Modeling Scientific Experiments with an Object Data Model *

I-Min A. Chen and Victor M. Markowitz

Data Management Research & Development Group
Information and Computing Sciences Division
Lawrence Berkeley Laboratory, Berkeley, CA 94720
E-Mail: {IAChen, VMMarkowitz}@lbl.gov

Abstract

We examine in this paper the main requirements for modeling scientific experiments and propose constructs that fulfill these requirements. We show that existing object-oriented and semantic data models do not provide such constructs.

Experiment (protocol) and object constructs can be combined in order to provide seamless object and experiment modeling. We present an example of combining protocol and object constructs into a unified framework, the Object-Protocol Model (OPM), and briefly describe the implementation of an OPM interface on top of commercial relational database management systems (DBMSs).

1 Introduction

Scientific applications require data management facilities for keeping track and querying the data generated by experiments, simulations, and measurements [5, 11]. Data need to be described, presented, browsed and queried in a way that scientists can understand. In practice, scientists often use proven commercial database technology, that is, relational database management systems (DBMSs). For example, most major public molecular biology databases, such as Genome Database (GDB), are developed using relational DBMSs.

Relational databases do not provide constructs for directly representing application-specific structures. Application-specific objects are usually represented in a relational database by disconnected tuples that are scattered among multiple tables. Because of the complexity of such representations, the development and maintenance of relational scientific databases are tedious, error-prone, and time-consuming processes.

The object and attribute concepts of object-oriented and semantic data models [8] are closer to the concepts naturally employed by scientists to describe the data structure of their applications. Although commercial object-oriented DBMSs provide languages that are better suited for specifying complex object structures, these languages entail the development of large specifications and are often cumbersome [6].

Both relational and object-oriented data models are limited in their capability of modeling scientific experiments. Such experiments are performed in a scientific (e.g., molecular biology) laboratory, where experiments usually have an input and generate an output. Experiment modeling is characterized by an abstraction mechanism that cannot be found in object-oriented and semantic data models, namely the recursive specification of experiments in terms of alternative, sequences of, and optional experiments.

We examine in this paper the requirements for modeling scientific applications and propose constructs that fulfill these requirements. We show that using existing constructs provided by object data models for modeling scientific experiments often results in complex and semantically inaccurate representations. Consequently, we propose extending these models with a protocol class construct for modeling experiments. A protocol class can be associated with regular attributes as well as input and output attributes used for specifying input-output protocol connections, and can be specified in terms of component subprotocol classes using a protocol expansion mechanism. Protocol relationships, such as relationships between generic and component protocols, can be further specified using rules expressing the effect of inserting, deleting or updating protocol instances on related instances. Finally, different views of experiment structures can be specified using derived protocol classes.

*This work is supported by the Office of Health and Environmental Research Program of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098.

An example of combining protocol and object constructs in order to provide seamless object and experiment modeling is the *Object-Protocol Model* (OPM). OPM has similarities with other object data models in supporting object classes, class hierarchy, attributes, attribute derivations, and derived object classes. Additionally, OPM supports specifying protocol classes, input-output attributes, protocol expansion, and derived protocol classes.

The main contribution of this paper consists of proposing constructs for modeling scientific experiments, including experiment-specific views, and showing how these constructs can be incorporated into an object data model. Furthermore, we show that an object data model extended with constructs for modeling experiments such as OPM, can be implemented on top of commercial relational DBMSs.

The rest of the paper is organized as follows. In section 2, we briefly describe a reference object data model. The requirements for modeling scientific experiments and the inadequacy of using existing object data models for this purpose are discussed in section 3. New constructs for modeling experiments are proposed in section 4. Section 5 describes the implementation of an object-protocol interface on top of a relational DBMS. Section 6 contains concluding remarks and discusses related work.

2 A Reference Object Data Model

We briefly overview below the main constructs of a reference object data model that has similarities with other semantic and object data models, and primarily with the *Semantic Database Model* (SDM) [7].

2.1 Object Classes and Attributes

Objects in this reference object data model are qualified by attributes and are classified into object classes. An object class has a class name, an optional class description and is associated with attributes; a subset of the attributes is specified as the external identifier (ID) for the objects in the class. An object class can be specified as the specialization (subclass) of other object classes; a specialization object class *inherits* the attributes of all its (direct and transitive) object superclasses. Examples of object classes are shown in figure 1.

Attributes have attribute names and take values from value classes. An attribute can be simple or composite. A simple attribute (e.g., **ssn** or **owns** of class

```
OBJECT CLASS Person
   ID: ssn
   ATTRIBUTE ssn: CHAR(11)
                  single-valued not null
   ATTRIBUTE name(firstName, middleIn, lastName):
                (CHAR(30), CHAR(1), CHAR(40))
   ATTRIBUTE title: CHAR(20)
   ATTRIBUTE affiliation: CHAR(80)
   ATTRIBUTE owns: Filter or Probe
                  multi-valued
   ATTRIBUTE ownedProbeName
                  DERIVATION: owns [Probe] name
OBJECT CLASS Probe
   ID: probeID
   ATTRIBUTE probeID: INTEGER
                  single-valued not null
   ATTRIBUTE name: CHAR(40)
   ATTRIBUTE description: VARCHAR(80)
   ATTRIBUTE subtype: ProbeSubtype
   ATTRIBUTE status: CHAR(20)
   ATTRIBUTE owner: Person
                  inverse of PERSON.owns
OBJECT CLASS Filter
   ID: filterID
   ATTRIBUTE filterID: INTEGER
                  single-valued not null
   ATTRIBUTE description: VARCHAR(80)
   ATTRIBUTE owner: PERSON
                  inverse of PERSON.owns
CONTROLLED VALUE CLASS ProbeSubtype
   { "unknown", "Cloned", "PCR" }
```

Figure 1: Examples of Object Classes

Person in figure 1) is associated with either a single value class (e.g., **ssn**) or a union of several value classes (e.g., **owns**). A composite attribute (e.g., attribute **name** of **Person**) consists of several component simple attributes.

Depending on the type of the associated value class, an attribute can be primitive or abstract. A primitive attribute is an attribute associated with a primitive class of atomic values of predefined data types (e.g., INTEGER, etc.) or a controlled class of enumerated atomic values (e.g., **ProbeSubtype** in figure 1). An abstract attribute (e.g., **owns** of **Person**) is an attribute whose associated value class is an object class or a union of object classes.

The structure of objects (instances) in an object class is defined below.

Definition 1 (Object Class Instances) *Let O_i be an object class and x be an instance (object) of O_i (denoted $x \in O_i$). Then $x = (oid(x), val(x))$, where $oid(x)$ is the globally unique (internal) object identifier*

of x and $val(x)$ is the value of x. $oid(x)$ takes values from a system-provided primitive value class of identifiers. $val(x)$ has the form $(A_1 : A_1(x), \ldots, A_n : A_n(x))$, where A_j, $1 \leq j \leq n$, is the name of an attribute of O_i, and $A_j(x)$ denotes the value of attribute A_j for x and is defined as follows:

1. *If A_j is a simple attribute associated with primitive value class P, then $A_j(x)$ consists of a set of elements from the domain of P.*
2. *If A_j is a simple abstract attribute associated with value class: O_1 or ... or O_m, then $A_j(x)$ is a subset of $\{ oid(y) \mid y \in O_k, 1 \leq k \leq m \}$.*
3. *If $A_j = (A_{j_1}, \ldots, A_{j_n})$ is a composite attribute, then $A_j(x)$ consists of a set of tuples of the form $[A_{j_1} : A_{j_1}(x), \ldots, A_{j_n} : A_{j_n}(x)]$, where each component $A_{j_k}(x)$, $j_1 \leq j_k \leq j_n$, is defined as in 1 and 2 above.*

By definition, if y is a value from a primitive value class then $oid(y) = y$.

Attributes are characterized by attribute constraints. Cardinality constraints regard the cardinality (single-valued or multi-valued) of the set of values an attribute can have for each object in a class. Null constraints refer to whether an attribute can have null (empty set) values. Inverse constraints regard pairs of abstract non-derived attributes and express object cross referencing: if an abstract attribute A of object class O_i (e.g., **owner** of **Probe** in figure 1) is defined as the inverse of abstract attribute B of class O_j, then for every instance x of O_i and every instance y of O_j, $oid(y) \in A(x) \Rightarrow oid(x) \in B(y)$.

2.2 Derived Attributes

Derived attributes are simple attributes that have values derived from the values of other attributes using arithmetic expressions, aggregate functions (min, max, sum, avg, count), or attribute composition. Arithmetic expression and aggregate function derivations are straightforward and are not further discussed here.

A simple attribute A of class O_i is a composition of other attributes if it is associated with a composition derivation consisting of a union of paths of the form $B_1\ [O_{i_1}]\ B_2\ [O_{i_2}] \ldots B_n\ [O_{i_n}]$, where O_{i_k} $(1 \leq k \leq n)$ denotes an object class and B_k $(1 \leq k \leq n)$ denotes

(i) an attribute C, where C is associated with $O_{i_{(k-1)}}$ $(O_{i_0} = O_i)$ and has a value class that either includes or is a superclass of O_{i_k}; or

(ii) the inverse of an attribute C, $!C$, where C is associated with O_{i_k} and has a value class that includes $O_{i_{(k-1)}}$ $(O_{i_0} = O_i)$.

In a path, in each subexpression of the form $B_k\ [O_{i_k}]\ B_{k+1}$, $(1 \leq k \leq n-1)$, where B_k denotes an attribute that has a value class consisting only of O_{i_k}, $[O_{i_k}]$ can be replaced by a dot (.); similarly, the last element in the path, $[O_{i_n}]$, can be omitted if only O_{i_n} is involved in the value class of attribute B_n.

The value of attribute A for x is the union of all the values that are derived for x from the paths involved in the composition derivation.

Definition 2 (Attribute Composition Value) *Let A be an attribute of class O_i associated with an attribute composition derivation of the form mentioned above. For each object x of O_i, $A(x)$ contains $oid(y)$ iff there exists a sequence of objects (instances) $x = x_0, x_1, \ldots, x_{n-1}$, followed by object or primitive value $x_n = y$, s.t. $\forall\ j = 1, \ldots, n$,*

1. *if $B_j = C$, then $x_j \in O_{i_j}$ and $oid(x_j) \in C(x_{j-1})$;*
2. *if $B_j = !C$, then $x_j \in O_{i_j}$ and $oid(x_{j-1}) \in C(x_j)$.*

Attribute **ownedProbeName** of object class **PERSON** in figure 1 is an example of a composition derived attribute.

2.3 Derived Object Classes

A derived object class can be (i) a subclass (specialization) of one or several object classes; (ii) a superclass (generalization) of several object classes; or (iii) an aggregate object class.

Attribute derivation in a derived class is similar to attribute derivation in an object class with attribute composition for a derived class, O_i, extended to: $[O_{j_0}]\ A_1\ [O_{j_1}] \ldots A_n\ [O_{j_n}]$, where O_{j_0} is a class involved in the derivation of O_i. O_{j_0} in the derivation is *optional* if A_1 is an (inherited) attribute of O_i.

A specialization derived object class, O_s, is specified as a subclass of one or (intersection of) several object classes, $O_1, \ldots, O_m$ $(m \geq 1)$, possibly required not to overlap other object classes, $O_{m+1}, \ldots, O_{m+n}$ $(n \geq 0)$, and/or using a condition. O_s consists of the subset of objects that belong to the intersection of classes O_i, $1 \leq i \leq m$, satisfy the associated condition (if any), and do not belong to classes O_{m+j}, $1 \leq j \leq n$ (if any).

A generalization derived object class, O_g, is defined as the superclass of object classes $O_1, \ldots, O_m$ $(m \geq 2)$, and consists of the union of objects belonging to these classes. O_g is associated with a derived identifier attribute that is defined as the union of the identifiers of classes $O_1, \ldots,$ and O_m.

An aggregate object class, O_a, consists of objects that are aggregations of objects from other object classes, $O_1, \ldots, O_m$ $(m \geq 2)$. The objects of O_a

are virtual objects that are in a one-to-one correspondence with the subset of the cross product of classes O_i, $1 \leq i \leq m$, consisting of tuples of objects that satisfy the associated condition.

3 Modeling Scientific Experiments

We present in this section an example of a scientific experiment and discuss the requirements for modeling such experiments. Then we show that the constructs usually provided by an object data model are not adequate for modeling scientific experiments.

3.1 Requirements

Experiments are instances of procedures (e.g., genomic sequencing protocols) performed in scientific (e.g., molecular biology) laboratories. Experiments are characterized by properties such as time or location. In addition, experiments

1. transform some input (resources) into output (experimental results);
2. can be preceded by other experiments that provide their input and/or can be succeeded by other experiments for which they generate (output that is taken as) input;
3. can be components of more generic experiments or can consist of sequences component experiments and/or alternative component experiments, where an experiment and its component experiments may share common inputs and/or outputs.

An example of a scientific experiment is the molecular biology laboratory protocol for labeling and blotting an electrophoretic gel. This experiment is depicted diagrammatically in figure 2 using a notation similar to the Data Flow Diagram (DFD) notation. Specifically, one can either label the gel directly (e.g., by staining) or one can transfer the DNA from the gel to a filter via southern blotting, and then probe the filter with a radioactively labeled hybridization probe, where southern blotting is optional [17].

Hence, a **Labeled Separation** sample (output) is the result of a **Label** experiment (protocol) instance applied on an **Electrophoretic Gel** (input). **Label** involves (i.e., can be carried out by) two alternative experiments: (i) a **Stain** experiment, or (ii) an optional **Southern Blot** experiment followed by a **Hybridization** experiment, where a **Hybridization** experiment generates a **Probed Filter** from a **Filter** generated by a **Southern Blot** experiment or directly

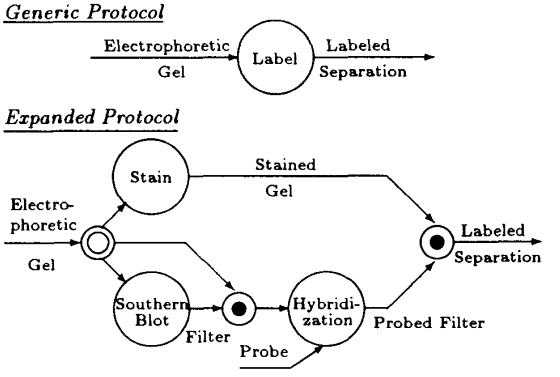

Figure 2: Protocol **Label**

from an **Electrophoretic Gel**. Note that experiments can share input and output. For example, an **Electrophoretic Gel** can be input for either **Stain** or **Southern Blot** which both eventually produce a **Labeled Separation**.

Relationships between experiments are further characterized by existence dependency rules. These rules express the enforcement of a specific order between experiments, allowing (or not allowing) to record experimental data when the input or output is unknown, etc. For example, **Hybridization** experiments can be required to have as input either non-null **Electrophoretic Gels** or **Filters** that are generated by some **Southern Blot** experiment. In an environment where missing data is tolerated or cannot be avoided, less restrictive requirements can be also expressed using existence dependency rules.

3.2 Modeling Problems

Constructs provided by object data models are not adequate for the accurate modeling of scientific experiments. Thus, the relationship between a generic experiment and its component experiments cannot be modeled as a class-subclass relationship or using attributes. Consider, for example, the molecular biology protocols described in the previous subsection and shown in figure 2. If **Southern Blot** and **Label** are represented using object classes then **Southern Blot** cannot be defined as a subclass of **Label** (and **Label** cannot be defined as a subclass of **Southern Blot**) since **Southern Blot** experiments do not share (inherit) output with **Label** experiments. Furthermore, **Label** experiments can be carried out using **Stain** experiments, and therefore some **Label** instances are not

even related to **Southern Blot** instances.

Modeling the relationship between a generic experiment and its component experiments using (regular) attributes is also inadequate. For example, suppose that **Label**, **Stain**, **Southern Blot**, and **Hybridization** are represented using object classes as follows:

```
ATTRIBUTE alternative1: Stain
ATTRIBUTE alternative2 (comp_1, comp_2):
                (SouthernBlot, Hybridization)
ATTRIBUTE alternative3: Hybridization
```

A representation such as that shown above requires using an attribute for each combination (alternative) of component experiments that form a generic experiment. Moreover, the order between component experiments or the fact that some components are optional is not properly captured. Similarly, the input and output relationships of experiments, (e.g., characterized by inclusion dependencies between the input and output of related experiments) cannot be modeled using regular object class attributes.

Experiment and input-output relationships can be expressed using *procedures* (e.g., Sybase triggers) in relational DBMSs or using *methods* in object-oriented DBMSs. However, this involves developing manually numerous and hard to comprehend procedures in a dialect of SQL or a programming language such as C/C++ or Smalltalk. Moreover, SQL procedures or methods developed for a specific DBMS cannot be ported to other DBMSs without substantial rewriting.

It is clear from the above discussion that constructs necessary for modeling experiments are not directly provided by general purpose semantic and object-oriented data models or even data models developed specifically for experimental scientific database applications such as *MOOSE* [9], *POSDBM* [14], or *ordering information model* [12]. Modeling experiments as objects fails to capture the semantics of relationships (e.g., precedence) specific to experiments.

Failure of supporting appropriate constructs for modeling experiments impairs not only the clarity and accuracy of representing experiments in a database, but also the way experiments can be viewed and queried. Providing constructs for modeling directly experiments allows developing query and view facilities that include experiment-specific constructs. Consider, for example, a database containing data only on experiments **Stain**, **Southern Blot**, and **Hybridization** mentioned above, but not on experiment **Label**. Then, for scientists interested in (generic) experiment **Label** rather than in the details of **Stain**, **Southern Blot**, and **Hybridization**, **Label** can be provided as a view.

4 Protocol Classes and Attributes

The protocol constructs presented in this section are designed to fulfill the requirements for modeling experiments discussed in the previous section. These constructs can be incorporated into an object data model, such as that described in section 2.

4.1 Protocol Classes

Protocol classes allow modeling experimental laboratory procedures. Similar to an object class, a protocol class has a class name, an optional class description, an identifier, and attributes. For example, protocols **Label**, **Stain**, **SouthernBlot**, and **Hybridization** described in section 3 can be modeled by the protocol classes shown in figure 3.

For expressing a generic experiment in terms of component experiments a protocol expansion mechanism is provided. Protocol expansion allows specifying *alternative* protocols, *sequences* of protocols, and *optional* protocols; "*or*", ",", and "[ ]" are used to denote alternative, sequences of, and optional protocols, respectively, and parentheses are used for specifying complex protocol compositions. For example, the expansion of **Label** in terms of **Stain**, **SouthernBlot**, and **Hybridization** is expressed by associating protocol class **Label** with the expansion specification shown in figure 3.

In addition to regular attributes representing various experimental parameters, such as time and temperature, a protocol class is in general associated with special attributes representing input, output and experiment connections.

4.2 Input and Output Attributes

Input and output attributes associated with a protocol class are simple attributes that represent input and output data regarding the experiment modeled by the protocol class. Additionally, these attributes are used to express sharing input or output attributes between a generic protocol and its component subprotocols as well as input-output connections of directly related protocols.

If a generic protocol is expanded into component (sub)protocols, then its input and output attributes are inherited by its subprotocols. This input-output attribute inheritance is expressed using 'input isa ...' statements (e.g., attribute **electroGel** of **Stain** in figure 3) and 'output isa ...' statements (e.g., attribute **probedFilter** of **Hybridization** in figure 3) in the

specification of the input and output attributes associated with subprotocols.

A protocol can take as input the output generated by another protocol. Thus, if a protocol P_i is followed directly by another protocol, P_j, then the input of P_j includes some or all of the output of P_i, and P_j is said to *take input* from P_i. Such input-output protocol connections are expressed using 'input from ... via ...' statements (e.g., attribute **gelFilter** of **Hybridization** in figure 3) in the specification of input attributes associated with protocols taking their input from other protocols: 'from ...' denotes a protocol and 'via ...' denotes an output attribute of the from protocol.

An input attribute can be defined with both input isa and input from specifications such as the input attribute **gelFilter** of protocol **Hybridization**. An input attribute can be associated with at most one input isa specification, thus reflecting the real-world restriction of an experiment being able to represent a step of at most one other experiment. The number of input from statements in an input attribute, however, is not restricted.

Two *system*[1] derived attributes, **InputFor** and **OutputOf**, are maintained for each object class, O_i, involved in the value class of an input and output attribute, respectively: **InputFor** (resp. **OutputOf**) is defined as a composition of attributes as follows: $!A_{j_1}$ $[P_{j_1}]$ or ... or $!A_{j_m}$ $[P_{j_m}]$, where for each $1 \leq k \leq m$, A_{j_k} is an input (resp. output) attribute of protocol class P_{j_k} and is associated with a value class involving O_i. These attributes are used for expressing existence dependency rules between protocols and their input and output (see section 4.4 below). For example, since object class **StainedGel** contains instances of the output of **Stain** and **Label** protocols, an attribute **OutputOf** is automatically generated for class **StainedGel**, where this attribute is associated with value class **Label** or **Stain**, and has the following derivation: **!labeledSeparation [Label] or !stainedGel [Stain]**.

Input-output attribute specifications imply input-output consistency constraints. Thus, for each pair of protocol classes P_i and P_j,

1. if P_j is the generic protocol class of P_i, then for each input or output attribute of P_i, A, that is specified using an input isa $P_j.B$ or output isa $P_j.B$ specification, the A value for each protocol of P_i must be a subset of the B value for the related generic protocol of P_j;
2. if P_i succeeds P_j, then for every input attribute of P_i, A, related using a statement involving a from ... via specification involving an output attribute of P_j, B, the values of A and B for two related protocols of P_i and P_j must be related according to the from ... via specification.

[1] System and connection attributes are automatically generated by the system during implementation based on protocol expansion and input-output connections. Therefore, they do not appear in the schema definition.

```
PROTOCOL CLASS Label
   ID: labelID
   EXPANSION: Stain or
              ([SouthernBlot], Hybridization)
   ATTRIBUTE labelID: INTEGER
              single-valued not null
   ATTRIBUTE electroGel: ElectroGel  input
   ATTRIBUTE labeledSeparation:
              StainedGel or ProbedFilter  output
PROTOCOL CLASS Stain
   ID: stainID
   ATTRIBUTE stainID: INTEGER
              single-valued  not null
   ATTRIBUTE electroGel: ElectroGel
              input isa Label.electroGel
   ATTRIBUTE stainedGel: StainedGel
              output isa Label.labeledSeparation
PROTOCOL CLASS SouthernBlot
   ID: expID
   ATTRIBUTE expID: INTEGER
              single-valued not null
   ATTRIBUTE electroGel: ElectroGel
              input isa Label.electroGel
   ATTRIBUTE gelNitro: GelNitrocellulose
   ATTRIBUTE filter: Filter  output
PROTOCOL CLASS Hybridization
   ID: hybID
   ATTRIBUTE hybID: INTEGER
              single-valued  not null
   ATTRIBUTE gelFilter: ElectroGel or Filter
              input isa Label.electroGel or
                   from SouthernBlot via filter
   ATTRIBUTE probe: Probe   input
   ATTRIBUTE probedFilter: ProbedFilter
              output isa Label.labeledSeparation
```

Figure 3: Protocol Label and its Components

4.3 Connection Attributes

Connection[1] attributes are used for maintaining relationships between experiments. A protocol class, P_i, can have the following relationships with other protocol classes:

1. P_i can represent an experiment that is a component of another generic experiment (represented

by another protocol class); an experiment can be a component of at most one generic experiment;
2. P_i can represent an experiment that consists of component (sub) experiments;
3. P_i can represent an experiment that is preceded by other experiments that provide its input;
4. P_i can represent an experiment that is succeeded by other experiments for which it provides input.

Let the four relationships listed above be represented by attributes **GenericProtocol**, **Subprotocols**, **PredecessorProtocols**, and **SuccessorProtocols**, respectively, associated with P_i. Several observations regarding these attributes are in order:

1. The value class associated with these attributes are implied by a given protocol class structure as follows: the value class of **GenericProtocol** is the generic protocol class of P_i; the value class of **Subprotocols** is the union of the component (sub) protocol classes of P_i; the value class of **PredecessorProtocols** is the union of the protocol classes directly preceding P_i; the value class of **SuccessorProtocols** is the union of the protocol classes directly succeeding P_i. Thus, the value classes of these attributes changes whenever a protocol class is added or removed during the process of specifying the protocol structure.
2. **Subprotocols** and **SuccessorProtocols** can be expressed as (derived) attribute compositions. Thus, **Subprotocols** = !**GenericProtocol** $[P_{j_1}]$ or ... or !**GenericProtocol** $[P_{j_m}]$, where for each $1 \leq k \leq m$, P_{j_k} is specified as a component (sub) protocol class of P_i; and **SuccessorProtocols** = !**PredecessorProtocols** $[P_{j_1}]$ or ... or !**PredecessorProtocols** $[P_{j_m}]$, where for each $1 \leq k \leq m$, P_{j_k} is specified as a protocol class succeeding P_i.

The observations above suggest that connection attributes **GenericProtocol**, **Subprotocols**, **PredecessorProtocols**, and **SuccessorProtocols** should be generated automatically for a protocol class. For example, since protocol **Hybridization** can be preceded by protocol **SouthernBlot**, and is a component of protocol **Label**, **Hybridization** can be automatically associated with attribute **GenericProtocol** with value class **Label**, and attribute **PredecessorProtocols** with value class **SouthernBlot**.

4.4 Rules

Protocol classes are associated with rules that express the enforcement of protocol-specific constraints, such as input-output consistency constraints. Of special interest for representing experiment relationships are the delete rules that express the effect of deleting a protocol on its input, output, and related protocols. Thus, the delete rule for a protocol class, P_i, includes specifying component delete rules associated with connection attributes **GenericProtocol**, **PredecessorProtocols**, **Subprotocols**, and **SuccessorProtocols** involving P_i, and system attributes **InputFor** and **OutputOf** involving P_i (i.e., whose value classes include P_i).

Component delete rules associated with connection attributes can have multiple options. The component delete rule of a protocol class P_i with regard to a connection attribute A can be:

1. if A is **InputFor** (or **OutputOf**) associated with object class O_j, then the delete rule can be: (i) **nullifies**, meaning that deleting a P_i protocol having an O_j object x as input (or output) entails nullifying the value of A for x (i.e., has no effect on O_j); or (ii) **cascades**, meaning that deleting a P_i protocol having an O_j object x as input (or output) entails deleting x from O_j;
2. if A is **GenericProtocol** (or **Subprotocols**, **PredecessorProtocols**, **SuccessorProtocols**) associated with protocol class P_j, then the delete rule can be: (i) **restricted**, meaning that a P_i protocol x cannot be deleted before deleting the P_j protocol related to x; (ii) **cascades**, meaning that deleting a P_i protocol x entails deleting the P_j protocol related to x; or (iii) **nullifies**, meaning that deleting a P_i protocol x entails nullifying the A value for the P_j protocols related to x.

Consider the object and protocol classes shown in figures 1 and 3, respectively. The delete rule associated with **SouthernBlot** involves component delete rules with regard to:

(i) attribute **Subprotocols** of protocol class **Label**; for example, if this rule is **nullifies** then deleting a **SouthernBlot** protocol will not affect the corresponding generic **Label** protocol;

(ii) attribute **OutputOf** of object class **Filter**; for example, if this rule is **cascades** then deleting a **SouthernBlot** protocol will delete the corresponding output **Filter** objects;

(iii) attribute **PredecessorProtocols** of protocol class **Hybridization**; for example, if this rule is **restricted** then a **SouthernBlot** protocol cannot be deleted unless there are no **Hybridization** protocols related to (i.e., succeeding) it.

4.5 Derived Protocol Classes

The derived object class constructs presented in section 2 are not adequate for specifying experiment-specific views for the same reasons as those discussed in section 3.2 regarding the inadequacy of object class constructs for modeling experiments. Consequently, we propose two derived protocol class constructs for specifying experiment-specific views.

A derived protocol class can be (i) a derived subprotocol class representing a subset of experiments satisfying a given condition or a component of an existing experiment; or (ii) a derived generic protocol class representing experiments that are constructed from experiments represented as instances of existing protocol classes.

A derived subprotocol class, P_s, consists of protocols representing a subset or a component (step) of an existing protocol class, P_i, and is specified by refining the definition of P_i. The definition of a derived subprotocol class is similar to the derivation of a specialization derived class, except that it only involves a single underlying (generic) protocol. This reflects the real-world restriction of an experiment being able to represent a step of at most one other experiment.

A subprotocol class inherits all attributes from the underlying protocol class, P_i, unless attributes are redefined in order to override the definition of attributes of P_i. A significant difference between derived subprotocol classes and derived specialization classes is the capability of redefining regular attributes of underlying protocol classes as input/output attributes of derived subprotocol classes. We will illustrate such a redefinition using an example. The `Southern Blot` protocol mentioned in section 3 consists actually of two steps: (i) an electrophoretic gel is first transferred to gel nitrocellulose, and (ii) then a flow of buffer is set up through the gel to cause the DNA fragments to flow out of the gel and bind to the filter [17]. Suppose that one is interested in the first step mentioned above and its intermediate result, gel nitrocellulose. A subprotocol class called `GelTransfer` can be defined as derived from generic protocol `SouthernBlot` as shown in figure 4. Derived subprotocol `GelTransfer` preserves the definition of input attribute `electroGel` from `SouthernBlot`, but redefines attribute `gelNitro` as an output attribute.

A derived generic protocol class, P_g, is associated with a derivation consisting of a protocol expansion expression involving existing protocols or derived protocols, where protocol connections defined in the underlying schema must be followed.

For example, a derived generic protocol class

```
DERIVED PROTOCOL CLASS GelTransfer
   DERIVATION: subclass of SouthernBlot
   ATTRIBUTE electroGel       input
        DERIVATION: [SouthernBlot] electroGel
   ATTRIBUTE gelNitro         output
        DERIVATION: [SouthernBlot] gelNitro
DERIVED PROTOCOL CLASS LabelThroughHybridization
   DERIVATION: [SouthernBlot] , Hybridization
   ATTRIBUTE expID
        DERIVATION: [Hybridization] HybID
   ATTRIBUTE gelFilter        input
        DERIVATION: [SouthernBlot] electroGel or
                    [Hybridization] gelFilter
   ATTRIBUTE probe            input
        DERIVATION: [Hybridization] probe
   ATTRIBUTE probedFilter     output
        DERIVATION: [Hybridization] probedFilter
```

Figure 4: Derived Protocol Class Examples

`LabelThroughHybridization` can be specified as consisting of an optional `SouthernBlot` protocol followed by a `Hybridization` protocol as specified in figure 4, where these two protocols are connected through output attribute `filter` of `SouthernBlot` and input attribute `gelFilter` of `Hybridization`.

A derived generic protocol class can redefine attributes of the underlying protocol classes representing (intermediate) inputs and outputs that are of no interests, as regular (i.e., non-input, non-output) attributes.

5 Object-Protocol Model Interface

We have integrated the protocol class constructs presented in the previous section with the constructs of an object data model similar to that presented in section 2. The result of this integration is the Object-Protocol Model (OPM). OPM provides a unified framework for modeling object and experiment structures. OPM is described in detail in [3].

OPM is currently used for developing several large genomic databases that are using commercial relational DBMSs such as Sybase. Consequently, we have developed data management tools that provide an OPM interface on top of relational DBMSs. Thus we have developed a graphical editor for specifying OPM schemas and a translator that maps OPM schemas into complete relational DBMS specifications.

The OPM schema translator works in two stages: (1) first, the translator generates system and connection attributes corresponding to the protocol class

and input/output attribute specifications in the OPM schema, and (2) next, the translator maps the OPM schema into relational (schema and SQL query) specifications. We briefly discuss below the second stage of the translator.

In a relational database, data on instances of an object or protocol class can be kept in one or several normalized relations,[2] or in one unnormalized relation. Using normalized relations has the advantage of reduced data redundancy and simpler data integrity maintenance, but has the disadvantage of data fragmentation, that is, having data on one object or protocol scattered among multiple relations. Conversely, using unnormalized relations has the advantage of having data on one object or protocol available in a single relation, but has the disadvantage of high data redundancy and complex data integrity maintenance.

One can benefit from both alternatives mentioned above by keeping data in normalized relations, while providing procedures for constructing *relational views* that are in a one-to-one correspondence with the object and protocol classes and that also include the values corresponding to the derived attributes. The DBMS specifications for constructing these views represent the retrieval methods and may involve large and multiple SQL queries. Thus, if data on the instances of an object or protocol class are kept in one normalized *primary* relation and several normalized *auxiliary* relations,[3] the relational view corresponding to this class involves recursively left-outerjoining the primary relation with the auxiliary relations.

Informally, mapping an OPM schema into a relational DBMS definition consists of mapping every OPM object or protocol class O_i into a relation-scheme R_i; depending on their type (primitive, abstract, simple, composite, etc.), non-derived attributes of O_i are mapped into local attributes of R_i, foreign-key attributes of R_i, and additional relation-schemes with appropriate foreign-key to primary-key references. The mapping of O_i also involves incrementally generating a retrieval query representing the OPM data retrieval method associated with O_i, and update queries representing the OPM insert, delete, and update methods associated with O_i. The retrieval query specifies the construction of O_i instances from R_i tuples together with related tuples representing the attribute values of O_i. Such a query involves in general outer joining the relation associated with R_i with (auxiliary) relations containing tuples representing the attribute values of O_i. Similarly, the update queries specify the insertion and deletion of O_i instances by inserting or deleting tuples into R_i together with related tuples representing the attribute values of O_i, and by inserting and deleting (if necessary) additional tuples as defined by the OPM rules associated with O_i. Derived attributes are not mapped into DBMS schema components and entail only modifying retrieval queries.

Mapping OPM schemas into relational database definitions and queries is defined in [4]. The OPM schema translator implementing this mapping as well as other OPM tools and OPM documents are available via World Wide Web using URL: `http://gizmo.lbl.gov/DM_TOOLS/OPM/opm.html`.

6 Concluding Remarks

We have examined in this paper the requirements for modeling scientific experiments and showed that existing data models are inadequate for this purpose. We have presented constructs for modeling scientific experiments and have illustrated the main constructs for modeling experiments with an example taken from the field of genomic databases. A full fledged example of using these constructs for modeling a large scale DNA sequencing database application in a molecular biology laboratory can be found in [3].

The constructs for modeling experiments presented in this paper can be integrated into an object data model. An example of such an integration is the Object-Protocol Model (OPM). We described the implementation of an OPM interface on top of relational DBMSs. OPM and the OPM data management tools are currently used for developing several genomic databases, such as the new version of the public Genomic DataBase (GDB) at Johns Hopkins School of Medicine, Baltimore.

Modeling scientific experiments is related to several database and information system research areas. This relationship is briefly discussed below.

Semantic and Object-Oriented Data Models. Semantic and object-oriented data models [2, 7, 8] provide capabilities for modeling object classes, attributes, and subclass hierarchies. However, these data models do not provide constructs for modeling experiments (see section 3).

[2] More than one relation may be required in order to represent constructs that are not supported directly in relational DBMSs, such as set valued attributes.

[3] The primary relation can contain the identifiers of (object and protocol) instances of an object or protocol class together with their primitive single-valued attribute values, while auxiliary relations can contain the values of attributes that cannot be represented in the normalized primary relation.

Data Models for Scientific Applications. Several data models have been designed in order to support modeling scientific applications. For example, *MOOSE* [9] has been designed for simulation environments, *POSDBM* [14] is a process-oriented scientific data model that provides capabilities for defining processes, transitions, and inputs/outputs, and *ordering information model* [12] is a data model that has been designed for genomic applications. Although modeling processes and inputs/outputs is supported by some of these data models, none of these models provides the constructs required for modeling experiments, such as those presented in section 4.

Procedural Modeling in Office Information Systems. The modeling of experiments is related to the procedural modeling in office information systems [10, 13] in the sense that both involve specifying how a procedure is carried out in terms of subprocedures or steps. However, procedures in an office environment are generally well defined, and the inputs and outputs of these procedures (in most cases, *forms* and *documents*) have simple structures. In contrast, the constructs proposed in this paper are intended mainly for modeling scientific experiments that can involve alternative, sequences of, and optional experiments and can have complex inputs and outputs.

Object-Oriented Views. Object-oriented views have been explored in several papers, such as [1, 15, 16]. These papers deal with derived object class constructs and do not have constructs similar to the derived protocol class constructs presented in this paper.

References

[1] Abiteboul, S., and Bonner, A., Objects and Views, Proc. of the 1991 *ACM SIGMOD* International Conference on Management of Data, SIGMOD Record 20, 2, 1991.

[2] Bancilhon, F., Delobel. C., and Kanellakis, P., Building an Object-Oriented Database System: The Story of O_2, Morgan Kaufmann Publishers, Inc., 1992.

[3] Chen, I.A., and Markowitz, V.M., The Object-Protocol Model (Version 2.4), Lawrence Berkeley Laboratory Technical Report LBL-32738, 1994.

[4] Chen, I.A., and Markowitz, V.M., Mapping Object-Protocol Schemas into Relational Database Schemas and Queries (OPM Version 2.4), Lawrence Berkeley Laboratory Technical Report LBL-33048, 1994.

[5] Frenkel, K.A., The Human Genome Project and Informatics, *Communications of ACM*, **34**, 11, (November 1991).

[6] Goodman, N., Reeve, M.P., and Stein, L., The design of MapBase: An Object-Oriented Database for Genome Mapping, Whitehead Institute for Biomedical Research, 1992.

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

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

[9] Ioannidis, Y.E., and Livny, M., MOOSE: Modeling Objects in a Simulation Environment, *Information Processing 89*, G.X. Ritter (ed), Elsevier Science Publishers B.V., 1989.

[10] King, R., and McLeod, D., A Database Design Methodology and Tool for Information Systems, *ACM Transactions on Office Information Systems*, **3**, 1 (January 1985).

[11] Lander, E.S., Langridge, R., and Saccocio, D.M., Mapping and Interpreting Biological Information, *Communications of ACM*, **34**, 11, (November 1991).

[12] Lee, A.J., Rundensteiner, E.A., Thomas, S., and Lafortune., S., An Information Model for Genome Map Representation and Assembly, *2nd Inter. Conf. on Information and Knowledge Management*, November 1993.

[13] Pernici, B., et. al., C-TODOS: An Automatic Tool for Office System Conceptual Design, *ACM Transactions on Information Systems*, **7**, 4 (October 1989).

[14] Pratt, J.M., and Cohen, M., A Process-Oriented Scientific Database Model, *ACM SIGMOD Record*, **21**, 3, (September 1992).

[15] Rundensteiner, E.A., MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases, *Proc. of the 18th VLDB Conference*, 1992.

[16] Scholl, M.H., Laasch, C., and Tresch, M., Updatable Views in Object-Oriented Databases, in *Deductive and Object-Oriented Databases*, Lecture Notes in Computer Science, vol. 566, C. Delobel & al.(eds), Springer-Verlag, 1991.

[17] Watson, J.D., Tooze, J., and Kurtz, D.T., Recombinant DNA: A Short Course, Scientific American Books, New York, New York, 1983.