A Moose and a Fox Can Aid Scientists with Data Management Problems [*†]

Janet L. Wiener
Yannis E. Ioannidis
Dept. of Computer Sciences, University of Wisconsin-Madison
1210 W. Dayton St., Madison, WI 53706 U.S.A.
{wiener,yannis}@cs.wisc.edu

Abstract

Fox (Finding Objects of eXperiments) is the declarative query language for Moose (Modeling Objects Of Scientific Experiments), an object-oriented data model at the core of a scientific experiment management system (EMS) being developed at Wisconsin. The goal of the EMS is to support scientists in managing their experimental studies and the data that are generated from them.

Moose is unique among object-oriented data models in permitting sets to have relationships to classes other than their elements' class, in providing a construct for indexing collections by other collections, such as time series, and in distinguishing structural relationships from non-structural ones.

Fox contains several new features necessary to manage experiments, such as support for associative element retrieval from (indexed) sets and highly expressive path expressions. Fox path expressions can traverse any relationship in the schema graph, including inheritance relationships, and in either direction of the relationship, which makes many queries more concise. Fox also supports a new form of deep equality based on structural information and a new, concise, description of periodic data, e.g., time series. Finally, Fox offers the only object-oriented bulk-loading facility of which we are aware, for loading data from a file.

1 Introduction

Scientific databases are expected to play an important role in enabling scientists from any discipline to study complex phenomena and systems of interest. We are currently involved in the development of a desktop *Experiment Management System (EMS)*, whose primary goal is to support individual (or small teams of) scientists in managing their experimental studies and the data that are generated from them [IL92, ILH+93].

As one of its components, the EMS under development includes an Object-Oriented Database Management System (OO-DBMS). Due to the special needs of many experimental sciences, we have developed our own data model, Moose (Modeling Objects Of Scientific Experiments), and declarative query language, Fox (Finding Objects of eXperiments). In this paper, we briefly describe the salient features of Moose and then focus on the most interesting aspects of Fox. More details on Moose and Fox are available elsewhere [WI93], as are descriptions of earlier versions of Moose [IL89, IL92].

Several characteristics of the data expected to be found in scientific experiments and of the ways scientists are expected to interact with an EMS led us to develop Moose and Fox. Specifically, a new data model was desirable because of the following:

[*]A shorter version of this paper appears in the Proceedings of the 4th International Workshop on Database Programming Languages, New York, NY, August 1993.

[†]This work has been partially supported by the National Science Foundation under Grants IRI-9224741 and IRI-9147368 (PYI Award) and by grants from DEC, IBM, HP, and AT&T.

- In many experimental studies, object collections (e.g., sets or multisets) are reused several times during the course of the study. In addition, they are often associated with other pieces of information, which may or may not depend on the contents on the collection, e.g., the number of objects in the collection or a name given to the collection. To serve these needs, Moose collections are individual objects that may be directly associated with objects besides their elements.

- Distinguishing the structural components (parts) of a complex object from any other objects with which it is associated is important to scientists. Such distinction captures the semantics of experiments, or any other type of information, more naturally [MF91, WCH87]. The additional semantics expressed by this distinction can be used in several ways. First, the structural components of a complex object may be seen as defining its scope, which determines certain properties of the object. For example, object comparisons may be made based only on the objects' structural components, and immutability of an object's relationships may be recursively propagated along structural components only. Second, incompletely specified path expressions in the query language may be disambiguated based on the differences in the two types of relationships [Las93].

- Scientists often need to represent collections that are indexed by other, arbitrary, collections, e.g., indexed by time series. Arrays (indexed by the set of consecutive integers $\{1, \ldots, n\}$, for some n) are the only special case of indexed collections supported by existing OO data models. Moose provides a direct construct to support arbitrary indexing.

- A large portion of the data relevant to scientific experiments is derived from other data based on some computation. In most current OO DBMSs, such computations are expressed as methods in the programming language supported by the system. In Moose, rules written in Fox may be used to express such computations, e.g., to specify aggregate computations over the members of collection objects. The same mechanism can also be used to implicitly identify interesting subclasses of a given class of objects.

Nowadays computation is part of many experimental studies, and most scientists (or at least their graduate students!) feel comfortable using some programming language, usually Fortran. The ability to use declarative queries for data retrieval and simple computations on the retrieved data is very appealing to many of them. Fox is the declarative query language that we developed for Moose; we also plan to develop an equivalent graphical query language, which we believe will be even more intuitive to use. Because Fox is completely declarative, queries are simple to write and have much optimization potential. Fox supports several interesting features that were motivated by the novel features of Moose and the needs of experimental scientists for interacting with the database. These needs include the following:

- Scientists need to access experimental data in ad-hoc ways. Most current OO query languages impose restrictions on how object relationships may be traversed in query path expressions, for instance, only allowing traversal in one direction [BCG$^+$87, CDV88, DGJ92, FBC$^+$90, NO92]. This restriction is too strict for queries in an EMS environment, because it forces scientists to start most queries with the root of the schema graph. Fox allows arbitrary path expressions that may traverse all types of relationships in both directions. As an important special case, path expressions in Fox may involve inheritance (is-a) relationships. This allows the user to restrict the classes involved in the path expression to specific subclasses. It also provides both explicit specification of inherited relationships and a mechanism for dealing with multiple inheritence.

- To support Moose collections indexed by other, arbitrary collections, Fox provides an indexing mechanism intuitively derived from array indexing. Any (or all) element(s) of the indexing collection may be used to access elements of the indexed collection.

- Many scientific experiments involve time series or spatial data. Most often, these sets are periodic in all their dimensions, e.g., the time instances in a time series are equally spaced. In most other OO query languages, such sets must be explicitly specified, which may be very cumbersome. Fox provides support for concise descriptions of such sets of arbitrary dimensions.

- Much data generated by scientific computations are initially stored in flat files. Although connecting the EMS directly to the computations is one of our goals, in many instances it is necessary for this data to be explicitly loaded into the system after it has already been generated and written into a file. Although all relational languages offer such a bulk-loading facility, we are aware of no OO language that does. Fox provides such a facility to create new objects based on the contents of files. The new objects may reference both other new objects specified in the file and pre-existing objects in the database.

In this paper, we describe the salient features of Moose and then focus on Fox. The Moose data model is described in Section 2 and Section 3 briefly covers the Moose data definition language. Section 4 details the Fox query language and Section 5 explains the data modification commands of Fox, including the bulk loader. We conclude in Section 6.

2 The Moose Data Model

Moose is an OO data model that supports complex objects, object identity, classes, and (multiple) inheritance. In Moose, real world entities are modeled by objects with unique object identifiers (OIDs) [KC86]. Objects are grouped together by uniquely named *classes*, which capture the objects' common properties. Every class maintains a class extent, stored in the database, to allow subobjects to exist independently of top level objects. For example, in an experimental study of plant growth, objects representing new species of plants may be stored even though they are not yet part of a top level experiment object. Binary *relationships* describe the connections between objects in the schema classes.

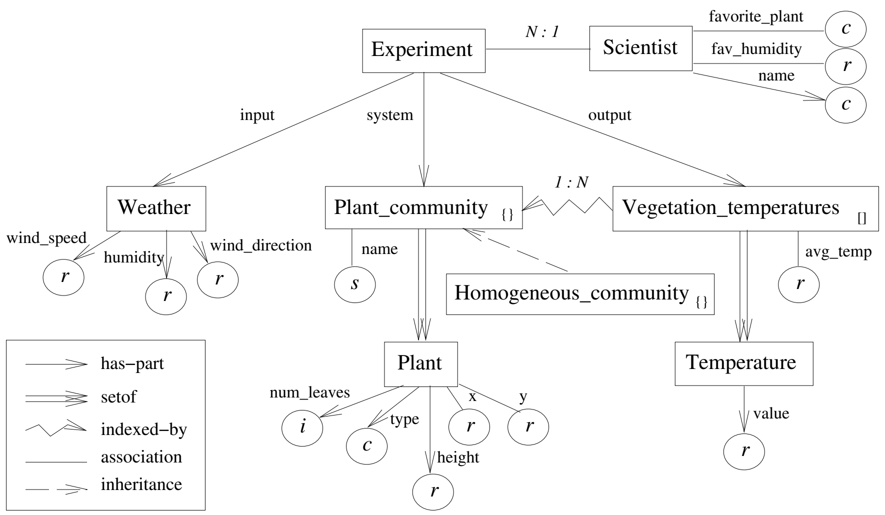

Figure 1: A simple Moose schema showing a soil science experiment to measure plant temperatures under various weather conditions.

Our convention is to represent the schema as a graph: each class is a node (rectangle or oval) and each relationship is an edge in the graph. Each relationship has a label in each direction, which if unspecified, is equal to the name of the target class of the relationship in that direction. For example, the default label of the relationship from Experiment to Scientist is *Scientist*. A simple Moose schema is shown in Figure 1, representing a (simplified) soil science study of how various weather characteristics affect the temperatures of different types of plants. Each experiment is modeled as a complex object, with subobjects representing the experimental system, *Plant_community*, the input parameters, *Weather*, and the output results, *Vegetation_temperatures* for each plant. In addition, each experiment is associated with an object from the *Scientist* class, representing the scientist who conducted the study. More details on the graph representation of Moose will be given along with the descriptions of the corresponding features.

2.1 Object classes in Moose

Each class has a *kind*, which describes the class's basic structure. There are three kinds of classes in Moose: *primitive*, *tuple*, and *collection*. The primitive classes are built into Moose and are currently *Integer*, *Real*, *Boolean*, and *Character String*; in Figure 1, they are represented by ovals, as are their subclasses, and abbreviated *i*, *r*, *b*, and *c*, respectively. (The same primitive class is sometimes represented by multiple ovals for ease of display.)

Tuple and collection classes are created by the user. Objects in tuple classes consist of a prespecified number of other objects, called *parts*, usually from several different classes. Each part is identified by a labeled relationship; labels are unique among relationships from a given class. A special case of tuple classes is *atomic* classes, whose objects have no parts.

Objects in collection classes consist of an arbitrary number of other objects, all from a single class, called the *elements* class. Moose has four specializations of collection classes: *set*, *multiset* (bag), *sequenced-set* (list or array), and *indexed-set*. An indexed-set is a generalization of a sequenced-set. Whereas the members of a sequenced-set are indexed by the set of consecutive integers $\{1, \ldots, n\}$, for some n, the elements of an indexed-set are indexed by (the elements of) an arbitrary collection object. This collection object is called the *keyset* for the indexed-set, because its elements provide indexing *keys* into the indexed-set. (We also say that the collection object's class is the keyset for the indexed-set class.) Each element in the keyset uniquely identifies an element in the indexed-set.[1] In Figure 1, Plant_community is a set class, and serves as the keyset for Vegetation_temperatures, which is an indexed-set class.

2.2 Object relationships in Moose

Like many other semantic and OO data models [HK87, BCG+87, CDV88, Deu90, ISK+93, NO92, SKL88], Moose provides two major categories of relationships between classes: *connection* relationships and *inheritance* relationships.[2] A connection relationship between two classes implies a logical or physical relationship between their object instances. It is very rare that an instantiation of such a relationship connects an object to itself; in general, connection relationships are only meaningful between distinct objects. For example, connection relationships can be used to describe a composite object: the relationships will connect the composite object class (e.g., Experiment) to the classes of its parts (e.g., its input, output, and system). On the other hand, an inheritance relationship between two classes identifies one of them as a *specialization* of the other and implies a semantic correspondance between their object instances. Abstractly, an instantiation of such a relationship always connects an object to itself, representing the fact that the object may play the roles of both the general and the specialized class.

2.2.1 Connection relationships

Many data models only provide one form of connection relationship [Che76, Deu90, FBC+90]. A major contribution of Moose is to provide different kinds of connection relationships, to reflect the different ways that objects may be related and capture additional semantics implied by these ways. Moose also provides several dimensions of flexibility for connection relationships, which are orthogonal to the kind of the relationship. These dimensions are the various properties a connection relationship may have, i.e., mutability constraints, a cardinality ratio, and the potential for null values, and they are detailed in Section 2.2.2. The rest of this section describes the kinds of connection relationships.

Moose has four kinds of connection relationships: *tuple-composition, collection-membership, collection-indexing*, and *association*. In general, each kind of a relationship may be interpreted in two different ways, from the perspective of each of the two classes it connects. In each interpretation, one of the two related classes plays the role of the *source* class and the other plays the *target* class, thus imposing a direction in the interpreted relationship. Table 1 lists each of the four kinds of relationships, the edge types used in

[1] The analogy between sequenced-sets and indexed-sets is even clearer in the query language. An integer is used to retrieve a sequenced-set element in typical array fashion, e.g., my_sset[5]. A keyset element is used in the same way to retrieve an indexed-set element, e.g., my_iset[my_key_elt].

[2] Connection relationships are related to *aggregation* relationships proposed in other semantic and OO data models [SS77, SKL88]. Not all connection relationships, however, represent aggregations of simpler to more complex objects, hence the different term.

Figure 1 to identify each relationship, and a brief description of their semantics. Table 2 shows their directed interpretations and any constraints on the kinds of the corresponding source and target classes.

Relationship Kind	Edge Type	Description
tuple-composition	single directed	Connects a tuple class to one of its part classes
collection-membership	double directed	Connects a collection class to the class of its elements
collection-indexing	zigzag directed	Connects an indexed-set class to its keyset collection class
association	undirected	Connects two classes that are mutually associated

Table 1. Kinds of relationships in Moose.

Relationship Kind	Directed Interpretation	Source Kind	Target Kind
tuple-composition	has-part	tuple	
	is-part-of		tuple
collection-membership	is-set-of	collection	
	is-element-of		collection
collection-indexing	is-indexed-by	indexed-set	collection
	indexes	collection	indexed-set
association	is-associated-with		
	is-associated-with		

Table 2. Constraints on relationships in Moose.

Note that associations have the exact same interpretation in both directions (they are *isotropic*), while the remaining three kinds of relationships have a different interpretation in each direction (they are *non-isotropic*). For example, in Figure 1, Experiment *has-part* Weather, but Weather *is-part-of* Experiment. For the non-isotropic relationships, we believe that one of their directed interpretations is dominant, i.e., has-part, (is-)set-of, and (is-)indexed-by are dominant for tuple-composition, collection-elements, and collection-indexing relationships, respectively. Often, we will draw non-isotropic relationships as directed edges from the source to the target class of their dominant directed interpretations, as we have done in Figure 1.

Non-isotropic relationships are *structural*: they define the structure of their dominant source class. There are constraints on the kinds and number of structural relationships that a class may have, as indicated in Table 1. Specifically, the structure of (objects in) a tuple class is defined by some arbitrary number of has-part relationships, which should be the only structural relationships of the tuple class. The structure of a collection class is defined by a single set-of relationship, with the exception that indexed-set classes are defined by a single set-of and a single indexed-by relationship. No other structural relationships should exist for a collection class. In addition to the above constraints on the relationships of individual classes, there is a global constraint on structural relationships. When only the dominant interpretations of the structural relationships are considered, all the structural relationships should form an acyclic directed subgraph of the Moose schema graph. (For example, objects of a class cannot have objects of the same class as parts).

Association relationships are not structural and therefore are not constrained. Associations connect objects in two arbitrary classes (including collection classes). For example, the average temperature of a set of temperatures may most naturally be associated with the corresponding collection class, as in Figure 1. In other data models, collection objects cannot be associated with other objects. To capture the above examples, one would have to create artificial tuple classes containing both the sets and the associated attributes.

2.2.2 Properties of connection relationships

Each relationship has several properties, which may take different values for each of its two directed interpretations (directions, for short). Like in many other data models, the properties supported in Moose include mutability constraints, a cardinality ratio, and the potential for null values.

Mutability constraints One property of a relationship in a given direction is whether or not an object in the source class must always be related to the same object in the target class, over the lifetime of the database. If changing the target object is always permitted, the relationship is *mutable* in that direction. If changing the target object is never permitted, it is *immutable*. Finally, if changing the target object is only permitted until the scientist explicitly *freezes* it, the relationship is *step-immutable* in that direction. When an object is frozen, immutability propagates recursively to all its step-immutable structural relationships. Mutability was introduced by SDM [HM81] and exists in Jasmine [ISK+93]; Moose is the first model to introduce step-immutability as an option.

Immutable relationships are important because they can prevent the accidental deletion of data. For example, making the relationship from experiment to scientist immutable prevents any user from modifying that information, ensuring that the scientist in charge of an experiment remains accountable for it. Step-immutability is used to allow experiments to be designed in steps. It is desirable to prevent any changes to an experiment's input once it has been conducted, so that it can always be used to interpret the obtained results. However, step-immutability allows the scientist to modify the input until then (and only until then), exploring various options. Similarly, the output of an experiment cannot be determined until the experiment is run, but should not be modified afterwards.

We note that mutability is not automatically propagated in composite objects. For example, although Plant_community may have an immutable set-of relationship to Plant, Plant may have mutable has-part relationships, such as to its height. Removing or replacing a plant may imply changing a community, but growing a plant does not imply creating a new plant.

Cardinality ratios In Moose, connection relationships have cardinality ratios, as in the Entity-Relationship model [Che76]. The ratios are separated into a cardinality for each direction of the relationship. Set-of relationships are multi-valued and indexed-by relationships are single-valued, while all other directions of relationships may be single-valued or multi-valued. The ability to *share* target objects is captured by a multi-valued relationship in the reverse direction; a single-valued relationship in the reverse direction means the relationship to the target object is *exclusive*. For example, the 1-N relationship from Scientist to Experiment in the schema of Figure 1 is a multi-valued, exclusive relationship.

Multi-valued cardinalities are actually a shorthand notation when used for has-part and association relationships. There is an implicit multiset class in the schema, whose instances contain the "many" objects of the relationship. For example, the multi-valued association relationship from Scientist to Experiment of Figure 1 is equivalent to the schema fragment of Figure 2, which places an explicit multiset class between the two.

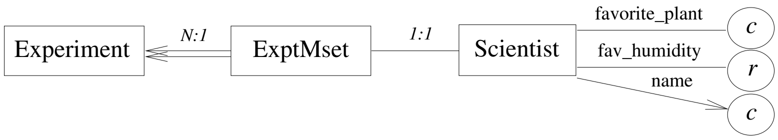

Figure 2: An equivalent schema showing the implicit multiset class.

Null objects The above cardinalities describe a relationship's maximum cardinality in each direction. Its minimum cardinality is also of interest. If a relationship must always have a target object in a direction, then the relationship is *required* in that direction. Otherwise, a *null* target object is allowed. This is a generic object that is potentially a member of all classes. Such an object in Moose corresponds to the "does not exist" null defined in GEM [Zan83], meaning that for a given source object the relationship currently has no target object in that direction. Null target objects may support the creation of subobjects before their parent objects by allowing the subobject to have a null parent. For example, plant objects representing a new species may be created before the plants are used in an experiment. Null target objects also allow complex objects to be created before all their subobjects are specified. For example, an Experiment object may be created with a null target object for output, indicating that no output object currently exists, perhaps because the experiment has not yet been completed. After completing the experiment, the null may be replaced with an object.

Complex object existence dependencies An object A is *existence-dependent* on another object B if it cannot exist without B, i.e., if A is automatically deleted when B is deleted. In Moose, an existence dependency can occur in either direction of any connection relationship. This is in contrast to other models that support such constraints, e.g., Orion [BCG+87], Extra/Excess [CDV88], and Jasmine [ISK+93], where existence dependency semantics are only attached to (specified) structural relationships, and only in the dominant direction. Existence dependencies among objects are not expressed as separate properties in Moose; they are determined by the properties of mutability and null permissions. If a relationship in a given direction is immutable or single-valued and does not allow a null object then the source is existence-dependent on the target.

2.2.3 Derived connection relationships

Independently of the dimensions of structure, mutability, cardinality ratio, and null object permissions, each relationship may optionally be *derived* in one direction. In that case, given an object of the source class, the corresponding target object is specified by a rule (often a query). When a query is not sufficient to express a rule, an arbitrarily complex procedure in a programming language, e.g., C++, may be used. Most aggregate characteristics of collections (e.g., sum, max, min, avg, count) will be derived. For example, the average temperature for a set of temperatures may be a derived relationship from the set of temperatures to a real number. Since scientists are often interested in the statistical properties of some phenomenon, it is extremely useful to be able to implicitly compute aggregates and other analytic results. Derived relationships may also be used in more general ways; for example, the output of the experiment in Figure 1 is dependent on the experiment input and system and is essentially derived by running the experiment. For a simulation experiment, a derivation rule for the output relationship could actually compute the output from the input by invoking the simulator.

2.2.4 Inheritance relationships

Suppose that the schema in Figure 1 is enhanced so that a scientist may be classified as a Staff member, Student, or Project Assistant (PA). The relevant subschema is shown in Figure 3, using dashed directed edges to capture the *inheritance* relationships. An inheritance relationship is a directed relationship between two classes, called the *superclass* and the *subclass*. Moose defines inheritance in terms of *inclusion inheritance* and *specialization inheritance*. Inclusion inheritance means that all objects in the subclass are also instances of the superclass. In Figure 3, the instances of Staff_member, Student, and PA are all instances of Scientist, and will be retrieved when iterating through the instances of Scientist. Several languages [BKK88, BNPS92] provide optional iteration over the entire class hierarchy, with the default being iteration over objects that are not instances of a subclass. We chose the opposite, because we believe that requests on all members of a class, including those of its subclasses, are most common in an experimental environment. Specialization inheritance means that the subclass inherits all of the relationships of the superclass. In Figure 3, the subclass Student inherits the relationships labeled *favorite_plant*, *fav_humidity*, and *name* from Scientist, and additionally has relationships labeled *gpa* and *dept*, which are applicable to Student objects but not

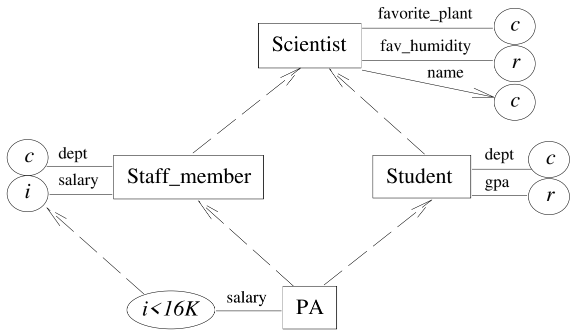

Figure 3: An inheritance hierarchy.

to Scientist objects. Staff_member also has a relationship labeled *dept* which is different from the *dept* relationship of Student, and PA inherits both *dept* relationships.

Relationships inherited from a superclass may be refined by the subclass, i.e., a new target class may be specified, which will be a subclass of the original target class. When this happens, the new relationship overrides the old relationship. In Figure 3, Staff_member has a relationship *salary* which is an integer, and PA refines the inherited *salary* relationship to a subclass of integer, integer<16K. Derived relationships may additionally be refined by specifying a new derivation rule for the subclass, without changing the target class. A subclass may also create a relationship having the same label as a superclass relationship that is not intended to override the latter. In this case, the subclass can access either relationship by using the appropriate Fox path expression.[3] Moose supports multiple inheritance and resolves name conflicts between inherited relationships by making both relationships available via their inheritance path. Inheritance is important for scientific experiments because it allows the scientist to gradually refine a highly complex experimental schema over the course of a study. This flexibility permits the scientist to retain a core schema, although the specific details for each study may vary.

2.2.5 Derived subclasses

Moose supports *derived* subclasses, which are defined using *constraint* inheritance [ABD+89]. The class extent of a subclass is computed intensionally from the extent of the superclass using a rule. The extents of derived subclasses are automatically maintained by the system. Derived subclasses are particularly useful when the scientist discovers that certain experiments with interesting characteristics have special behaviors. For example, in the experiments represented by Figure 1, it may turn out that plant communities whose plants all have the same type exhibit certain temperature patterns. The user could then create a derived subclass of Plant_community, Homogeneous_community (also shown in Figure 1), to keep closer track of those communities and the experiments that use them. Encore [SZ89] supports derived subclasses whose rule is a predicate.

3 The Data Definition Language of Moose

A Moose schema is composed of classes and relationships between classes. The data definition language provides statements to create and destroy both classes and relationships, and to relabel existing relationships between classes. Each class is created in a separate statement. Relationships may be created independently or as part of a **class create** statement, for ease when defining a large schema at once. Independent creation of relationships allows a user, for example, to create the schema of a complex object top-down, specifying the most important classes before worrying about (or creating) the classes for related subparts. It also permits a user to modify the schema easily, or to build on an existing schema. Iris also allows both forms of relationship

[3] We expect many scientific schemas to contain hundreds of classes. It is quite easy to forget what names have already been used for relationships.

creation [Bee88]. Each **class create** statement must specify the kind of the class. Each **relationship create** statement may specify the properties of the relationship in each direction. If left unspecified, the properites obtain default values of association, N-1, mutable, and with null objects permitted. The following statements create a portion of the schema in Figures 1 and 3.

create tuple class Experiment;

create tuple class Weather **with**
  haspart wind_speed **to** Real
  haspart humidity **to** Real,
  haspart wind_direction **to** Real;

create tuple class Plant;

create set class Plant_community **with**
  assoc name **to** String,
  immutable required setof to Plant;

create immutable haspart input **from** Experiment **to** Weather;

create immutable required haspart plants **from** Experiment **to** Plant_community;

Note that the relationship from Plant_community to Plant uses the default label *Plant*. The keyword **required**, borrowed from Iris [Bee88], indicates that null values are not allowed. The next statement changes the label from Experiment to Plant_community to be *system* (to match Figure 1).

relabel plants **from** Experiment **to** Plant_community **with** system;

Inheritance is specified as part of the **class create** statement, before listing the new class's relationships. The following statements create the class Staff_member as a subclass of Scientist.

create tuple class Scientist;

create tuple class Staff_member **inherits from** Scientist **with**
  assoc dept **to** String,
  assoc salary **to** Integer;

Relationships may be refined by a subclass, so that the subclass's relationship overrides that of the superclass. The next example creates PA as a subclass of Staff_member and refines the (integer) *salary* relationship inherited from Staff_member to a subclass of integer, i<16K.

create class i<16K **inherits from** integer;

create tuple class PA **inherits from** Staff_member **with**
  assoc salary **to** i<16K **refines** PA@Staff_member.salary;[4]

The **class destroy** statement removes the class from the schema. It also removes all relationships connected to the class, since they can only exist to connect two classes. Relationships may also be removed separately, using the **relationship destroy** statement. If there is more than one relationship between two classes, the relationship label is used to indicate which relationship to remove. The first statement below removes Plant_community from the schema, including its relationships to Plant and Experiment. The second statement removes the relationship from Weather to Real labeled *wind_direction*.

destroy class Plant_community;

destroy wind_direction **from** Weather **to** Real;

[4] PA@Staff_member.salary is a path expression describing which inherited relationship is being refined. Path expressions are described in Section 4.1.2.

The entire process of defining a schema may take a long time, especially in scientific experiments that involve hundreds of input parameters and generate hundreds of output results. In addition, the schema may be very complex and a user may iterate over the schema several times, making adjustments. While a user is creating or adjusting the schema, the schema may be incorrect, e.g., a collection class might be missing a set-of relationship. In order to allow intermediate incorrect schemas, the statements defining a Moose schema must be placed between a **begin schema** and an **end schema** statement. The schema is checked for correctness after the **end schema** statement.

Often, scientists run several experiments before they realize what the critical parameters are, and sometimes want to change a parameter that was initially constant and left out of the "input" specification. Schema evolution is the process of changing the schema. While supporting schema evolution on an unpopulated schema is fairly straightforward (we allow any changes), schema evolution after the schema has been populated with objects raises some questions. For example, when a new relationship is added to an existing source object, what target object should be associated with it? We currently plan to adopt the schema evolution semantics outlined for Orion [BCG+87] and possibly adapt it to an specialized needs that may arise in scientific applications.

4 Fox: the query language for Moose

Fox (Finding Objects of eXperiments) is the declarative query and data modification language for Moose. In addition to the novel features mentioned in the introduction, Fox has the following useful characteristics. It allows arbitrary object construction; it provides closure [ASL89], so that query results can be used in later queries; it provides optional persistence of query results, so that query results can be stored when needed later, but need not clutter the database; and it supports queries based on both named objects and class extents.

There are several other declarative query languages that have been proposed for OO data models, namely O2query [BCD89], Extra/Excess [CDV88], CQL++ [DGJ92], COQL [NO92], OQL [ASL89], IQL [AK89], XSQL [KKS92], Ontos's OSQL [Ont92], and the Orion query language [Kim89], as well as the Equal query algebra [SZ89]. Fox differs from them in several aspects. The two most prominent ones are its ability to use complex path expressions to navigate any path in the schema graph in any direction, even through inheritance relationships, and its ability to associatively retrieve individual elements of an indexed-set given the corresponding elements of its keyset. The power of Fox in these two aspects is discussed in more detail in the appropriate subsections below. We compare Fox to other languages directly in Section 4.3, primarily to O2query.

4.1 Structure of a query

The basic structure of a Fox query is derived directly from SQL. A Fox query has the form

> **for** <range-binding-list>
> **select** <projection-list>
> **where** <qualification>
> **as** <name>;

The following subsections explain the various clauses. First, we describe object naming, because names for objects play central roles in the other clauses.

4.1.1 Object naming

Any Moose object may be *named*. A name acts as a surrogate object identifier for the object, one that is user-generated and likely to be mnemonic (in contrast to system OIDs, which are system-generated and would have no particular meaning to a user). Named objects serve two important purposes: they allow users to access and reuse query results easily, and they provide users with a quick handle to special or frequently used objects.

Each (sub)query has an optional *naming* clause, "**as** <name>", which attaches <name> to the query result object. These names must be unique throughout the database, and may subsequently be used to identify the object. Class extents are essentially special cases of named objects: the name of the class is automatically attached to the class extent when the class is created.

The set of all persistent names of objects is maintained as a special ("set of string") object, named *names*. (It belongs to a set of string class defined by the system for this purpose.) It is therefore easy to see which names are currently attached to objects, using a query on *names*, and to free names for reuse, using a **delete** statement to remove the desired name from *names* (without affecting the named object).

Both Iris [FBC+90] and CQL++ [DGJ92] provide names called instance variables and object variables, respectively. However, their names cannot be persistently stored in the database, which means they cannot be used to identify frequently used objects. Although O2query uses named objects, it does not allow names to be attached to query results [BCD89]. The only way to persistently attach a name to an object is to embed the query in a programming language, and assign the query result to a persistent program variable. Fox allows the names to be created in the query language, as well as to be accessible from it. COQL [NO92] provides a similar feature.

4.1.2 Object variables and path expressions

In the **for** clause of Fox, *object variables* may be bound to the members of class extents or collection objects described by path expressions. At any point, the object variable represents a single member of the collection and can be used as if it were a name for the object. More than one object variable may be bound in the **for** clause, and the object variables may be dependent on each other. The scope of the object variable bindings includes all subqueries that do not rebind the variable, as well the **select** and **where** clauses. Fox's object variables are similar to tuple variables in SQL [DW89], iteration variables in Orion [BKK88], reference variables in Daplex [Shi81], and range variables in CQL++ [DGJ92]. The names of all named objects used in the query must also appear in the **for** clause. The following is an example **for** clause that binds three object variables and indicates the use of a named (Scientist) object Jo.

for e **in** Experiment, p **in** e.system.Plant, s **in** Scientist, Jo

Path expressions are the primary mechanism of specifying what information to retrieve. A path expression corresponds to a path in the schema graph. A path expression starts with the name of an object or an object variable, called the *path expression root*, and continues by traversing relationships. Fox path expressions are more expressive than those of other languages because they can traverse both connection and inheritance relationships in both directions. OQL "contexts" [ASL89] are similar to Fox path expressions and can indicate paths through any relationships. However, OQL contexts are only used to restrict the results (which are sub-databases) and do not directly evaluate to objects. In some OO models, such as ObjectStore [LLOW91], Versant [Ver91], and Objectivity [Obj91], it is possible to define an inverse relationship, which can then be used to traverse the relationship "in reverse." However, these inverse relationships must be explicitly declared.

The syntax of a path expression begins with the path expression root object, which is specified either by an object variable or by a named object. Then, for each relationship traversed, the path expression contains a symbol corresponding to the type of relationship and the relationship label. Inheritance relationships use the symbol '@' (pronounced "as"). Connection relationships use the symbol '.' (pronounced "dot"). Some sample path expressions are: *s.Experiment.input.humidity*, *e.system@Homogeneous_community.Plant*, and *p.Plant_community.Experiment.input*. The first path expression indicates the humidity of the input of some experiment run by the scientists. The second expression traverses an inheritance indicates all plant objects that are members of a homogeneous community that is the system for some experiment *e*. The third expression, which traverses some relationships in their non-dominant direction, indicates the input of all experiments run using a plant community containing the plant *p*.

The result of evaluating a path expression is all objects reachable from the path expression root. If all relationships traversed are single-valued, the result is a single object. If any relationship is multi-valued (e.g., a set-of relationship), then the result is potentially many objects and these objects must become the elements of a collection, specified implicitly or explicitly by the query. If the path expression traverses multiple set-of relationships, then the effect is that of unioning the innermost collection elements [BNPS92, KKS92, HM81]. If two path expressions begin with the same prefix, then they are implicitly joined by their common prefix when they are evaluated.

A complete path expression to an inherited relationship (such as the *name* relationship of the Student class) always traverses the inheritance relationships involved. This solves any name conflict problems, including those caused by multiple inheritance. The PA class in Figure 3 has two relationships labeled dept, but their path expressions are different: *PA@Student.dept* and *PA@Staff_member.dept*. A path through an inheritance relationship toward a subclass restricts the class of the result object, by requiring that it be an instance of the subclass.

Path compression techniques may be used to permit the specification of incomplete paths. That is, *PA.salary* may be expanded by the system to *PA@Staff_member.salary*. We are investigating ways of efficiently determining the correct complete path, since there may be more than one possible completion [Las93].

4.1.3 Object retrieval (select clause)

Fox supports query results that include arbitrary projections on objects and arbitrary joins between objects, which frequently lead to new objects being created by the query. While projections and joins are a fundamental part of relational DBMSs, many OO systems have chosen to allow only the retrieval of existing objects from their declarative query languages, not the creation of new ones. (In order to create new objects in these systems, the programming language must be invoked.) For example, Orion's query language only returns the OIDs of existing objects [BKK88], Ontos's OSQL returns relations [Ont92], and O2query results are values (not objects) [BCD89].

Fox allows the user to explicitly specify the structure of the result of a query. The **select** clause specifies not only what to retrieve, but also exactly what the resulting objects should look like. For each binding specified in the **for** clause, an object is produced. The object has a connection relationship to the objects produced by evaluating each component (path expression or subquery) in the **select** clause, and the object's kind is given by one of the keywords **tuple**, **set**, **multiset**, **sequencedset**, and **indexedset**. If the new object would have only one relationship, the user may instead directly project that component by omitting the kind keyword. The final result is a single object, which is a multiset containing all the objects produced.

When a query creates new objects of a given structure, it also creates a corresponding class, outside the persistent class hierarchy, to which the new objects belong. The query below returns a multiset of new tuple objects, each having a humidity and fav_humidity. The query also implicitly creates the new multiset and tuple classes.

> **for** e **in** experiment
> **select tuple** e.input.humidity, e.scientist.fav_humidity;

The type of a connection relationship optionally may be given, preceding the path expression or subquery. The keywords **haspart**, **assoc** or **association**, **setof** and **indexedby** are used. A connection relationship is exclusive, immutable, and allows null values in both directions by default. The cardinality ratio property is determined by evaluating the component. In the example query below, each object in the result is a set object which is associated with the name of the plant community and whose elements are plant types.

> **for** p **in** Plant_community
> **select set** p.name, **setof** p.plant.type;

A new label may also be given to some or all of the new object's relationships, which otherwise acquire the label of the last relationship in the path expression. The following example relabels two relationships that were originally both labeled *name*.

> **for** e **in** Experiment
> **select tuple** sname = e.scientist.name,
> comm_name = e.system.name;

To choose just one object from the multiset that represents the result, one of the keywords **the** and **any** should follow **select**. The **the** keyword expects the multiset to contain only one element, and returns an error otherwise. The **any** keyword chooses an element at random. We expect **the** to be used most often when the **for** clause indicates iteration, and hence a multiset result, but the **where** clause qualification restricts the multiset to a single element. Similar constructs include **the** in Daplex [Shi81] (from which we adopted the keyword), the **element** function in O2query [BCD89], and the **member** message in COQL [NO92]. Fox introduces **any**. We expect **any** to be used frequently when more than one object may satisfy the **where** clause qualification, but any such object is acceptable. For example, a user creating a new experiment may want to use any existing system whose plants are all above a certain height. In the following example, all the plants in a homogeneous community have the same type, and so it is only necessary to retrieve the type of one plant.

> **for** h **in** Homogenous_community, p **in** h.plant
> **select any** p.type;

If a multiset is produced by a query, the keyword **unique** following **select** specifies that the multiset should be converted to a set. Uniqueness is conceptually evaluated before the new objects are created, based on the components for the new objects. Uniqueness of components is determined by object identity. The following query retrieves only unique temperatures that appear in an experiment's output.

> **for** e **in** Experiment, t **in** e.output.temperature
> **select unique** t.value;

Individual elements may be retrieved from a sequenced-set or indexed-set according to their position or indexing element. Sequenced-set element retrieval looks just like list element retrieval in O2query [BCD89], and like array element retrieval in many programming languages, e.g., C and Pascal. If Plant_community in Figure 1 were a sequenced-set, the following query would retrieve the first plant in each experiment's system.

> **for** e **in** Experiment
> **select** e.system[1];

The query syntax to retrieve indexed-set elements is a natural extension: instead of identifying the position of the indexed-set elements, its indexing element is specified. However, since Moose is unique in providing an associative connection from a set to an indexed-set, Fox is unique in providing associate retrieval. The next query retrieves the height of each plant in an experiment, and the temperature value that was associated with it in the experiment.

> **for** e **in** Experiment, p **in** e.system.Plant
> **select tuple** p.height, e.output[p].value

If the indexing element were a single, named object, its name could be used in place of the object variable p inside the brackets.

4.1.4 Selection of result objects (where clause)

The qualification in the **where** clause is a single boolean expression, which is evaluated for each potential object in the result. Fox supports object comparisons by identity, shallow equality, and deep equality, using the operators **is**, =, and ==, respectively. We extend earlier definitions of shallow and deep equality [BNPS92, SZ89] to differentiate a complex object from its associations: only the has-part, set-of, and keyset relationships of the objects are compared, directly for shallow equality and recursively for deep equality. Fox also provides comparison operators for comparing scalars to collections and collections to collections by membership. All membership tests are by object identity. In addition to the operators provided by other languages [KKS92, BNPS92], Fox adds the **disjoint** and **overlaps** operators for comparing two collections (although they may be simulated using set intersection and an (in)equality comparison with the empty set). Collection objects may also be combined with set-theoretic union, intersection, and difference, and additive union [VD91]. Note that shallow equality for collection objects is the same as a membership test for equality.

As an example, the following query finds the input parameters of all experiments using a system that Jo used.

```
for e in Experiment
select tuple e.input.wind_speed, e.input.humidity,
        e.input.wind_direction
where e.system in (for s in Scientist
                        select s.Experiment.system
                        where s.name = "Jo");
```

4.1.5 Constants and periodic data

Constants may be used in queries wherever an object is needed, e.g., in the **for** clause or for comparisons. When a collection is expected, a constant may be specified as a series of element values inside {}'s. In addition, Fox provides a special construct for creating set constants for periodic data, such as a time series. Consider the set {5, 10, 15, 20, 25}. Explicitly typing out the elements of the set may be very tedious, especially when the set contains tens or hundreds of elements. Fox supports the construct "(**from** <start> **to** <finish> **step** <interval>)", which can be used to represent the set {<start>, <start> + <interval>, ..., <finish>}. For example, the above set may then be specified as (**from** 5 **to** 25 **step** 5). We expect this construct to be used often to intensionally specify the members of periodic sets, in combination with a derived **setof** relationship. Similar functionality could be obtained using a looping construct provided by a programming language, but that would require mixing the programming language and query language statements. By keeping all the constructs inside the declarative query language, Fox permits optimization of the whole query by the query optimizer. It also prevents crossing the boundary between the query language and the programming language at run time, which is generally quite time-consuming. The following query example generates a time series that includes all weekdays in the eighth through twelfth weeks of the semester.

```
for w in (from 8 to 12 step 1), d in (from 2 to 6 step 1)
select tuple week = w, day = d
as my_time_series;
```

4.1.6 Ordering, grouping, and aggregate functions

Any set result may be ordered, making it a sequenced-set. Ordering is specified by an optional **order by** clause as in SQL. The query below lists, for each experiment, the scientist's name, their favorite humidity, and the humidity they used in the experiment, ordered alphabetically by the name of the scientist.

```
for e in Experiment
select tuple e.scientist.name, e.input.humidity,
            e.scientist.fav_humidity
order by asc e.scientist.name;
```

There is no **group by** clause in Fox. The effects of grouping can be accomplished with subqueries that include a join on the grouping attributes [DW89]. Aggregate functions, which are expressed in an unintuitive manner in SQL, operate on a subquery in Fox. This more intuitive syntax was first proposed for SQL/NF [RKB87]. Group qualifications are expressed in the **where** clause for the query, instead of in a **having** clause. If the subquery is needed twice, e.g., for two aggregate functions or for the **where** clause, the result of the subquery may be named and the name used instead of repeating the subquery. The following query will print the name of each experiment's system and the average height of all the plants in that system, when the average height exceeds 60 inches.

```
for e in Experiment
select tuple e.system.name, avg(for p in e.system.plant
                                select p.height
                                as heights)
where avg(heights) > 60.0;
```

4.2 Persistence of results

Moose supports three levels of persistence for query results: *transient*, *temporary*, and *persistent*. These levels apply to both the result class and the result objects. Transient results exist only for the duration of a query. All query results are transient by default. If a query result is named by the query, it becomes temporary, along with all the the objects created by that query and the name itself. Temporary results persist until the user exits the current session with the database. Transient and temporary classes are not part of the persistent class hierarchy. To make query results persistent, the user must use the **insert** statement to explicitly store the objects in an existing class in the database. Names attached to persistent objects are also persistent.

Temporary results provide additional flexibility by allowing the user to examine and perhaps refine a result before deciding whether to make it persistent. If all results were persistent, the database would quickly become cluttered with objects the user does not want to see again. For example, scientists often generate tens or hundreds of graphs plotting various points, when they only want to choose one or two for a paper or talk. Making the graphs temporary would allow the scientist to make a final selection before entering the most useful ones into the database for future use. CQL++ also supports temporary results [DGJ92], although in CQL++ the result objects need to be explicitly inserted into a special, temporary class, while Fox infers the class definitions for temporary objects from the query.

4.3 Comparison with other query languages

The major differences between Fox and other query languages lie in the expressiveness of Fox's path expressions and the ability to have associative indexing. In this section, we give examples of Fox queries with their corresponding (more complicated) equivalents in O2query [BCD89]. We also show queries that cannot be expressed in O2query, or any other declarative query language besides Fox. We chose O2query because it is one of the most expressive declarative query languages, and also the language most similar to Fox.

Traversing relationships in non-dominant directions Consider the following Fox query on a plant object named *tall_corn*.

> **for** tall_corn
> **select** tall_corn.Plant_community.Experiment;

This query retrieves all experiments that used the plant *tall_corn*. It is a simple query because the **select** path expression can navigate directly from *tall_corn* to the experiments, along the non-dominant direction of the relationships between Experiment and Plant_community and between Plant_community and Plant. The equivalent O2query query follows. (Since O2query syntax is similar to Fox syntax, we show the O2query syntax.)

> **select** e
> **from** e **in** Experiment, p **in** e.system.Plant
> **where** p **is** tall_corn;

In the above query, it is necessary to first indicate looping through all experiment objects, and then through all the plant objects of each experiment, to see if any of them are the object *tall_corn*. The O2query query requires much more effort to write, since it is necessary to introduce variables for all the collections in the path to the corn object, instead of only one, and to introduce a **where** clause that is unnecessary in Fox. In the general case, the O2query query would require an extra object variable and an extra conjunct in the **where** clause for every collection in the path expression, greatly increasing the complexity of formulating it.

Traversing inheritance relationships Path expressions through inheritance relationships can be used in Fox to restrict which objects belong to the query result. The following query retrieves plant objects that are members of a homogeneous community used as the system for some experiment. The inheritance relationship restricts the Plant_community objects to only Homogeneous_community objects, and thus additionally restricts the Plant objects to only members of a Homogeneous_community.

```
for e in Experiment
select e.system@Homogeneous_community.Plant;
```

To get the same result without traversing an inheritance relationship, a join between the Plant_community objects and Homogeneous_community objects is required. This also necessitates an object variable ranging over the Homogeneous_community objects, which is not necessary in the above Fox query. The following is the equivalent query in O2query[5].

```
select e.system.Plant
from e in Experiment, h in Homogeneous_community
where e.system is h;
```

Path expresssions may also traverse inheritance relationships in the direction toward the superclass, to resolve potential name conflicts that arise from multiple inheritance. That is, although two superclasses of the same class may each have a relationship labeled L, both L relationships are accessible. The desired L is specified by including the inheritance relationship to the correct superclass in the path expression. In O2query and all other languages we know, it is not possible to traverse the inheritance relationship, and so one of the L relationships is lost.

Indexed element retrieval One of the most novel constructs in Moose is indexed-sets. Accordingly, the query facilities that deal with indexed-sets are the most novel part of Fox, and the most difficult (if not impossible) to mimic in any other query language.

In O2, it would be possible to model the indexed-set Vegetation_temperatures as a list. If Plant_community were also a list, rather than a set as in Figure 1, then a correlation could be maintained between the elements of the two lists. (This is a variation of parallel arrays.) However, the user would need to maintain the correlation explicitly, and we know of no way to associatively retrieve the elements of Vegetation_temperatures. Given a particular Plant_community element, there is no way in O2query to find out its position, and hence no way to retrieve the corresponding Vegetation_temperatures element. Also, although it would be possible to retrieve the corresponding elements for a given position (by specifying the position), there is no way to iterate through all the possible positions. O2query's iteration mechanism treats lists as sets, and ignores position.

It would also be possible in O2 to model the indexed-set Vegetation_temperatures as a tuple with three parts: one each for the plant, the temperature, and the experiment. However, this would result in a lot of redundancy inside the database. For each plant in a given experiment, the experiment would be repeatedly stored. For each experiment that used the same plant community, the plants would be repeatedly stored. Also, the additional tuple classes would add complexity to the schema, when the single indexing relationship suffices in Moose.

Deep equality Fox's deep equality is similar to that of other languages, in that it recursively traverses the relationships of two objects to compare them. The objects are deep equal if and only if the same object is found along each relationship path. However, the deep equality of other languages traverses all relationships. Because Moose distinguishes structural and non-structural relationships, Fox's deep equality is able to traverse only structural relationships, resulting in more meaningful comparisons. For example, when comparing two Experiment objects of the schema in Figure 1, the associated Scientist object should not affect the result. The Experiments are deep equal if they used the same input parameters, plants with the same characteristics, and produced the same collection of temperature values. It does not matter whether the same scientist ran both experiments or not.

Queries in other languages would need to explicitly compare the input, output, and system of the Experiment object. Although a function could be written specifically for Experiment, a different function would be necessary for each class, since the other models do not distinguish structural relationships. (Even if a model did distinguish structural relationships, we know of no way to test whether a relationship is structural or not, inside a user-defined function.)

[5] XSQL [KKS92] provides a simple syntax for expressing the join in the path expression, although it still requires the extra object variable.

5 Data Modification

Fox's data modification statements are closely integrated with its query language. There are four such statements: **insert**, **update**, **delete**, and **load**.

5.1 Insert

The **insert** statement adds new objects to the database. There are two forms of **insert**: in the first one, the objects related to the new object via the appropriate relationship are explicitly specified, whereas in the second one they are obtained by a query. For explicit specification, **insert** uses the keyword **instance**, followed by the list of related objects in parentheses. Like Iris [FBC+90], we have extended **insert** to allow multiple object insertions in one statement and we allow the objects to be named when they are created. Each instance is followed by an optional naming clause. The following query creates three new corn plants, and attaches the name "tall_corn" to the second one.

```
insert into plant(num_leaves, type, height, x, y)
    instance(3, "corn", 2.3, 0.2, 1.2),
    instance(8, "corn", 5.0, 0.8, 1.4) as tall_corn,
    instance(4, "corn", 3.7, 1.2, 2.2);
```

Insert can also use an arbitrary query to create objects. Although each **insert** statement inserts objects into exactly one class, we have extended **insert** to allow nested insertions. This extension permits insertions into multiple classes in the same statement. For example, the entire experiment input may be specified in one statement. The result of an **insert** statement is a multiset containing the inserted objects. The multiset may optionally be named to facilitate using the collection of new objects in later queries. This may be less advantageous compared to inserting instances explicitly, where each new object may be individually named in the **insert** statement.

A nested **insert** statement is needed to make a temporary object into a persistent one when the object is related to other temporary objects, since they must also be inserted. Nested **insert** statements also have the advantage of allowing the user to mostly copy a complex object, but make a few changes. For example, when creating a new simulation experiment the scientist often wants to use almost the same input as the last experiment, with a few modifications. The following example creates new experiments based on those that used a system named "Nebraska," but increases the humidity of each experiment by 5%. Each new experiment will have a null output object until it has been run.

```
for s in Scientist
select the s
where s.name = "Sally"
as Sally;

insert into Experiment(Scientist, system, input)
for e in Experiment
select tuple Sally, e.system,
          (insert into Weather(wind_speed, humidity, wind_direction)
          select tuple e.input.wind_speed,
                    e.input.humidity * 1.05, e.input.wind_direction)
where e.system.name = "Nebraska";
```

Note that the query to name the Scientist object Sally could have been nested in the insert statement instead.

5.2 Update

Fox's **update** statement extends that of other languages by allowing multiple values of the same object to be updated in the same statement. However, only objects of the same class may be updated in one statement.

A name may optionally be attached to the multiset containing all updated objects, to keep track of which objects were updated, for instance. This example grows all the corn plants by both 1 leaf and 3 cm in height, which would require two **update** statements in other languages [DW89, DGJ92]. (We have not seen declarative update statements for most systems, e.g., O2, Extra/Excess, Iris, and Orion.)

for p **in** plant
update p.num_leaves **to** p.num_leaves + 1,
p.height **to** p.height + 3.0
where p.type = "corn";

5.3 Delete

The **delete** statement removes objects from a given class from the database. The objects may be specified either by name, if they are named, or by a query. Related objects are deleted only if there is an existance dependency from the deleted object to them, i.e., if the relationship to the deleted object is immutable or if it is single-valued and does not allow a null object. The first example below deletes the (Scientist) object named Sally, and the second example deletes the output of Ted's experiments, which were corrupted.

delete Sally;

for e **in** Experiment
delete e.output
where e.Scientist.name = "Ted";

5.4 Load

The **load** command is for bulk insertion of data from a file, generally into multiple classes at once. Many scientific experiments have input and output parameters that number in the hundreds and thousands, and must be loaded into the EMS for each experiment. There needs to be an easier and less time-consuming way to store the new data in the EMS than using multiple **insert** statements (or "new" statements in a programming language), the only method available in many other OO systems. The **load** command provides an easy way to load the new data: the scientist generates one text file containing all the data in the load format and calls **load**. The system can then optimize the loading process for the specified new objects, while multiple **insert** or **new** statements would be executed in the (probably non-optimal) order they were specified. Additionally, the ability to load data that already exists in flat files or notebooks is critical to providing an effective new environment for scientists with ongoing experiments. By writing small programs to transform the old flat file data into the load file format, a scientist can easily load old data into a Moose database.

Load facilities exist in most relational DBMS, but in almost no OODBMS. In a relational DBMS, there is no cross-referencing between tuples and therefore data is easily represented in a file. In OO data models, the use of OIDs makes the representation of bulk data harder. We address the problem in Moose by allowing the user to generate surrogate OIDs for the objects. One of the relationships of the objects may be used as a surrogate OID if it uniquely identifies objects [PG88]; otherwise an arbitrary identifier may be used, e.g., meaningful strings or integers automatically generated by a counter in the program generating the data. Surrogate OIDs may be assigned to new objects created in the file, and also to already existing objects in the database by using queries (either before calling **load** or inside the file) to name them.

The following example is a sample data file for part of the schema of Figure 1. Within the text file, objects are grouped together by class, although a given class may appear more than once in the file and the order of the classes is not important. Each class is described by its name, labels for the relationships for which target objects will be given, and a surrogate key type. If a relationship of the class is not specified, then new objects get a null target object for that relationship. Next, the objects in the class are listed. Each object begins with a surrogate key (possibly one of its target objects, as in Plant_community below), continues with a comma-separated list of its (other) target objects, and ends with a semicolon. The target objects for a **setof** relationship are listed inside curly brackets. Strings are surrounded by quotes, have no maximum length, and may contain any characters.

Weather<wind_speed, humidity, wind_direction> **key** string
{
"hot&dry": 5.2, 0.08, 35.0;
"damp": 15.5, 0.73, 186.0;
"sticky": 12.0, 0.89, 320.0
}
Plant_community<name, Plant> **key** name
{
"tall", {"corn2", "corn6", "rye1"};
"corn", {"corn5", "corn2", "corn6", "corn7"};
}
Plant<num_leaves, type, height, x, y> **key** string
{
"corn2": 24, "corn", 62.2, 14.2, 3.3;
"corn5": 10, "corn", 33.5, 24.3, 22.8;
"corn6": 38, "corn", 64.8, 16.6, 4.7;
"corn7": 14, "corn", 42.0, 2.6, 18.7;
"rye1": 50, "rye", 70.2, 34.5, 5.6;
"wheat8": 9, "wheat", 20.6, 89.3, 17.2;
}
Experiment<input, system> **key** integer
{
1: "hot&dry", "corn";
2: "damp", "corn";
3: "sticky", "corn";
}

There are several features worth noting in the above example. The surrogate keys for Plant_community objects are the (string) target objects of its *name* relationship. They are listed only once, as the first target relationship, and there is no surrogate key field in the object description. For Experiment objects, the target objects for the *input* relationship are the surrogate identifiers of the Weather objects, and the new Experiment objects will have a null Scientist target object. Also, there are new Plant and Plant_community objects which are not used in any Experiment.

6 Conclusions

In this paper, we have presented a new declarative query language, Fox, for the Moose data model, which has been designed specifically for the needs of an Experiment Management System. We have completed a preliminary implemention of Moose and Fox on top of the Exodus storage manager [CDG+90]. The implementation is limited in that it only processes the **select** clause. However, it is the foundation for our current efforts to add more processing capabilities. We are also designing a query algebra to correspond to the expressive power of Fox, and investigating various techniques for efficient query processing.

7 Acknowledgements

We would like to thank Miron Livny and the rest of the Zoo project members, Eben Haber, Renée Miller, and Odysseas Tsatalos, for generating a stimulating environment in which many of the ideas presented in this paper were conceived. We would also like to thank Sophie Cluet, Mark McAuliffe, Scott Vandenberg, and the anonymous reviewers from Brown University whose comments improved many aspects of this paper.

References

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, pages 223–240, Kyoto, Japan, December 1989.

[AK89] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 159–173, Portland, OR, June 1989.

[ASL89] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: A Query Language for Manipulating Object-0riented Databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 433–442, Amsterdam, The Netherlands, August 1989.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for the O2 Object-Oriented Database System. In R. Hull, R. Morrison, and D. Stemple, editors, *Proceedings of the International Workshop on Database Programming Languages*, pages 122–138. Morgan-Kaufman, Inc., San Mateo, CA, 1989.

[BCG+87] J. Banerjee, H. T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H. J. Kim. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3–26, January 1987.

[Bee88] D. Beech. A foundation for evolution from relational to object databases. In *Proceedings of the International Conference on Extending Database Technology*, pages 251–170, Venice, Italy, March 1988.

[BKK88] J. Banerjee, W. Kim, and K. C. Kim. Queries in Object-Oriented Databases. In *IEEE Conference on Data Engineering*, pages 31–38, Los Angeles, CA, February 1988.

[BNPS92] E. Bertino, M. Negri, G. Pelagatti, and L. Sbattella. Object-Oriented Query Languages: The Notion and the Issues. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):223–237, June 1992.

[CDG+90] M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In Stanley B. Zdonik and David Maier, editors, *Readings in Object-Oriented Database Systems*, pages 474–499. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

[CDV88] M. Carey, D. DeWitt, and S. Vandenberg. A Data Model and Query Language for Exodus. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 413–423, Chicago, IL, June 1988.

[Che76] P. P. Chen. The Entity-Relationship Model — Towards a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[Deu90] O. Deux. The Story of O2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[DGJ92] S. Dar, N. H. Gehani, and H. V. Jagadish. CQL++: A SQL for a C++ Based Object-Oriented DBMS. In *Proceedings of the International Conference on Extending Database Technology*, pages 201–216, Vienna, Austria, April 1992.

[DW89] C. J. Date and C. J. White. *A Guide to DB2.* Addison Wesley, Reading, MA, 3rd edition, 1989.

[FBC+90] D.H. Fishman, D. Beech, H.P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T.A. Ryan, and M. C. Shan. Iris: An Object-Oriented Database Management System. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 216–226. Morgan-Kaufman, Inc., San Mateo, CA, 1990.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1987.

[HM81] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[IL89] Y. Ioannidis and M. Livny. Moose: Modeling Objects in a Simulation Environment. In *Proc. IFIP 1989, 11th World Computer Congress*, pages 821–826, San Francisco, CA, August 1989.

[IL92] Y. Ioannidis and M. Livny. Conceptual Schemas: Multi-Faceted Tools for Desktop Scientific Experiment Management. *International Journal of Intelligent and Cooperative Information Systems*, 1(3):451–474, December 1992.

[ILH+93] Y. Ioannidis, M. Livny, E. Haber, R. Miller, O. Tsatalos, and J. Wiener. Desktop Experiment Management. *IEEE Data Engineering Bulletin*, 16(1):19–23, March 1993.

[ISK+93] H. Ishikawa, F. Suzuki, F. Kozakura, A. Makinouchi, M. Miyagishima, Y. Izumida, M. Aoshima, and Y. Yamane. The Model, Language, and Implementation of an Object-Oriented Multimedia Knowledge Base Management System. *ACM Transactions on Database Systems*, 18(1):1–50, March 1993.

[KC86] S. Khoshafian and G. Copeland. Object Identity. In *Proceedings the International Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 406–416, 1986.

[Kim89] W. Kim. A Model of Queries for Object-Oriented Databases. In *Proceedings of the International Conference on Very Large Data Bases*, pages 423–432, Amsterdam, The Netherlands, 1989.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In M. Stonebreaker, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 393–402, San Diego, CA, June 1992.

[Las93] Y. Lashkari. Domain Independent Disambiguation of Vague Query Specifications. Technical Report 1181, Department of Computer Sciences, University of Wisconsin-Madison, October 1993. Master's Thesis.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore Database System. *Communications of the ACM*, 34(10):50–63, October 1991.

[MF91] G. Miller and C. Fellbaum. Semantic Networks of English. *Cognition*, 41(1-3):197–229, 1991.

[NO92] M. Nabil and S. L. Osborn. COQL: A Query Language for an Object-Oriented Database System. Unpublished manuscript, May 1992.

[Obj91] Objectivity, Inc. *Objectivity/DB Documentation Vol. 1*, 1991.

[Ont92] Ontos. *Ontos Object SQL Guide*, 2.2 edition, February 1992.

[PG88] N. W. Paton and P. M. D. Gray. Identification of Database Objects by Key. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems: 2nd International Workshop on Object-Oriented Database Systems*, pages 280–285, Berlin, Germany, September 1988. Springer-Verlag.

[RKB87] M. A. Roth, H. F. Korth, and D. S. Batory. SQL/NF: A Query Language for ¬1NF Relational Databases. *Information Systems*, 12(1):99–114, January 1987.

[Shi81] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[SKL88] S. Su, V. Krishnamurthy, and H. Lam. An Object-oriented Semantic Association Model (OSAM). In S. Kumara, A. L. Soyster, and R. L. Kashyap, editors, *A.I. in Industrial Engineering and Manufacturing: Theoretical Issues and Applications*, chapter 17, pages 463–494. American Institute of Industrial Engineering, 1988.

[SS77] J. M. Smith and D. C. P. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105–133, June 1977.

[SZ89] G. M. Shaw and S. B. Zdonik. An Object-Oriented Query Algebra. In R. Hull, R. Morrison, and D. Stemple, editors, *Proceedings of the International Workshop on Database Programming Languages*, pages 103–112, San Mateo, CA, 1989. Morgan-Kaufman, Inc.

[VD91] S.L. Vandenberg and D.J. DeWitt. Algebraic Support for Complex Objects with Arrays, Identity and Inheritance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 158–167, Denver, CO, May 1991.

[Ver91] Versant Object Technology. *VERSANT System Reference Manual, Release 1.6*, 1991.

[WCH87] M. Winston, R. Chaffin, and D. Herrmann. A Taxonomy of Part-Whole Relations. *Cognitive Science*, 11:417–444, 1987.

[WI93] J. L. Wiener and Y. Ioannidis. A Moose and a Fox Can Aid Scientists with Data Management Problems. Technical Report 1182, Department of Computer Sciences, University of Wisconsin-Madison, October 1993.

[Zan83] C. Zaniolo. The Database Language GEM. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 207–218, San Jose, CA, May 1983.