the same input. This would allow them to validate Cupid and calibrate it appropriately if any major discrepancy arises in the results of the two studies.

3. Queries are a commodity as well, and can be saved and reused at will. A query that resulted in data that leads to interesting insights needs to be saved (together with its result), for later reuse, to serve as a starting point for modifying it to obtain other similar queries, or to be eventually reported in scientific publications. Moreover, it can be used in the context of other studies hoping to generate similar insights, to identify correlations between results of different studies, etc.

4. Multiple visualizations can be used for the same schema, query, or object. For example, three separate visualizations could be stored for the schema captured in Figure 2: the specific graph layout of the figure, a completely different graph layout (possibly generating different intuitions about the experiment design), and some other non-graph representation. Scientists can switch among these depending on their aesthetic preferences or needs and the appropriate visualization would come to the screen.

5. Visualizations are a commodity as well and can be manipulated accordingly. All schema and query copying and sharing in points 1 and 3 above are in fact done using Opossum and Squid through visualizations, which are copied and shared as well. Likewise, visual models for schemas, queries, or object are a commodity that can be reused. For example a visual model for graphs generated to capture Moose schemas, may be useful in capturing many other types of objects as well. The designer can simply copy it instead of reconstructing it.

The above schema/meta-object approach has important implications on information transfer between Zoo modules as well. Everything that is shipped is an object in some schema that has also been specified in the same or an earlier shipment. Schemas, queries, objects, visualizations, etc., everything is shipped as an instance of a known schema. By receiving the schema before the object, the recipient module has all the necessary information to interpret the object appropriately. Thus, all Zoo modules communicate with each other following the exact same generic protocol, using the exact same code, which we call *shipping* code.

The shipping code has been developed so that each module could run on a separate machine, communicating via messages. Thus, on both ends of a communication link between modules, there are transformation mechanisms between the internal representation used by the module and the Ascii representation required by the shipping code. This has given great portability to Zoo, since it does not depend on any machine-specific characteristic for communication.

Since the shipped objects may be quite complex (essentially, arbitrary Moose objects), their shipping (Ascii) representation is quite flexible. Essentially, an object to be shipped is an array with an entry for each individual object that is part of the overall object shipped. The relationships between these objects are captured as indices into the array rather than memory pointers as these have to be shipped across the network. Figure 4 shows the full structure of an input object based on the schema of Figure 2 in graph form. This is similar to the schema graph, so what the actual object is should be clear. Figure 5 shows the corresponding shipping representation for that object.

5 The Horse Database Server

Horse is the backbone of the entire Zoo environment, being a server for all other modules[3]. It has been developed in a layered fashion for flexibility and extensibility. At the front-end, it accepts Moose data definition requests and Fox query and update requests in shipping form, but can also operate in a stand-alone mode, accepting textual requests. At the back-end, it is currently using the Informix relational database system as a storage server. There are many reasons for the Informix choice. First, we wanted a commercial piece of software that was unquestionably reliable so that scientists would feel comfortable storing their data under it. Second, we preferred a relational over an object-oriented database system (which would have a data model conceptually closer to Moose) because the OO systems that existed when we started required that the schema of a database

[3] It is truly a workhorse!

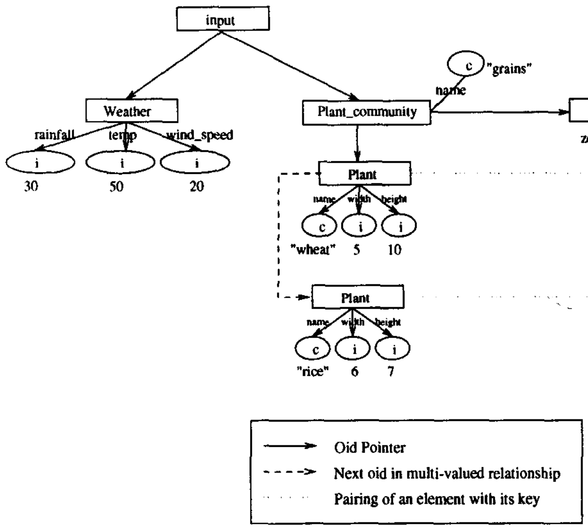
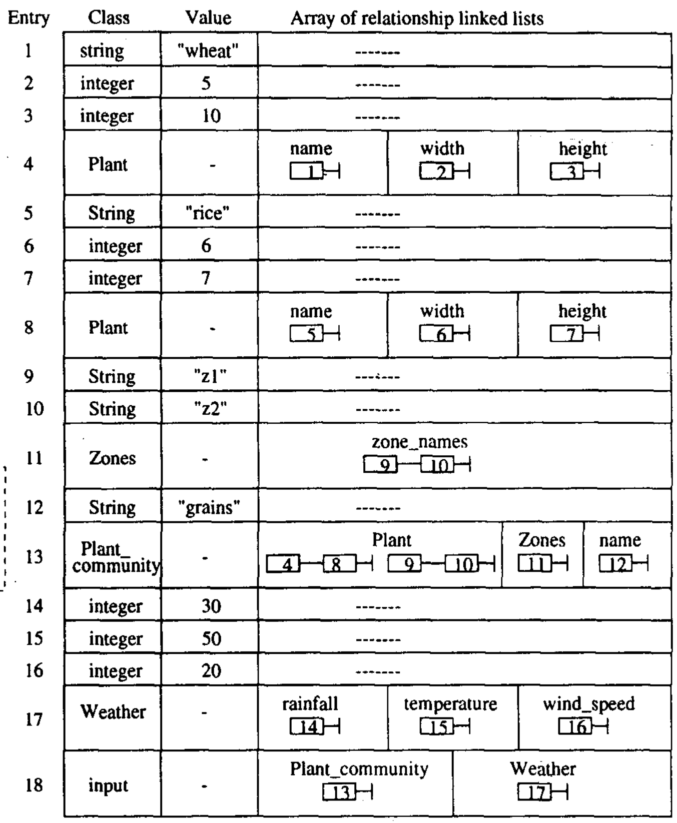


Figure 4: Graph representation of input object



Entry	Class	Value	Array of relationship linked lists
1	string	"wheat"	-------
2	integer	5	-------
3	integer	10	-------
4	Plant	-	name: 1; width: 2; height: 3
5	String	"rice"	-------
6	integer	6	-------
7	integer	7	-------
8	Plant	-	name: 5; width: 6; height: 7
9	String	"z1"	-------
10	String	"z2"	-------
11	Zones	-	zone_names: 9 – 10
12	String	"grains"	-------
13	Plant_ community	-	Plant: 4 – 8, 9 – 10; Zones: 11; name: 12
14	integer	30	-------
15	integer	50	-------
16	integer	20	-------
17	Weather	-	rainfall: 14; temperature: 15; wind_speed: 16
18	input	-	Plant_community: 13; Weather: 17

Figure 5: Shipping representation of same object

be essentially compiled with any code developed on top of them, which is clearly inadequate for a dynamic environment like Zoo where many experiment/schema designs will be defined over time. Third, Informix was donated to us for free! Our experience with Informix has been extremely positive, so we are happy that we made that choice. Moreover, thanks to the layered implementation of Horse, we can easily port the system to use as a storage server any other relational or non-relational system. (We have taken some initial steps in this direction, to port Horse on top of Shore [C+94].)

The most significant function of Horse is to translate Moose and Fox into the Relational model and SQL, respectively. Most aspects of these translations are quite straightforward and similar to other efforts. The interesting parts are those that deal with unique or uncommon Moose and Fox features: sets and indexed-sets (which are very important in scientific experiments) and deep path expressions. Due to lack of space, we do not present any details on these translations, which can be found in the extended version of this paper.

6 Experimentation Management

As mentioned earlier, in Zoo, experimentation and any other form of external processing is achieved through the derived relationship mechanism. To support this mechanism, Horse uses two system-defined classes, the *Task class* and the *Agent class*, whose schemas and their relationship are shown in Figure 6. At any point, the Task class contains one object for each external computation/activity scheduled. It records a status code for the computation (waiting-to-be-scheduled, scheduled, and complete), the class of the output object (an object in the schema database (Figure 3)), the oid of the object whose relationship is derived (embedded in the FullId class, which we use to express objects of any class), and a list with the oids of the input objects (whose parts will be placed in the input file). The Agent class contains one object for each registered agent. The attributes of the class include the agent's name (which is what Emu uses to call the agent for execution), the map-file names for the input and output files for the agent's corresponding external system, and possibly the map-file name for the file returned with status information from that external system during processing.

When Horse receives a request for data that requires some experimentation (or other processing) at an external system, it generates an object in the Task class and populates its parts with all the necessary information for the experiment to run. Emu periodically inquires Horse and whenever it finds new objects in the Task class, it initiates the corresponding experiments. Specifically, it first calls Turtle and passes to it the oid(s) of the object(s) that capture the experiment's input as well as the name of the appropriate map-file that has been generated during experiment

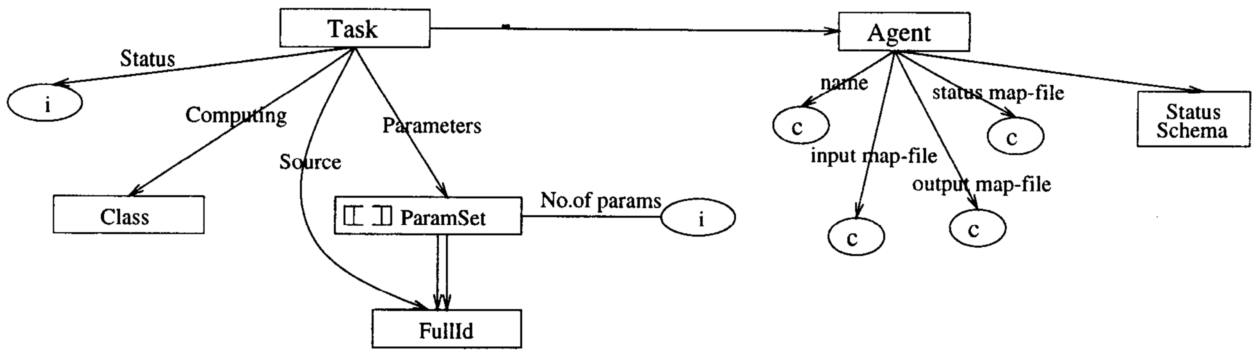


Figure 6: Schemas of the Task and Agent classes

design by Frog, all of which are found by Emu in the Task and Agent objects related to the experiment. Turtle interprets the map-file, uses the given object oid(s) to call Horse and extract the needed data from the database, and eventually constructs the appropriate input file. Then, Emu communicates with the appropriate agent sending to it the input file, and the experiment starts. Periodically, Emu polls the agents and when it detects that the experiment is over, it calls Turtle again and passes to it the name of the output file generated by the experimentation system and the name of a different map-file that has also been generated by Frog earlier. Turtle now operates in the opposite direction, interprets the map-file, and constructs database object(s) from the output file, which are then stored in the database and possibly sent to Squid for visualization as well.

During an experiment's execution, status requests by the scientists are supported through the last relationship of the Agent class. Specifically, registration of an agent may also involve the specification of a schema for status information, which in general would be different for various agents. To accommodate this schema, conceptually, the class that represents the external derivation process is accompanied by a shadow subclass, which is *not* part of the scientist's schema and is therefore not normally visible. At any point, the shadow subclass contains only objects representing derivations that are in-flight, which are removed when the derivation process completes. The root of the status schema is connected to the shadow subclass via a derived relationship. Any status query triggers the appropriate derivation, just like any other external request, and results in a status call to the appropriate agent and eventually to the external system. Since status schemas are (on purpose) not visible, queries requesting the output of an experiment when that output has the special value *inflight* are interpreted as status queries. Horse redirects the query to the shadow subclass and modifies it to request a status object instead of an output object.

For example, consider the schema in Figure 2. The Experiment class captures external computations and its relationship to the Output class provides connection to the results of these computations. Ignoring the structure below the Input class, Figure 7 shows the same schema enhanced with the shadow subclass IF_Experiment ('IF' for In-Flight) and a simple status schema connected to it. The enhancement is shown in dashed lines, as it is not normally visible. In the example, the status consists of the experiment's elapsed time, the last stage in the external process that has completed, and a prediction of the amount of time required for final convergence. Any query requesting for the output object of some experiment that is still in progress will return a status object consisting of the three pieces of information mentioned above.

7 Status and Experience

Zoo is being implemented in C++. Not counting any visual libraries (Interviews, Tcl/Tk) or database libraries (Informix) that it uses, Zoo is currently approximately 144K lines of code, with about 51K in Horse and the shipping code, 74K in Opossum, 8K in Squid (not counting the Opossum code it uses), and 11K in Frog, Turtle, and Emu. An initial version of the system is operational and used for testing by experimental scientists. Some of the functionality described in this paper as part of the system's design is still under development: only the Moose metaschema hierarchy of Section 4 is supported within Horse; Squid provides a visual language that captures only a subset of the expressive power of Fox; the meta-model and metaphor files are not complete for Opossum (Figure 1), so models and metaphors are specified in text form; the Emu and agent functionality is provided by the same module and so a Zoo-based CDEMS only works with a single external system at a time.

Zoo has been successfully tested by the Cupid group for experimentation. A custom-built Emu/agent combination has been implemented as a single module, for communication with the Cupid simulator. The resulting CDEMS has been used to drive test runs on Cupid. The interface offered by Opossum for experiment design has played a key role in the positive reception the system has had. Our soil-science collaborators have been able to learn the tool and then organize and layout large experiment schemas with hundreds of classes within a few hours. (The input part of the full Cupid schema alone has 159 classes.) The tool has also brought substantial improvements in other aspects of the scientists'

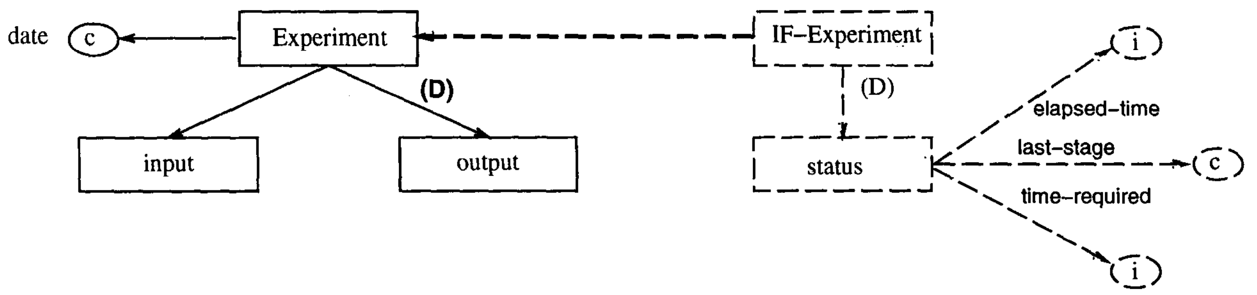


Figure 7: Schema enhanced with shadow Experiment class to capture status

work. For example, before Opossum, the Cupid group had to refer to the input data file to a Fortran program, which had grown increasingly fractured and confusing over the years. They now use the corresponding visual schema as their reference in thinking about the model, planning experiments, and explaining the model and experiments to other scientists. Likewise, not having to deal directly with large numbers of input and output files is considered a great benefit of using Zoo.

In addition to the Cupid group in Soil Sciences, Zoo has also been tested by a collaborating group in Biochemistry. A Zoo-based CDEMS has been partially developed for experiments run on various spectrometers. Test experiments have been designed with Opossum (having in the order of 350 classes in the corresponding schemas), customized Emu/agent modules have been developed, and object-to-file translations have been specified with Frog. Tests with the system have been successfully run and the reaction of the scientists involved has been very positive again. The same group has also used Opossum to design a large relational schema for the Biological Magnetic Resonance Databank, an international repository of data on biological macromolecules derived from NMR spectroscopy.

Finally, Opossum has been used as a stand-alone schema manager for the relational and E-R data models, using a variety of visual models and metaphors. In fact, customized (through its input files) to visualize E-R schemas as the traditional E-R diagrams, Opossum is currently being used in our database courses as a database design tool.

8 Current Practice and Related Work

Currently, many scientists use notebooks to keep track of where data is stored and search these notebooks manually whenever they need to retrieve data. They have to write application programs in procedural languages in order to access the data, and for every different type of question they want to explore, a different program has to be written. In some cases, relational database systems are used, but what they can offer is not adequate. Declarative textual languages do not correspond to the intuition of scientists and thus are hard to use. For example, writing an SQL query that retrieves information regarding a phenomenon that has hundreds of parameters is a very time consuming process. Scientists must look around the catalogs to identify the appropriate relations and the query must be spelled out, which involves specifying a large number of joins and selections. Finally, in some cases, scientists use specialized software tools developed just for experiments in their field. For example, CERN (European Center for Nuclear Physics) has developed PAW [BCVZ88] and HEPDB [Shi93], which are currently used by high-energy physicists to deal with data from accelerator experiments. Tools like these, however, usually lack much of the desired functionality that Zoo intends to offer, especially the visual style of user interaction, the ability to communicate with several external systems from the same interface, and of course genericity. Thus, the current state of affairs forces scientists to spend time on learning data access and manipulation software systems, when they should be focusing on scientific analysis.

There are a few other projects that share some of the goals of Zoo. These include the "Laboratory Notebook" effort at Los Alamos [Nel90], the "Chromosome Information System" (CIS) database at LBL supported by the SDT [MF91] and ERDRAW [SM91] design tools, the OPM model and tools at LBL as well [CM95], the Computational Chemistry Database project at OGI [CMR+94], the "Software Testpilot" project on DBMS performance assessment at CWI [KK93], and others. The focus of each project is different, so comparisons with Zoo are not always meaningful. In general, the most important aspects of Zoo are its generic nature, its flexible schema/meta-object hierarchy for all forms of information, its powerful data model and query language, its emphasis on visual interaction based on declaratively specified metaphors, and its open architecture to multiple external systems.

With respect to experimentation management, the OPM data model [CM95] employs special *protocol* classes to capture the flow of experiments, something that Zoo is capturing indirectly through the derived relationships in Moose. The relative expressibility of the two approaches is a question that we plan to investigate in the future. The *computational proxies* mechanism [CMR+94] addresses the issue of finding out what input objects are the output objects of an experiment associated with. Within Zoo, this is handled by the translation mechanisms of Turtle.

Finally, with respect to data management independent of experiment management, Horse offers some unique characteristics not found elsewhere. In addition to the novel features of Moose and Fox [WI93], the view of schemas as commodity objects, separate from their instantiations, opens up many new opportunities for information sharing and system communication. This also holds for all the other commodity objects discussed in Section 4 enabling varied visualizations for the same information, experiment result sharing, etc. We are not aware of any other system following this approach. All database systems store their schemas in catalogs as data, but the catalogs are known only to the particular database.

9 Conclusions

The Zoo effort has been driven by two major forces: a) to advance the state of the art in experiment management technology and b) to enhance the productivity of many experimentation laboratories. The Zoo environment that we have discussed in this paper represents the contributions that we have made in (a), while the positive feedback that we have been getting by our collaborating scientists testing the system captures the achievements in (b).

There are several issues that we plan to continue working on in the future, enhancing the performance, functionality, and applicability of Zoo. In addition to completing its implementation based on this papers' description, some of the other important issues are porting Zoo from on top of Informix to on top of Shore [C+94] (so that the use of our tool does not depend on the purchase of a commercial DBMS), porting Opossum and Squid to Tcl/Tk or Java, and providing the ability to express arbitrarily complex experiment protocols that may involve communicating with several external systems as part of a single experiment instance. When the system is thoroughly tested, we intend to distribute it over the network to external laboratories.

Acknowledgements. We would like to thank all those who have been or currently are associated with Zoo: Natassa Ailamaki, Martha Anderson, Vaishnavi Anjur, David Argentar, Jian Bao, Roger Chylla, Eben Haber, Murali Krishnan, John Markley, David Melski, Renee Miller, John Norman, Andy Therber, Odysseas Tsatalos, Tom Wang, Janet Wiener, and Eldon Ulrich.

References

[AIL96] V. Anjur, Y. Ioannidis, and M. Livny. Frog and Turtle: Visual bridges between files and object-oriented data. In *Proc. 8th International Conference on Scientific and Statistical Database Management*, pages 76–85, Stockholm, Sweden, June 1996.

[BCVZ88] R. Brun, O. Couet, C. Vandoni, and P. Zanarini. Physics analysis workstation. Technical Report Program Library Q121, CERN Computer Center, 1988.

[C+94] M. Carey et al. Shoring up persistent applications. In *Proc. of the 1994 ACM-SIGMOD Conference on the Management of Data*, pages 383–394, Minneapolis, MN, May 1994.

[CM95] I-M. Chen and V. Markowitz. An overview of the object protocol model (opm) and the opm data management tools. *Information Systems*, 20(5):393–418, July 1995.

[CMR+94] J. Cushing, D. Maier, M. Rao, D. Abel, D. Feller, and D. DeVaney. Computational proxies: Modeling scientific applications in object databases. In *Proc. 7th International Conference on Statistical and Scientific Database Management*, Charlottesville, VA, September 1994.

[FJP90] J. C. French, A. K. Jones, and J. L. Pfaltz. Summary of the final report of the NSF workshop on scientific database management. *ACM-SIGMOD record*, 19(4):32–40, December 1990.

[HIL94] E. Haber, Y. Ioannidis, and M. Livny. Foundations of visual metaphors for schema display. *Journal of Intelligent Information Systems*, 3(3/4):263–298, July 1994.

[HIL95] E. Haber, Y. Ioannidis, and M. Livny. Opossum: Desk-top schema management through customizable visualization. In *Proc. 21st International VLDB Conference*, pages 527–538, Zurich, Switzerland, September 1995.

[IL89a] Y. Ioannidis and M. Livny. Data model mapper generators in observation dbmss. In *Proc. Workshop on Heterogeneous Database Systems*, Chicago, IL, December 1989.

[IL89b] Y. Ioannidis and M. Livny. MOOSE: Modeling objects in a simulation environment. In G. X. Ritter, editor, *Information Processing 89*, pages 821–826. North Holland, August 1989.

[IL92] Y. Ioannidis and M. Livny. Conceptual schemas: Multi-faceted tools for desktop scientific experiment management. *Journal of Intelligent and Cooperative Information Systems*, 1(3):451–474, December 1992.

[ILBH96] Y. Ioannidis, M. Livny, J. Bao, and E. Haber. User-oriented visual layout at multiple granularities. In *Proc. 3rd International Workshop* *on Advanced Visual Interfaces*, pages 184–193, Gubbio, Italy, May 1996.

[KK93] M. L. Kersten and F. Kwakkel. Design and implementation of a dbms performance assessment tool. In *Proc. 4th International DEXA Conference*, pages 265–276, Prague, Czech Republic, September 1993.

[KLK91] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with semantic discrepancies. In *Proc. ACM-SIGMOD Conference on the Management of Data*, pages 40–49, Denver, CO, May 1991.

[Liv88] M. Livny. *DeNet User's Guide, Version 1.0.* Computer Sciences Dept., University of Wisconsin, Madison, March 1988.

[LLM88] M. Litzkow, M. Livny, and M. W. Mutka. Condor - a hunter of idle workstations. In *Proc. of the 8th International Conference on Distributed Computing Systems*, San Jose, CA, June 1988.

[MF91] V. M. Markowitz and W. Fang. SDT - a database schema design and translation tool. Technical Report LBL-27843, Lawrence Berkeley Laboratory, Berkeley, CA, May 1991.

[NC83] J. M. Norman and G. S. Campbell. Application of a plant-environment model to problems in irrigation. In D. I. Hillel, editor, *Advances in Irrigation*, volume II, pages 155–168. Academic Press, New York, NY, 1983.

[NC89] J. M. Norman and G. S. Campbell. Canopy structure. In R.W. Pearcy et al., editors, *Physiological plant ecology: Field methods and instrumentation*, pages 301–325. Chapman Hall, Ltd., London, UK, 1989.

[Nel90] D. Nelson. The laboratory notebook technical manual. Technical Report LA-UR 88-1256, Los Alamos National Laboratory, Los Alamos, NM, 1990.

[Shi93] J. Shiers. High-energy physics data base. Technical Report Program Library Q180, CERN Computer Center, 1993.

[SM91] E. Szeto and V. M. Markowitz. ERDRAW - a graphical schema specification tool. Technical Report LBL-PUB-3084, Lawrence Berkeley Laboratory, Berkeley, CA, May 1991.

[SOW84] A. Shoshani, F. Olken, and H. K. T. Wong. Characteristics of scientific databases. In *Proc. 10th International VLDB Conference*, pages 147–160, Singapore, August 1984.

[WI93] J. Wiener and Y. Ioannidis. A Moose and a Fox can aid scientists with data management problems. In *Proc. 4th International Workshop on Database Programming Languages*, pages 376–398, New York, NY, August 1993.

[Zei76] B. P. Zeigler. *Theory of Modeling and Simulation.* John Wiley & Sons, New York, N.Y., 1976.