AN OVERVIEW OF THE OBJECT PROTOCOL MODEL (OPM) AND THE OPM DATA MANAGEMENT TOOLS[†]

I-MIN A. CHEN and VICTOR M. MARKOWITZ

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

(Revised April 23, 1995)

Abstract — In this paper, we overview the Object-Protocol Model (OPM) and a suite of data management tools based on OPM. OPM is a data model that allows specifying database structures and queries in terms of objects and protocols specific to scientific (e.g., molecular biology laboratory) applications. Thus, scientific experiments and their resources can be described using OPM in a unified way. OPM data management tools provide facilities for specifying and querying relational databases in terms of OPM constructs, and automatically generate database specifications and queries for implementing OPM on top of commercial relational database management systems (DBMSs). OPM tools increase the efficiency of developing scientific databases using relational DBMSs, while insulating scientists from the underlying DBMSs.

Key words: data management tools, object data model, scientific database

1. INTRODUCTION

Keeping track and querying data generated by scientific experiments, simulations, and measurements require database management facilities [12, 18]. In practice, scientists often use proven commercial database management technology, that is, relational database management systems (DBMSs). For example, numerous public molecular biology databases, such as the Genome Data Base (GDB) at Johns Hopkins School of Medicine, Baltimore, the Genome Sequence Data Base (GSDB) at the National Center for Genome Resources, Santa Fe, the Flybase at Harvard University, the Protein Data Bank (PDB) at Brookhaven National Laboratory, and the Mouse Genome Database (MGD) at Jackson Laboratory, Maine, are developed using the Sybase relational DBMS. It has been predicted that in the near future genomic applications cannot avoid using multiple relational and object-oriented DBMSs [12].

Relational database languages are at a lower level of abstraction than most scientific applications and do not provide constructs for directly representing application-specific structures. Application-specific objects are usually represented in a relational database by disconnected tuples scattered among multiple tables. Consequently, relational database definitions for scientific databases are usually large and hard to comprehend, and entail tedious, error-prone, and time-consuming database development and maintenance processes. Commercial object-oriented DBMSs languages are better suited for specifying complex object structures, but also entail the development of large specifications and are often cumbersome [13]. Moreover, these DBMSs provide poor (if at all) query and view mechanisms. Consequently, there is a need for data management tools that would allow scientists to interact efficiently with both relational and object-oriented DBMSs.

The object and attribute constructs of object-oriented [3] and semantic data models [15] are closer to the concepts naturally employed by scientists to describe the data structure of their applications. These data models, however, are still limited in supporting different views of the same application and in their capability of modeling scientific experiments (protocols). Experiments are instances of procedures such as genome sequencing protocols performed in molecular biology laboratories; each individual experiment has an input and generates an (experimental result) output. Protocol modeling is characterized by an abstraction mechanism that cannot be found in object-oriented and semantic data models, namely the recursive specification of protocols in terms of alternative, optional, and sequences of protocol steps.

Because of the inadequacy of traditional data models for modeling scientific experiments [8], we have developed the *Object-Protocol Model* (OPM). At the same time, we have developed a

[†]In press, *Information Systems*, 1995

suite of OPM-based data management tools that allow constructing efficiently scientific as well as traditional (e.g., financial) database applications using commercial relational DBMSs. OPM has similarities with other object-oriented and semantic data models in supporting classes, inheritance, and (regular and derived) attributes. Unlike other object-oriented and semantic data models, OPM provides 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. OPM also provides *update rules* for expressing the effect of inserting, deleting or updating an OPM database (non-derived) element on related elements.

OPM supports the specification of different *views* of a database application and provides a query language. An OPM view consists of (a subset of) OPM classes specified in an OPM schema together with new *derived* object and protocol classes. Every OPM view is associated with a view construction method. OPM *queries* consist of associating conditions with base or derived (view) classes. Complex (e.g., multi-class) queries are constructed using nested query expressions and/or using derived classes and derived attributes.

Implementing OPM data management tools targeting specific DBMSs involves automatically translating OPM schemas into DBMS database definitions and procedures, where the DBMS procedures are used for maintaining constraints (e.g., referential integrity constraints) and for implementing OPM retrieval and update methods. We have developed such tools for relational DBMSs and plan to develop similar tools for object-oriented DBMSs. The OPM tools are currently used for developing several large scientific databases, such as version 6 of GDB at Johns Hopkins School of Medicine, and the new version of PDB at Brookhaven National Laboratory. OPM data management tools, documents, and examples of using these tools are available via World Wide Web:†

We have developed the following OPM-based data management tools (see figure 1):

1. a graphical editor for specifying OPM schemas (OPM Schema Editor);

2. a translator of OPM schemas into relational definitions and procedures (OPM Schema Translator); this translator also generates a (metadata) mapping file containing information on the OPM-DBMS mapping;

3. a generic graphical querying, browsing and data entry tool with a WWW interface, for entering OPM data into an OPM-based database, specifying OPM queries and browsing their results (OPM Browsing & Querying); and

4. a tool for determining the OPM schema definition of existing relational database schemas (Rel2OPM); this tool is complemented by a graphical editor for adding missing referential (foreign-key) information to existing relational databases (Add_Ref).

The OPM data management tools can be used for both new and existing scientific database applications. For a new application, a domain expert can design an OPM schema using the OPM Schema Editor. This schema can be translated into relational database definitions and procedures using the OPM Schema Translator. The OPM Browsing & Querying tool can be used for entering, browsing, and querying data in the database.

For an existing scientific database application implemented with a relational DBMS, the relational to OPM schema conversion tool (Rel2OPM) can be used for generating an OPM view of the relational database. A domain expert can use the OPM Schema Editor for modifying this OPM view as needed. The OPM schema can then be translated into a relational database specification as described above. Alternatively, an OPM schema can be specified independently and then *retrofitted* on top of an existing relational database, by entering the correspondence between OPM and relational database constructs in the OPM-DBMS mapping file. Data in the existing relational database can be converted (i.e., restructured) according to the new database definition.

The main contribution of the work presented in this paper consists of (1) defining constructs for modeling experiments and incorporating these constructs into an object data model, and (2)

†http://gizmo.lbl.gov/opm.html

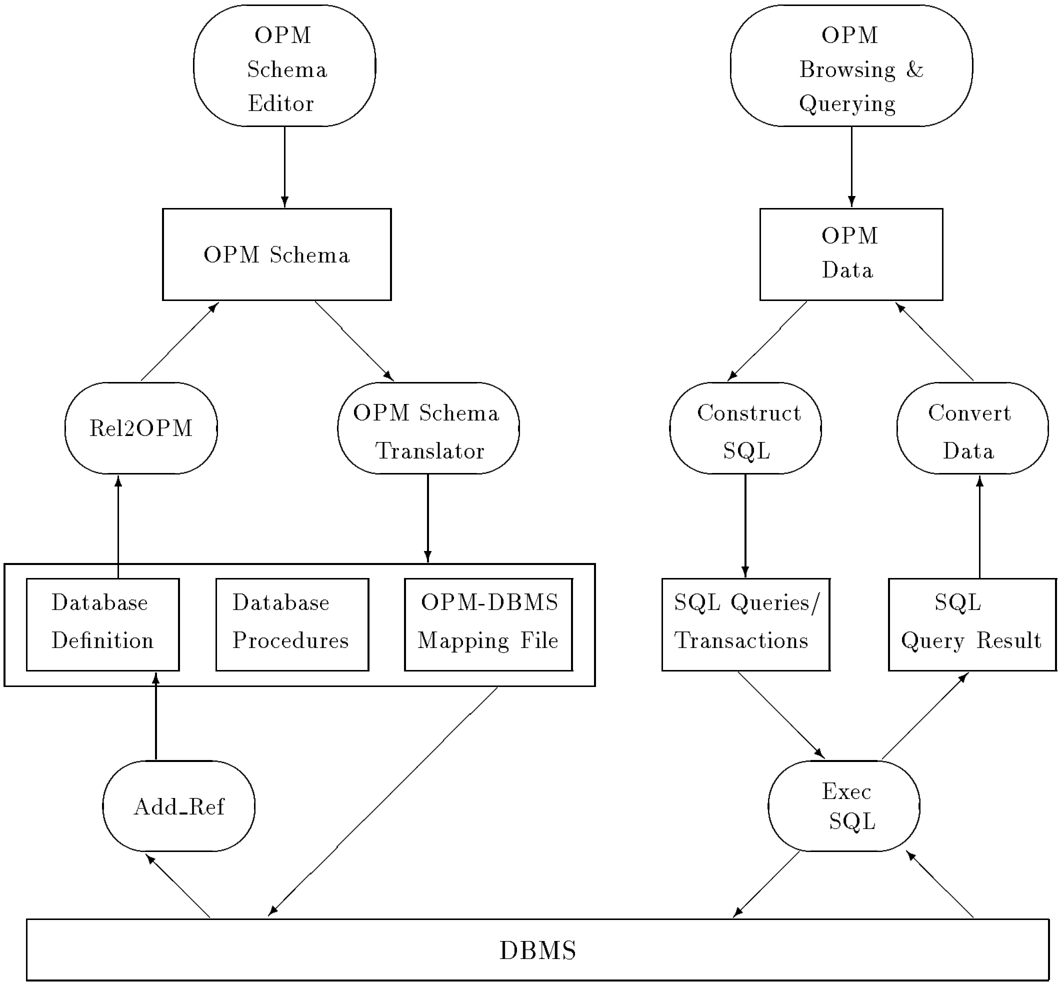


Fig. 1: The OPM Data Management Tools

developing data management tools that provide high-level (OPM) interfaces to commercial relational DBMSs, thus allowing users to interact efficiently with these DBMSs using clear and concise constructs and operations.

The rest of the paper is organized as follows. Requirements specific to scientific database applications and the motivation for developing OPM are discussed in section 2. OPM object and protocol classes are reviewed in sections 3 and 4, respectively. OPM views and queries are presented in section 5. The OPM data management tools are briefly described in section 6. In section 7, we discuss our experience in supporting scientific database applications. Section 8 reviews related work and contains concluding remarks.

2. MOTIVATION

We present in this section an example that illustrates the need for new constructs for modeling scientific experiments and for appropriate data management tools.

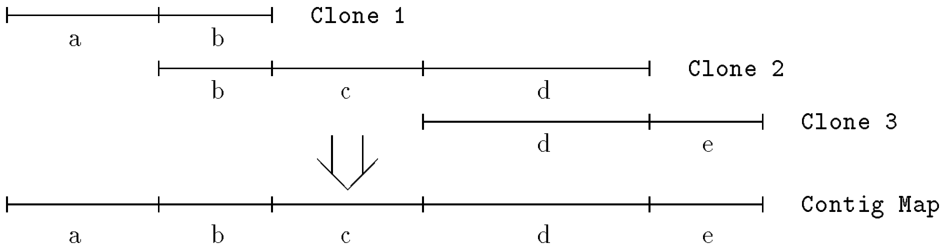
Experiments are instances of procedures such as genome sequencing protocols performed in molecular biology laboratories. Experiments are characterized by properties such as time or location. In addition, experiments

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;

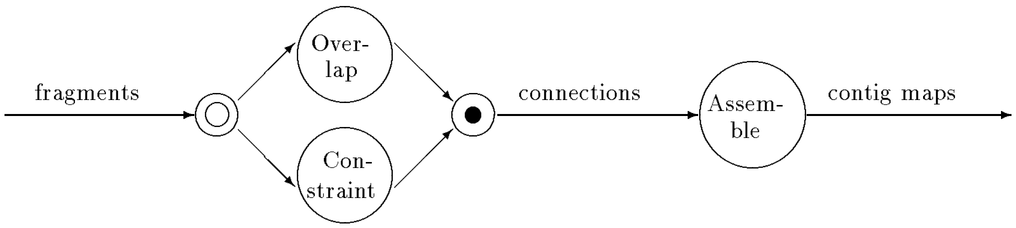
3. can be components of more generic experiments or can consist of sequences of component experiments and/or alternative component experiments, where an experiment and its component experiments may share common inputs and/or outputs.

Our example of an experiment regards the construction of *contig maps* to determine the complete DNA sequence of various organisms. Since existing technology permits sequencing only fragments of a few hundred nucleotides, chromosomal DNA is cut into smaller fragments, the fragments are propagated as clones, and then assembled into contig maps as shown in the following diagram, where `a`, `b`, `c`, `d`, and `e` represent DNA fragments:



Each contig map has properties such as (i) `contig_id` representing the unique contig map identifier; (ii) `owner` representing the person owning the contig map; (iii) `(entry, position)` representing the fragments with their positions in the contig map.

Contig maps of ordered DNA fragments can be constructed using a `Construct` protocol (experiment); such a protocol is applied on DNA `fragments` (input) and results in `contig maps` (output). This `Construct` protocol can be carried out by (expanded into) two alternative protocols, `Overlap` and `Constraint`, both followed by protocol `Assemble`, as shown below, where a notation similar to the Data Flow Diagram (DFD) notation [27] is used for representing protocols diagrammatically:



Protocol `Overlap` regards the comparison of two DNA fragments using a computer program, protocol `Constraint` regards the manual comparison of two DNA fragments according to certain constraints, and protocol `Assemble` regards the assembly of DNA fragments into a contig map according to information in the connection tables regarding possible connecting positions of two DNA fragments. The recursive specification of protocols in terms of sequence, alternative, or optional component subprotocols is called *protocol expansion*.

Different scientists often have different *views* of the same database application. Consider, for example, a database containing data only on protocols `Overlap`, `Constraint`, and `Assemble` mentioned above. Then, for scientists interested in (generic) protocol `Construct` mentioned above, rather than in the details of the (sub)protocols stored in the database, this generic protocol can be provided as a view.

In a scientific experiment environment, scientists are often interested in relationships among different objects and experiments (protocols). Typical queries have a nested structure and involve finding a *primary* set of objects and (not necessarily directly) related objects. For experiments, these queries regard finding the input and output or experiment steps of an experiment.

Relationships between various elements of a scientific application also involve complex 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. Requiring each `Assemble` protocol to be preceded by an `Overlap` or `Constraint` protocol is an example of such a rule. In addition, existence dependency rules involve specifying the effect of inserting, deleting or updating an element on related elements.

Constructs provided by object data models are not adequate for modeling accurately scientific experiments [8]. For example, the relationship between a generic experiment and its component experiments cannot be modeled as a class-subclass relationship. Generic protocol `Construct` cannot be modeled as a subclass of protocol `Overlap`, because `Construct` can also be carried out by subprotocol `Constraint`, and therefore some `Construct` instances are not even related to `Overlap` instances. Subprotocol `Overlap` cannot be modeled as a subclass of `Construct` either, because `Overlap` instances do not inherit the output (i.e., contig map) of `Construct`.

Modeling the relationship between a generic experiment and its component experiments using attributes is also inadequate. For example, suppose `Overlap`, `Constraint`, and `Assemble` are modeled as three attributes of `Construct`. Then it is not clear that `Construct` can be carried out by alternative protocols `Overlap` and `Constraint`, and then followed by another protocol `Assemble`. In this particular example, subprotocols of `Construct` can be modeled using a tuple attribute such as: `ATTRIBUTE steps(step1, step2): (Overlap or Constraint, Assemble)` if the underlying data model supports both tuple attributes and union value classes (such as `Overlap` or `Constraint`). However, such representations are not appropriate for more complex protocol expansions especially when some experimental steps can be optional. Moreover, such representations cannot capture the input and output relationships of experiments. For example, the input and output connections between an `Overlap` experiment and its corresponding `Assemble` experiment are not modeled by attribute `steps` shown above.

Constructs necessary for modeling scientific applications are not supported by general purpose semantic [14, 15] and object-oriented data models [3], or even data models developed specifically for experimental scientific database applications such as *MOOSE* [16, 33], *POSDBM* [26], or *ordering information model* [19]. Protocol connections and expansion can be simulated in object-oriented DBMSs using *methods*. However, using methods involves manually developing numerous procedures in a programming language such as C++ or Smalltalk. Such procedures are hard to comprehend and time-consuming to develop. Moreover, methods developed for a specific object-oriented DBMS cannot be ported to other DBMSs without substantial rewriting.

OPM has been designed primarily for supporting modeling scientific experiments in the context of an object model. In addition, OPM supports the specification of views, controlled vocabularies of scientific terms, and nested queries prevalent in scientific applications. Since OPM is intended for a community of scientists that currently use relational DBMSs and are likely to move to object-oriented DBMSs in the future, we are developing OPM-based tools targeting these DBMSs. For object-oriented DBMSs, these tools will automatically generate the DBMS methods implementing the OPM data retrieval, update and view construction methods, thus resolving the problems mentioned above regarding the manual development of DBMS methods.

3. MODELING OBJECT STRUCTURES IN OPM

OPM is the result of incorporating constructs for modeling scientific experiments into an object data model. The object part of OPM has similarities with other semantic and object data models, primarily with the *Semantic Database Model* (SDM) [14] and O_2 [3]. The object constructs of OPM are reviewed below.

3.1. Object Classes and Attributes

An OPM object class has a class name, an optional (text) class description, and is associated with one or several attributes. A subset of the attributes associated with an object class is specified as the *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) superclasses.

Attributes in OPM have attribute names, take values from value classes, and can be simple or consist of a tuple of simple attributes. An attribute can be single-valued, set-valued, or list-valued. A *simple* attribute is associated with either a single value class or a union of value classes. A *tuple* attribute (e.g., `(left_fragment, right_fragment)` of object class `Connection` in figure 2) has an optional name and consists of several *component* simple attributes, where each component attribute is associated with a single value class or a union of value classes. Depending on the type of its value class, an attribute can be primitive or abstract: a *primitive* attribute has as value class a *controlled* class of enumerated atomic values or ranges of values, or a class of atomic values of predefined data types; an *abstract* attribute has as value class an OPM class or union of classes (e.g., see attribute `owns` of object class `Person` in figure 2).

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

Definition 1 (Object Class Instances) Let O_i be an object class and x be an instance of O_i (denoted $x \in O_i$). Then $x = (oid(x), val(x))$, where $oid(x)$ is the globally unique 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, $1 \le j \le n$, 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 V, then $A_j(x)$ consists of one element, a set of elements, or a list of elements from the domain of V.

2. If A_j is a simple abstract attribute associated with value class: O_1 or ... or O_m, then $A_j(x)$ consists of one element, a set of elements, or a list of elements from $\{\ oid(y) \mid y \in O_k,$ $1 \le k \le m\ \}$.

3. If $A_j = (A_{j_1}, \ldots, A_{j_n})$, then $A_j(x)$ consists of a tuple, a set of tuples, or a list of tuples of the form $[A_{j_1} : A_{j_1}(x), \ldots, A_{j_n} : A_{j_n}(x)]$, where each component $A_{j_k}(x)$, $j_1 \le j_k \le j_n$, consists of one element of the form defined in 1 or 2 above.

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

Attributes are characterized by null and inverse constraints. *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 object class `Contig_Map` in figure 2) is defined as the *inverse* of abstract attribute B of class O_j (e.g., `owns` of object class `Person` in figure 2), then for every instance x of O_i and every instance y of O_j, $oid(y) \in A(x) \Rightarrow oid(x) \in B(y)$.

3.2. Delete Rules

Update, insert, and delete rules refer to the enforcement of constraints specified in an OPM schema, and are maintained during OPM data manipulation operations (see section 5.2). Unlike insert and update rules, delete rules can be explicitly specified in an OPM schema.

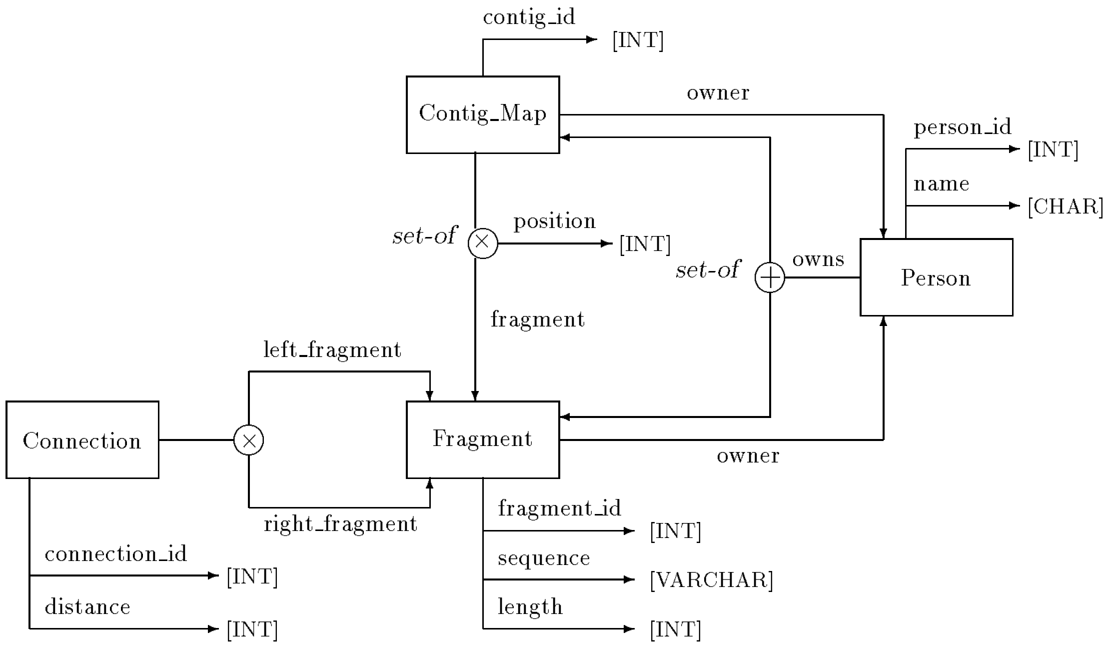
A delete rule is associated with an object class, O_i, and expresses the effect of deleting instances of O_i on other classes. Such a rule is specified in an OPM schema as a set of component rules associated with non-derived attributes whose value classes include O_i. Let attribute A of class O_j be such an attribute; the component delete rule for O_i with regard to A can be:†

1. *restricted*, meaning: if x is the unique A value for some instance y of O_j then x cannot be deleted from O_i;

2. *cascades*, meaning: if A has object x of O_i as one of its values for some instance y of O_j, then deleting x from O_i entails deleting y from O_j; or

3. *nullifies*, meaning: if A has object x of O_i as one of its values for some instance y of O_j, then deleting x from O_i entails removing x from the set/list of multiple A values for y, or nullifying the unique A value for y.

† These rules are the object-oriented analogs of the relational referential integrity rules (e.g. see [11]).



Notations:

- Labeled rectangles represent object classes.
- Arrows represent attributes, with the arrow pointing to the value class associated with the attribute.
- Circles containing a "+" denote unions of value classes.
- Circles containing a "×" denote tuple attributes.

Fig. 2: Diagrammatic Representation of the Object Classes Specified in the Appendix

For example, consider object classes `Person`, `Fragment`, and `Contig_Map` shown in figure 2. The delete rule for `Person` consists of the component delete rules regarding attributes `owner` of `Contig_Map` and `owner` of `Fragment`. If these two rules are *cascades* and *nullifies*, respectively, then if an instance x is deleted from `Person`, then all `Contig_Map` instances, y, having $oid(x)$ in `owner` (y) are also deleted, while the `owner` values of `Fragment` instances that contain $oid(x)$ are nullified.

3.3. Derived Attributes

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

A simple attribute A of a class O_i, can be defined as a *composition* of other attributes. A composition derivation consists of one path or a union of paths of the following form: B_1 $[O_{i_1}]$ B_2 $[O_{i_2}]$ $\ldots$ B_n $[O_{i_n}]$, where each O_{i_k} $(1 \leq k \leq n)$ denotes a class, and each B_k $(1 \leq k \leq n)$ denotes either (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, denoted $!C$, 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 O_{i_n} is the only class involved in the value class of attribute B_n.

For example, consider the object classes shown in figure 2; `left_fragment [Fragment] owner` and `left_fragment [Fragment] !fragment [Contig_Map]` are examples of paths starting with an attribute of class `Connection`, while `left_fragment [Fragment] owner or right_fragment`

†An example of an aggregate attribute associated with object class `Person` shown in figure 2 is attribute `owned_elements: count of owns`.

```
DERIVED OBJECT CLASS Long_Fragment
   DESCRIPTION: "fragments longer than 100 base pairs"
   DERIVATION: subclass of Fragment
   CONDITION: length > 100
   ATTRIBUTE owner_name       DERIVATION: owner.name
DERIVED OBJECT CLASS Contig_or_Fragment
   DESCRIPTION: "generalization of Contig_Map and Fragment"
   DERIVATION: Contig_Map or Fragment
   ID: derived_id
   ATTRIBUTE derived_id        DERIVATION: [Contig_Map] contig_id or
                                           [Fragment] fragment_id
   ATTRIBUTE owner             DERIVATION: [Contig_Map] owner [Person] or
                                           [Fragment] owner [Person]
DERIVED OBJECT CLASS Contig_Fragment
   DESCRIPTION: "pairs of Contig Maps and Fragments having common owners"
   DERIVATION: aggregate of Contig_Map, Fragment
   CONDITION: [Contig_Map] owner = [Fragment] owner
   ID: contig_id, fragment_id
   ATTRIBUTE contig_id         DERIVATION: [Contig_Map] contig_id
   ATTRIBUTE fragment_id       DERIVATION: [Fragment] fragment_id
   ATTRIBUTE owner             DERIVATION: [Fragment] owner [PERSON]
   ATTRIBUTE fragment_length  DERIVATION: [Fragment] length
```

Fig. 3: Derived Object Classes Defined Over the Object Classes of Fig. 2

`[Fragment] owner` is an example of a union of two paths.

The value of attribute A for an object 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 Values) Let A be an attribute of class O_i associated with an attribute composition derivation of the form mentioned above. For each instance x of O_i, $A(x)$ contains $oid(y)$ iff there exists a sequence of instances $x = x_0, x_1, \ldots, x_{n-1}, x_n = y$, s.t. $\forall$ $j = 1, \ldots, n$, then $x_j \in O_{i_j}$ and

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

For example, the following derived attribute can be associated with object class `Person` shown in figure 2: `contig_map DERIVATION: owns [Fragment] !fragment [Contig_Map]`. For an instance x of `Person`, `contig_map` consists of instances of `Contig_Map` whose `fragment` values contain `Fragment` instances owned by x.

3.4. Derived Object Classes

A derived object class in OPM 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. These constructs have similarities to analogous constructs defined in [1, 29, 30].

Derived classes can have their own derived attributes. Attribute composition involved in defining a derived class, O_i, is extended to: $[O_{j_0}]$ A_1 $[O_{j_1}]$ $\ldots$ A_n $[O_{j_n}]$, where O_{j_0} is an underlying class mentioned 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). O_s *inherits* all attributes from its superclasses O_1, ..., O_m. O_s can also have its own derived attributes. An example of a specialization derived object class is class `Long_Fragment` shown in figure 3.

A *generalization* derived object class, O_g, is defined as the superclass of object classes O_1, ..., 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, ..., and O_m. O_g can have explicitly defined derived attributes, and attributes from subclasses O_1, ..., O_m can be defined on O_g using attribute composition derivations. An example of a generalization derived object class is class `Contig_or_Fragment` shown in figure 3.

An *aggregate* object class, O_a, consists of objects that are aggregations of objects from other object classes, O_1, ..., 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. `Contig_Fragment` shown in figure 3 is an example of an aggregate derived object class.

4. MODELING EXPERIMENTS IN OPM

In this section we review the OPM constructs that allow modeling experimental procedures.

4.1. Protocol Classes

Similar to object classes, protocol classes have class names, optional class descriptions, identifiers, and are associated with attributes.

Protocol modeling is characterized by the recursive specification (*expansion*) of generic protocols in terms of component *alternative* subprotocols, *sequences* of subprotocols, and *optional* subprotocols; "*or*", ",", and "[ ]" are used to denote alternative, sequences of, and optional protocols respectively, and parentheses are used for specifying complex protocol compositions.

Definition 3 (Protocol Expansion) Protocol expansion expressions are defined as follows:

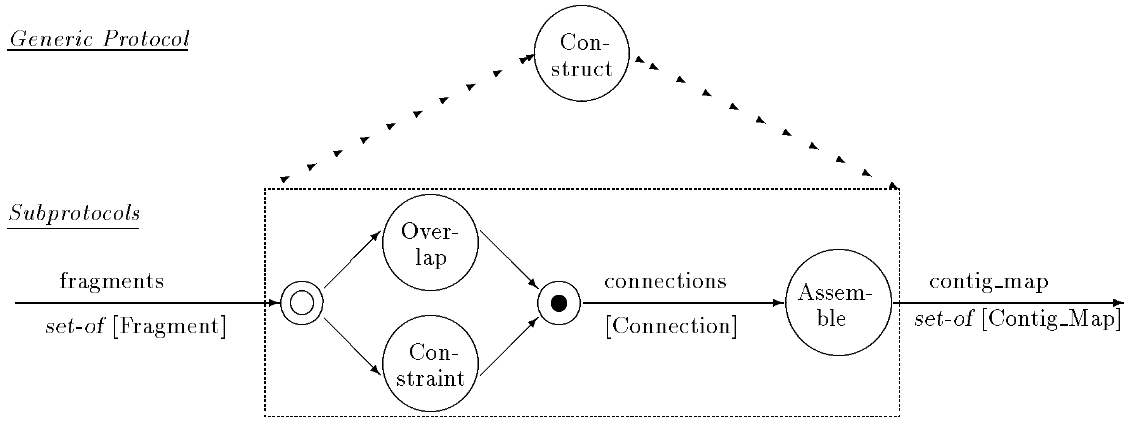
1. if P_1 and P_2 are protocol class names or protocol expansion expressions, then (i) P_1,P_2 (ii) P_1 or P_2 (iii) [P_1],P_2 (iv) P_1,[P_2] are protocol expansion expressions;
2. if P is a protocol expansion expression, then (P) is a protocol expansion expression.

For example, the expansion of class `Construct` shown in figure 4 is specified in OPM using the following expansion expression: `(Overlap or Constraint), Assemble`.

The structure of instances (protocols) in a protocol class is similar to the structure of object class instances, and is defined below.

Definition 4 (Protocol Class Instances) Let P_i be a protocol class and x be an instance of P_i (denoted $x \in P_i$). Then $x = (oid(x), gen(x), sub(x), pre(x), suc(x), val(x))$, where

1. $oid(x)$ and $val(x)$ are defined as in definition 1;
2. if P_j is specified as the generic protocol class of P_i and $gen(x) = oid(y)$, then $y \in P_j$, otherwise $gen(x)$ is not defined;
3. if P_1 or ... or P_m are specified as (sub)protocol classes of P_i, then $sub(x)$ consists of a list of elements from $\{ oid(y) \mid y \in P_k, 1 \leq k \leq m \}$, otherwise $sub(x)$ is not defined;
4. if P_1 or ... or P_m are specified as protocol classes that directly precede (i.e., provide input for) P_i, then $pre(x)$ consists of a set of elements from $\{ oid(y) \mid y \in P_k, 1 \leq k \leq m \}$, otherwise $pre(x)$ is not defined;
5. if P_1 or ... or P_m are specified as protocol classes that directly succeed (i.e., take input from) P_i, then $suc(x)$ consists of a set of elements from $\{ oid(y) \mid y \in P_k, 1 \leq k \leq m \}$, otherwise $suc(x)$ is not defined.



Notations:

- Labeled circles represent protocol classes.
- Arrows represent input-output attributes, where an arrow incident and directed to (resp. from) a circle representing a protocol class P represents an input (resp. output) attribute of P. Labels on arrows represent the names of the attributes and their associated value classes.
- Double-line circles denote common inputs shared by multiple protocols.
- Circles with shaded centers denote common outputs generated by different (alternative) protocols.

Fig. 4: Diagrammatic Representation of the Protocol Classes Specified in the Appendix

For each instance x of a protocol class, P_i, $gen(x)$, $sub(x)$, $pre(x)$, $suc(x)$ are expressed in OPM as values of special, system maintained *connection* attributes. These attributes are automatically generated during the implementation of an OPM schema (e.g., with a DBMS) based on the protocol expansion expressions in the schema. Accordingly, these attributes do not appear in OPM schema definitions. However, users can define delete rules for these attributes in an OPM schema (see section 4.3), or use these attributes in queries (see section 5).

A protocol class, P_i, is associated with the following connection attributes:

1. if P_i is specified as a subprotocol of a generic protocol class, then P_i is associated with connection attribute `GenProt`; for each instance x of P_i, $gen(x) = \texttt{GenProt}(x)$;
2. if P_i is specified as a generic protocol class of other (sub) protocol classes, then P_i is associated with connection attribute `SubProt`; for each instance x of P_i, $sub(x) = \texttt{SubProt}(x)$;
3. if P_i is specified as a protocol class that takes input from other protocol classes, then P_i is associated with connection attribute `PreProt`; for each instance x of P_i, $pre(x) = \texttt{PreProt}(x)$;
4. if P_i is specified as a protocol class that provides input for other protocol classes, then P_i is associated with connection attribute `SucProt`; for each instance x of P_i, $suc(x) = \texttt{SucProt}(x)$.

An experiment can be a component of at most one generic experiment, therefore attribute `GenProt` is single-valued; attribute `SubProt` is list-valued; attributes `PreProt` and `SucProt` are set-valued. The value class associated with connection attributes are implied by a given protocol class structure as follows: the value class of `GenProt` is the generic protocol class of P_i; the value class of `SubProt` is the union of the component (sub) protocol classes of P_i; the value class of `PreProt` is the union of the protocol classes directly preceding P_i; the value class of `SucProt` is the union of the protocol classes directly succeeding P_i.

Note that attributes `GenProt` and `SucProt` can be expressed as derived attribute compositions as follows:

1. `GenProt`: `!SubProt` $[P_j]$, where P_j is specified as the generic protocol class of P_i;
2. `SucProt`: `!PreProt` $[P_{j_1}]$ or ... or `!PreProt` $[P_{j_m}]$, where for each $1 \leq k \leq m$, P_{j_k} is specified as a protocol class succeeding P_i.

4.2. Input and Output Attributes

In addition to regular and connection attributes, a protocol class can be associated with special input and output attributes. Input and output attributes associated with a protocol class represent input and output data regarding the experiment modeled by the protocol class. 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 several subprotocols, then its input and output attributes are inherited by some of its subprotocols. This input-output attribute inheritance is expressed in OPM using '`input isa` ...' statements and '`output isa` ...' statements in the specification of the input and output attributes associated with subprotocols. For example, the definitions of attributes `fragments` of `Overlap` and `contig_map` of `Assemble` of the OPM schema shown in the appendix contain '`input isa` ...' and '`output isa` ...' statements, respectively.

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. Such input-output protocol connections are expressed in OPM using '`input from` ... `via` ...' statements 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. For example, the definition of attribute `connections` of `Assemble` of the OPM schema shown in the appendix contains an '`input from` ... `via` ...' statement.

Note that expansion expressions in an OPM schema are not sufficient for establishing unambiguously the relationships between input and output attributes. For example, a protocol class can inherit only some (not necessarily all) input or output attributes of its generic protocol class. Similarly, a protocol class, P_i can take as input only part of the output generated by other protocol classes (preceding P_i).

Input and output attributes can be perceived as regular attributes that must satisfy additional *input-output consistency constraints*. Thus, for each pair of protocol classes P_i and P_j, (i) 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; (ii) 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.

Two derived attributes, `InputFor` and `OutputOf`, are maintained by the system 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, as shown below.

For example, since object class `Contig_Map` contains instances of the output of `Assemble` protocol and its generic protocol `Construct`, an attribute `OutputOf` is automatically generated for class `Contig_Map`, where this attribute is associated with value class `Construct` or `Assemble`, and has the following derivation: `!contig_map [Construct]` or `!contig_map [Assemble]`.

4.3. Using Delete Rules for Expressing Protocol Relationships

Defining experiment relationships includes specifying for each protocol class, P_i, the delete rules associated with the connection and system attributes involving P_i (i.e., whose value classes include P_i). Thus, the component delete rule of a protocol class P_i with regard to an `InputFor` (or `OutputOf`) attribute, A, associated with object class O_j, can be:

1. *nullifies*, meaning that the deletion of a protocol of P_i having an object x of O_j as input (or output) entails nullifying the value of A for x (i.e., has no effect on O_j); or

2. *cascades*, meaning that the deletion of a protocol of P_i having an object x of O_j as input (or output) entails deleting x from O_j.

```
DERIVED PROTOCOL CLASS Overlap_Assemble
   DESCRIPTION: "generic protocols representing Overlap followed by Assemble"
   DERIVATION: Overlap, Assemble
   CONDITION: [Assemble] connections in [Overlap] connections
   ID: overlap_id, assemble_id
   ATTRIBUTE overlap_id                DERIVATION: [Overlap] overlap_id
   ATTRIBUTE assemble_id               DERIVATION: [Assemble] assemble_id
   ATTRIBUTE fragments      input      DERIVATION: [Overlap] fragments [Fragment]
   ATTRIBUTE contigs        output     DERIVATION: [Assemble] contig_map [CONTIG_MAP]
DERIVED PROTOCOL CLASS Insert_Comparison
   DESCRIPTION: "compare fragments using the insert program"
   DERIVATION: subprotocol of Overlap
   CONDITION: name = "insert"
```

Fig. 5: Derived Protocol Classes Defined Over the Protocol Classes of Fig. 4

Similarly, the component delete rule of a protocol class P_i with regard to a GenProt (or PreProt, SubProt, SucProt) attribute, A, associated with protocol class P_j, can be:

1. *restricted*, meaning that a protocol x of P_i cannot be deleted before deleting the P_j protocol related to x;

2. *cascades*, meaning that the deletion of a protocol x of P_i entails deleting the P_j protocol related to x; or

3. *nullifies*, meaning that the deletion of a protocol x of P_i entails nullifying the A value for the P_j protocols related to x.

Since connection and system attributes are not explicitly defined in an OPM schema, delete rules for these attributes are specified using statements of the form INPUT_FOR delete-rule, OUTPUT_OF delete-rule, GENERIC_PROTOCOL delete-rule, SUBPROTOCOLS delete-rule, PREDECESSOR_PROTOCOLS delete-rule, and SUCCESSOR_PROTOCOLS delete-rule, respectively. Consider the OPM schema shown in the appendix. The delete rule associated with protocol class Constraint involves the following component delete rules:

1. Rule OUTPUT_OF DELETE CASCADES specified for object class Connection means that deleting a Constraint protocol will delete the corresponding output Connection objects;

2. Rule SUBPROTOCOLS DELETE NULLIFIES specified for protocol class Construct means that deleting a Constraint protocol will not affect the corresponding Construct protocol;

3. Rule PREDECESSOR_PROTOCOL DELETE RESTRICTED specified for protocol class Assemble means that a Constraint protocol cannot be deleted unless there are no Assemble protocols related to (i.e., succeeding) it.

4.4. Derived Protocol Classes

OPM supports two derived protocol class constructs: (i) derived generic protocol classes can be used for representing experiments that are constructed from experiments represented as instances of existing protocol classes, and (ii) derived subprotocol classes can be used for representing subsets of experiments satisfying a given condition or a component of an existing experiment.

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.

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. An example of a derived generic protocol class is class `Overlap_Assemble` in figure 5 consisting of protocols that are defined as compositions of `Overlap` and `Assemble` protocols.

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. A subprotocol, P_s, inherits attributes from its generic protocol, P_i, unless attributes are redefined in order to override the definition of attributes of P_i.

The definition of a derived subprotocol class is similar to the derivation of specialization derived object class. A significant difference between derived subprotocol classes and derived specialization classes is the capability of redefining regular attributes of underlying protocols as input/output attributes of derived subprotocol classes. Class `Insert_Comparison` in figure 5 is an example of a derived subprotocol class.

5. OPM QUERIES AND UPDATES

OPM provides constructs for expressing queries that are prevalent in scientific applications. The OPM query language is briefly described below. A detailed definition is provided in [6].

5.1. Queries

The query languages supported by many object-oriented DBMSs, such as O_2 [3] and *Ode* [10], follow the structure of relational SQL. These versions of SQL also provide object-oriented specific query constructs, such as path expressions and method invocation [4].

The OPM query language follows the same SQL-like structure and supports path expressions and set predicates. Other object-oriented specific query constructs have not been included in the current version of OPM for the sake of simplicity and in order to avoid the complexity of implementing them on top of relational DBMSs.

An important property of query languages is the *closure* of the language, that is, the capability of using the result of a query in another query. For object-oriented query languages the closure property requires the ability of defining the query result as a (derived) class and finding its place in the class hierarchy [4]. While the result of single target class queries (i.e., queries with a single class name in the `FROM` statement) can be defined as a derived subclass of the target class, the type of the result of a multi-target query is harder to infer. We have avoided this problem by restricting OPM queries to single target classes, while this limitation can be offset by using derived classes when necessary, as shown later in this section.

An OPM query on target class O_i can involve local, inherited, derived and system attributes associated with O_i, and path expressions starting with these attributes. Consider, the OPM object and protocol classes shown in figures 2 and 4. The following OPM query expresses the selection of fragments owned by John, together with the protocols using these fragments as input:

```
SELECT fragment_id, InputFor
FROM   Fragment
WHERE  owner.name match "John";
```

Note that system attribute `InputFor` refers to protocols `Construct`, `Overlap`, and `Constraint` that may have fragments as their input. The employment of connection attributes in OPM queries is illustrated in the following OPM query regarding the selection of contig maps that are generated by assemble experiments that follow only (i.e., take input from) overlap experiments and that are owned by Bill:

```
SELECT contig_map
FROM   Assemble
WHERE  PreProt in Overlap and contig_map.owner.name contains "Bill";
```

Although each OPM query is associated with a single target class, this limitation can be offset using abstract attributes (referencing other object or protocol classes) in `SELECT` and `WHERE` statements. Furthermore, multi-target class queries can be constructed using derived classes and derived attributes. Suppose one wants to find the pairs of contig maps and fragments that are longer than 120 base pairs, where contig maps and fragments have common owners. This query can be expressed in two steps: first, aggregate derived object class `Contig_Fragment` is defined as shown in figure 3; then the following query is expressed over `Contig_Fragment`:

```
SELECT contig_id, fragment_id
FROM   Contig_Fragment
WHERE  fragment_length > 120;
```

A method frequently employed for exploring scientific data involves (1) selecting subsets of instances from different classes; and (2) examining (browsing) related instances in these subsets. In OPM, this type of data exploration can be expressed by:

1. defining derived subclasses and derived attributes for these subclasses; and
2. examining related instances in these derived subclasses related via derived attributes.

For example, consider the OPM object and protocol classes shown in figures 2 and 4. In order to examine overlap protocols that are using `insert` program version 3.0, and their related fragments that are longer than 100 base pairs, two derived subclasses need to be specified: (i) derived subclass `Long_Fragment` shown in figure 3; and (ii) a derived subprotocol of `Overlap` defined as follows:

```
DERIVED PROTOCOL CLASS Insert_Overlap
  DESCRIPTION: "overlap protocols using insert program version 3.0"
  DERIVATION: subprotocol of Overlap
  CONDITION: name = "insert" and version = "3.0"
  ATTRIBUTE long_fragments DERIVATION: fragments [Long_Fragment]
  ATTRIBUTE sequence       DERIVATION: long_fragments [Long_Fragment] sequence
```

where attribute `long_fragments` is defined by restricting the range of inherited attribute `fragments` to the instances of `Long_Fragment`. Subsequently, one can examine (browse) instances of subclass `Insert_Overlap` where for each such instance, x, fragments longer than 100 base pairs that are related to x can also be browsed.

The OPM query language provides a nested query construct that allows retrieving together related instances of subclasses of the form described above. Such a nested query involves (object or protocol) classes, O_0, ..., O_n, where O_0 is called the *primary* target class of the nested query and for each $0 \leq i \leq n$:

1. O_i can be restricted using a (`WHERE`) condition;
2. existing attributes as well as new derived attributes of O_i can be selected for retrieval;
3. for every abstract attribute of O_i, A, selected for retrieval, steps (1) and (2) above can be repeated recursively, so that for a class, O_{i+1} ($i \leq n-1$), involved in the value class of A, O_{i+1} can be restricted using a condition and attributes of O_{i+1} can be selected for retrieval.

The result of a nested query consists of defining subclasses O'_i of classes O_i, ($0 \leq i \leq n$) so that:

1. O'_0 consists of instances of O_0 that satisfy the condition associated with O_0 in the query;
2. for every pair of classes O_j and O_k, $0 \leq j, k \leq n$, where attribute A of O_j has been selected for retrieval and O_k is involved in the value class of A:

    (a) a derived attribute A' is associated with O'_j, so that the range of A' is restricted to O'_k;

(b) O'_k contains instances of O_k, so that each such instance, x, satisfies the condition associated with O_k in the query and there exists an instance y of O'_j so that $A(y)$ contains $oid(x)$.

For example, the following nested OPM query can be used as an alternative way of exploring overlap protocols that are using `insert` program version 3.0, and their related fragments longer than 100 base pairs:

```
SELECT overlap_id,
       fragments:(SELECT sequence,  owner.name
                  FROM Fragment
                  WHERE length > 100)
FROM   Overlap
WHERE  name = "insert" and version = "3.0";
```

Note that all instances of protocol class `Overlap` generated by insert program version 3.0 are included in the result of this query regardless of fragment lengths; however, the query result contains only fragments that are both longer than 100 base pairs and referenced by an `Overlap` instance included in the query result.

5.2. Updates

OPM provides update constructs that are similar to analogous constructs in other query languages. OPM update constructs consist of `INSERT`, `DELETE` or `UPDATE` statements.

An `INSERT` statement involves the values of all local and inherited attributes associated with a target object or protocol class, where abstract attribute values are represented by external identifier values used for identifying object or protocol class instances, together with the names of the corresponding classes. For example, the following statement expresses the insertion of a person who owns a contig map and a fragment:

```
INSERT Person: person_id = 20, name = "Mary",
               owns = { (Contig_Map: contig_id = 11),
                        (Fragment: fragment_id = 19) }
```

Conditions are associated with an `INSERT` statements only for identifying instances of an existing object or protocol class, O_i, upon insertion into subclasses of O_i.

A `DELETE` statement is associated with a target object or protocol class, and can involve a condition for the target class. Finally, an `UPDATE` statement involves local and inherited attributes associated with a target object or protocol class, and can also involve a condition for the target class.

Carrying out updates entails enforcing the appropriate insert, delete, or update rules. Update rules are associated with each non-derived attribute, A, and define the enforcement of null, referential integrity, inversion, and input-output consistency constraints when the values of A are updated.

Insert rules are associated with each class, O_i and define the enforcement of attribute constraints for each non-derived attribute of O_i. Moreover, since instances of O_i must also belong to the superclasses of O_i, an instance that is inserted into O_i is also automatically inserted into (propagated to) all the superclasses of O_i.

As mentioned in sections 3 and 4, non-derived attributes, connection attributes and system attributes can be associated with delete rules that define the enforcement of referential integrity constraints by specifying the effect of deleting an instance of a class, O_i, on related classes. Furthermore, when an instance x is deleted from a class O_i, x is automatically removed from all the subclasses of O_i. Additional delete rules define the enforcement of null, inversion, and input-output consistency constraints upon deleting object or protocol instances.

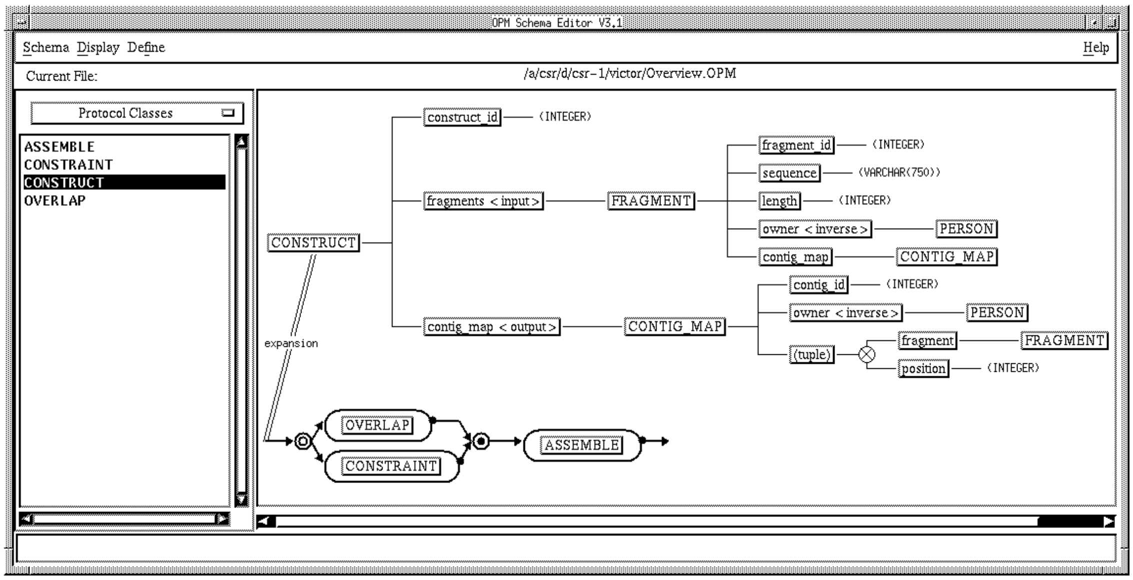


Fig. 6: Example of Browsing OPM Classes with the OPM Schema Editor

6. OPM DATA MANAGEMENT TOOLS

We briefly describe in this section the following OPM data management tools: the (DBMS-independent) OPM Schema Editor; the OPM Schema Translator and OPM Browsing and Querying tool that target relational DBMSs; a reverse engineering tool that allows examining existing relational database schemas and determining its corresponding OPM schema definition. These tools have been implemented using C++, yacc++, and lex++.

6.1. OPM Schema Editor

OPM schemas can be specified using either a regular text editor or using the graphical OPM Schema Editor.

The OPM Schema Editor allows specifying interactively and incrementally object and protocol structures by providing facilities for defining new OPM classes, modifying existing OPM classes, and combining (merging) classes from different OPM schemas. The OPM Schema Editor provides facilities similar to other schema editors [3, 28], such as browsing, specifying, and modifying definitions of object and protocol classes. For example, the schema browsing facility of the editor is illustrated in figure 6, where protocol class `CONSTRUCT` is displayed together with its attributes and its (subprotocol) expansion, and the value classes of the abstract attributes of `CONSTRUCT` are expanded.

OPM schemas are kept in ASCII files that can be passed to the OPM Schema Translator described below, in order to generate the corresponding DBMS schemas and queries. The OPM Schema Editor also provides facilities for generating Postcript diagrams, Latex documents and HTML files representing OPM schemas, where the latter allows exploring OPM schemas via WWW. †

We plan to develop an OPM View Editor that will be similar to the OPM Schema Editor, and will allow defining new derived object classes, protocol classes, attributes, and hiding or renaming classes and attributes in an OPM schema or view.

†Examples OPM schemas that can be browsed via WWW at http://gizmo.lbl.gov/opm.html

6.2. OPM Tools for Relational Database Management Systems

Since numerous scientists currently use commercial relational DBMSs such as Sybase, we have developed OPM data management tools targeting relational DBMSs. Developing such tools entails translating OPM schemas, (retrieval and update) methods, and queries into relational DBMS specifications. In a relational database, data on instances of an OPM (object or protocol) class can be kept in one or several normalized relations,[‡] 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 OPM object or protocol scattered among multiple relations. Conversely, using unnormalized relations has the advantage of having data on one OPM 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 OPM classes and that also include the values corresponding to the OPM derived attributes. The DBMS specifications for constructing these views represent the OPM retrieval methods and may involve large and multiple SQL queries. Thus, if data regarding the instances of an OPM class are kept in one normalized *primary* relation and several normalized *auxiliary* relations,[†] the relational view corresponding to this class involves recursively *left-outerjoining* the primary relation with the auxiliary relations. The relational representation for an OPM schema must also include data integrity constraint specifications (e.g., referential integrity and domain constraints) expressing the constraints implied by the OPM constructs. For relational DBMSs that have procedural integrity mechanisms (e.g., the trigger mechanism in Sybase, the rule mechanism in Ingres) these specifications are very large and complex. The complexity problems regarding the relational representation of OPM classes mentioned above are compounded by the diversity of integrity mechanisms and SQL dialects supported by various relational DBMS products.

The OPM tools address the problems mentioned above while insulating users from the underlying relational DBMS. Thus, the OPM Schema Translator described below uses normalized relations for representing OPM classes, and generates procedures implementing the OPM retrieval and update methods that can be used for constructing the relational views mentioned above.

6.2.1. OPM Schema Translator

The OPM Schema Translator automatically translates OPM schemas into relational DBMS database definitions and DBMS procedures implementing the OPM (retrieval and update) methods.

Informally, mapping an OPM schema into a relational DBMS definition consists of mapping every OPM object or protocol class O_i into a *primary* relation R_i; depending on their type (primitive, abstract, simple, tuple, etc.), non-derived attributes of O_i are mapped into local attributes of R_i, foreign-key attributes of R_i, and additional *auxiliary* relations with appropriate foreign-key to primary-key references.

User-defined attributes, system attributes and protocol connection attributes are all mapped using the same strategy. For example, suppose that a protocol class, P_i, is associated with protocol connection attribute `PreProt` whose value class is P_j (see section 4.1). `PreProt` is a set-valued attribute and therefore is represented using an auxiliary relation involving a relational foreign-key attribute referencing the primary relation representing P_i, and another foreign-key attribute referencing the primary relation representing P_j.

Mapping a class 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

[‡]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.

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

O_i instances from R_i tuples together with related tuples representing the attribute values of O_i. Such a query involves left-outerjoining 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 tuples into or deleting tuples from 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.

The OPM Schema Translator also generates a (metadata) mapping file containing information on the OPM-DBMS mapping.

The mapping of OPM schemas into relational database definitions and queries is defined in [5]. The OPM Schema Translator implementing this mapping is described in [7].

6.2.2. OPM Browsing and Querying Tool

The OPM Browsing and Querying tool [31] allows browsing OPM schemas and browsing, querying, and manipulating (inserting, deleting, updating) OPM object and protocol instances in an OPM-based relational database.

The OPM Browsing and Querying tool has a WWW (currently Mosaic) interface (see figure 7). Accordingly, OPM metadata (schemas) and data (class instances) are represented and browsed in HTML format. Figure 7 contains several windows that illustrate schema and data browsing. Schema browsing starts with the list of all OPM classes (e.g., see the top-right window in figure 7), displays the list of attributes for a selected class, and can recursively display the definition of OPM classes that are value classes of selected abstract attributes.

Instances in an OPM class (e.g., class `FRAGMENT` in figure 7), can be browsed by first listing all instances in that class (e.g., see the top-left window in figure 7); subsequently, each selected instance in this list is displayed with all its attribute values (e.g., see the middle window in figure 7), so that abstract attribute values can be selected (expanded) for displaying related objects (e.g., see the bottom-right window in figure 7 containing a `CONTIG_MAP` instance related to the `FRAGMENT` instance displayed in the middle window of the same figure).

The OPM Browsing and Querying tool allows inserting, deleting or modifying instances in an OPM-based database. OPM data manipulations are translated by the OPM Browsing and Querying tool into SQL insert, delete or update statements for the underlying relational DBMS as follows:

1. OPM data manipulation statements are parsed and auxiliary (side-effect) data manipulations entailed by the OPM rules are automatically generated; auxiliary data manipulations include propagating insertions to superclasses, updating inverse attributes, and propagating deletions according to the deletion rules.

2. primary and auxiliary data manipulation statements are transformed so that external identifiers mentioned in these statements are replaced by the corresponding object identifiers (surrogate keys); new object identifiers are generated for new object or protocol instances;

3. attribute and referential integrity constraints are checked, and users are notified of errors and prompted to correct them;

4. SQL data manipulation transactions are generated and executed.

Finally, the OPM Browsing and Querying tool allows specifying OPM queries and browsing the results of these queries. The translation of OPM queries into SQL is carried out by an OPM Query Translator that uses the information in the mapping file generated by the OPM Schema Translator, and then further processes results returned by the relational DBMS by converting flat relational query results into structured OPM objects or protocols. Expressing certain types of OPM query conditions, such as conditions involving set-valued attributes, in the SQL dialect of commercial DBMSs such as Sybase, is so complex that we have decided to bypass these conditions

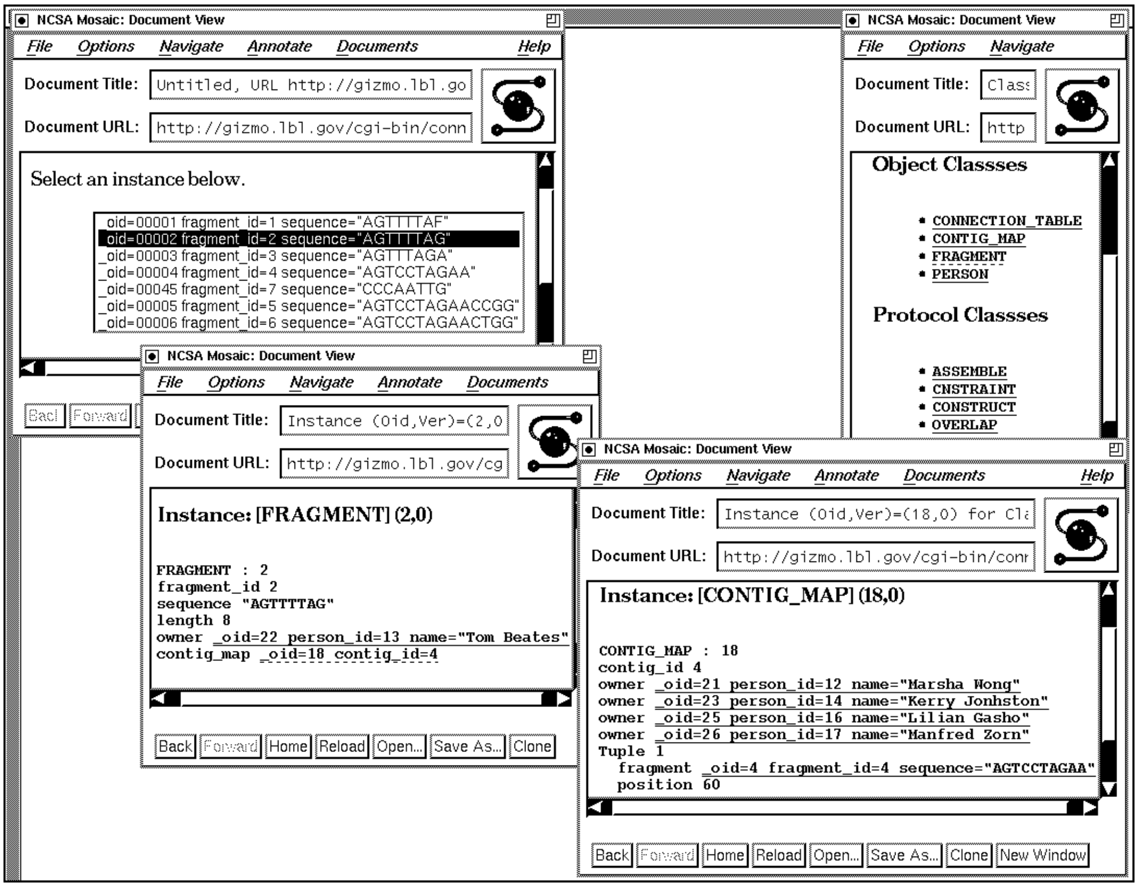


Fig. 7: Browsing OPM Class Instances using the OPM Browsing and Querying Tool

during the query translation. Values of the attributes involved in these conditions are retrieved using SQL queries and subsequently processed by the OPM Query Translator.

The OPM Browsing and Querying tool can be examined on two sample databases via World Wide Web.†

6.3. Relational to OPM Schema Conversion Tool

The relational to OPM schema conversion tool, Rel2OPM, can be applied to existing relational DBMS database definitions. Every relation definition is mapped into an OPM class and every relational attribute is mapped into an OPM attribute. Next, if the relational database definition contains foreign-key (referential integrity) information, then (i) the type of abstract attributes (e.g., single or set valued, inverse) is automatically derived by examining the structure of primary and foreign-keys in each relation; and (ii) for each abstract attribute, foreign-key attributes are removed from the class definition. Finally, the resulting OPM schema can be refined using the OPM Schema Editor by changing, if needed, the type of some attributes (e.g., by grouping simple attributes to form a tuple attribute).

For example, consider the abstract relational schema example shown in figure 8. This schema is converted by Rel2OPM into an OPM schema consisting of classes `Person`, `Author`, and `Citation`, where `Author` is a subclass (specialization) of `Person`, `Person` has primitive attributes `person_id`

†http://gizmo.lbl.gov/opm.html

```
RELATION Person   ( person_id int NO NULLS
                    name  char(100) NULLS ALLOWED
                    PRIMARY KEY (person_id) )
RELATION Author   ( author_id int NO NULLS
                    PRIMARY KEY (author_id)
                    FOREIGN KEY (author_id) REFERENCES Person )
RELATION Citation ( citation_id int NO NULLS
                    title char(50) NO NULLS
                    PRIMARY KEY (citation_id) )
RELATION Citation_Author ( author_id int NO NULLS
                           citation_id int NO NULLS
                           PRIMARY KEY (author_id, citation_id)
                           FOREIGN KEY (author_id) REFERENCES Author
                           FOREIGN KEY (citation_id) REFERENCES Citation )
```

Fig. 8: A Relational Schema Example

and `name`, `Author` has abstract attribute `citation` with value class `Citation`, and `Citation` has primitive attributes `citation_id` and `title`, and abstract attribute `author` with value class `Author`, where `author` is the inverse of `citation`.

Foreign-key information that is missing in the database definition can be filled in using a graphical editor called Add_Ref. Subsequently, Rel2OPM can be used for automatically generating procedures for testing the consistency of the underlying database with respect to the specified foreign-keys.

The conversion procedure mentioned above works well for normalized relational databases that have been originally designed using an object (e.g. Entity-Relationship) design methodology. However even for such databases it is often hard to determine algorithmically what relations represent classes of objects and what relations represent attributes (e.g., set-valued attributes) that cannot be represented as attributes in the normalized relations representing classes. For such databases, one can follow a more flexible approach by specifying OPM schemas independently and then *retrofitting* these schemas on top of existing relational databases, by entering the correspondence (mapping) between OPM and relational database constructs in the OPM-DBMS mapping file. We have carried out such a retrofitting for the Genome Sequence Data Base (GSDB) at the National Center for Genome Resources, Santa Fe.

7. DEVELOPING SCIENTIFIC DATABASE APPLICATIONS WITH OPM

The development of OPM and OPM data management tools started in the summer of 1992. OPM was developed originally for modeling molecular biology experiments. The protocol class construct was developed by interacting directly with scientists in the Division of Biology of the Science and Technology Center at Caltech, Pasadena. Since we aimed at applying OPM to a large variety of scientific as well as traditional database applications, we have incorporated the protocol class construct into an object data model.

The OPM data management tools have been first used for developing a prototype database for the large scale genome sequencing project at Caltech [9]. Subsequently, OPM and the OPM data management tools have been adopted for developing version 6 of Genome Data Base (GDB) at Johns Hopkins School of Medicine, Baltimore, and the new Macromolecular Structure Database (MSD) of the enhanced Protein Data Bank (PDB) at the Brookhaven National Laboratory. Recently, additional groups, developing large scientific databases at the National Center for Genome Resources, Santa Fe, at the Chromosome 22 Center, University of Pennsylvania, Philadelphia, and Myriad Genetics, Inc, Salt Lake City, also started to use OPM and the OPM data management tools.

In practice, large scientific databases, such as GDB and MSD, are developed using proven commercial relational DBMSs. The structure of these databases is defined by database administrators in terms of relational tables based on documents describing application-specific semantics. Scientists do not understand relational database definitions and therefore cannot verify whether these definitions comply with the semantics of their applications.

In addition to traditional capabilities provided by DBMSs, managing scientific data requires mechanisms for presenting, browsing, and querying data in a way that scientists can understand. These mechanisms are provided by specialized programs that usually support certain (limited) ways of manipulating data and are based on predefined (canned) SQL queries. Again, scientists do not understand SQL and therefore cannot verify that the SQL queries express the intended data manipulations. Furthermore, they are not provided with ad-hoc high-level query capabilities, and therefore are restricted to a limited set of data manipulations.

Finally, scientific applications evolve in time. This evolution entails modifying the structure of the underlying databases and programs. The gap between scientific applications and their underlying databases entails lengthy database restructuring and program rewriting even for minor application changes.

OPM provides scientific database groups with high-level DBMS-independent languages for specifying the structure of and manipulating data in databases. Although using OPM still requires learning, OPM class definitions and queries are concise and clear and therefore can be used for communicating with scientists and involving them in the database development process. Thus, scientists participated directly in developing several OPM schemas (e.g., for MSD) and/or in discussing the OPM representation of their database applications. OPM provided the language (and WWW provided the technology) for involving interested biologists worldwide in the development of scientific databases such as GDB and MSD.

The OPM tools provide facilities for developing and maintaining efficiently large databases on top of existing commercial relational DBMSs. The databases developed with the OPM tools are documented using the OPM Schema Editor and are tightly coupled to the interfaces used by scientists via self-explanatory ASCII metadata files. The OPM tools insulate both database developers and users from the underlying DBMSs, and therefore facilitate transfer to other (e.g., object-oriented) DBMSs in future. Furthermore, users of OPM tools are reporting that the automatic generation of database definitions and procedures has saved them substantial amount of coding and debugging time.

Interactions with our scientific collaborators revealed the need for additional features that were important for developing scientific database applications. These features are briefly summarized below; some of them are already part of OPM while others will be incorporated into OPM in future.

1. Schema and data representation and browsing. Graphical representations of object and protocol class definitions, class hierarchies and protocol expansions were suggested by scientists in order to facilitate understanding OPM schemas. The capability of displaying protocols and subprotocols at various levels of expansion was considered especially helpful. Since scientists are commonly using WWW for exploring scientific data, the OPM tools have been extended with facilities for representing and browsing OPM schemas and data in HTML format.

2. Controlled vocabularies and documentation. Vocabularies consisting of finite sets of existing application-specific terms (such as medical terms, loci names), are prevalent in scientific applications. These terms are often codified (e.g., for resolving name conflicts or allowing synonyms) and need to be associated with detailed descriptions. Similarly, detailed descriptions need to be associated with classes and attributes in order to make database schemas self-descriptive. OPM has been extended in order to support controlled vocabularies and descriptions of OPM schema elements.

3. Data consistency. An important function of a database system is to ensure the consistency and integrity of data. However, data in scientific databases are usually incomplete or inconsistent due to missing information or human errors. Strictly enforcing integrity constraints on these databases could be too restrictive since it could prevent recording important partial information. Completely abandoning integrity constraint enforcement as done in scientific database applications such as [13] could lead to hard to detect data inconsistencies. OPM

provides constraints and update rules that allow users to control the desired level of integrity enforcement.

4. Versioning. Versions are essential for modeling alternative representations and keeping track of the evolution of scientific (e.g., experimental, analysis) observations. In response to requirements of the genomic application underlying GDB and in collaboration with members of the GDB staff, OPM has been extended with data versioning constructs and the OPM tools have been extended with version management facilities.

5. Multidatabases. Scientific data are currently scattered among multiple databases. Comprehensive studies of scientific data often involves examining data across several (heterogeneous) databases. Retrieving data from several databases is based on establishing and/or using relationships (links) between data in these databases and resolving differences between data shared by these databases. OPM has been extended with a *foreign class* construct that allows cross-referencing multiple databases[†] and will be extended with multidatabase querying facilities. This facility also allows developing modular multidatabase systems (rather than a single monolithic system) for scientific applications. For example, the new version of GDB is developed as a multidatabase system consisting of six interconnected databases.

6. Customized and application-specific operations. Scientific databases, such as GDB, often use existing customized software for specialized data manipulations, such as data replication and error handling. In order to allow integrating such software into OPM, OPM update methods have been provided with 'hooks' to external procedures. Furthermore, scientific applications often need special application-specific operations, such as operations for comparing DNA fragments. In order to allow incorporating such operations into OPM, OPM will be extended with application-specific methods.

7. Database efficiency. Scientific databases generally contain very large amounts of data. In order to efficiently retrieve data from large databases, certain physical database techniques (e.g., indexing, segment allocation, data duplication) are required. Currently, OPM tools provide only limited support for physical database design, such as generation of default indexes and default segment allocations. We plan to extend the OPM tools with more powerful physical database design facilities.

8. CONCLUDING REMARKS

We have overviewed in this paper the Object-Protocol Model (OPM). OPM provides constructs for modeling scientific database applications in terms of objects and protocols. OPM allows users to have different views of the same database application and provides them with high-level query capabilities. The main OPM constructs have been illustrated with an example taken from the field of molecular biology database applications. Examples of using OPM for modeling scientific databases are provided via World Wide Web.[‡]

We have also developed a suite of data management tools based on OPM. Some of these tools, such as the OPM Schema Editor, are DBMS-independent. Other tools, such as the OPM Schema Translator, are currently implemented on top of relational DBMSs. We plan to develop similar tools targeting commercial object-oriented or object-relational DBMSs.

OPM and the OPM data management tools overviewed in this paper are 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 [3, 14, 15, 22] provide capabilities for modeling object classes, attributes, and subclass hierarchies. However, they fail to support one or more of the following capabilities: (i) specifying tuple attributes; (ii) allowing attributes to take values from different (unions of) domains of values; (iii)

[†]Note that OPM provides only a structural framework for cross-referencing. Data cross-references, ambiguities, and conflicts need to be resolved using additional procedures or techniques.

[‡]http://gizmo.lbl.gov/opm.html

specifying derived attributes; (iv) modeling protocol and protocol expansion; (v) specifying views; and (vi) powerful query and update languages. The first two constructs above allow avoiding artificial schema elements that do not represent application-specific objects and that unnecessarily increase the complexity and obscure the semantics of database specifications. OPM supports the modeling of objects, protocols, attributes, and provides all the features mentioned above.

Data Models for Scientific Applications. Several data models have been designed, like OPM, to support modeling scientific applications. For example, *MOOSE* [16, 33] has been designed for simulation environments, *POSDBM* [26] is a process-oriented scientific data model that provides capabilities for defining processes, transitions, and inputs/outputs, and *ordering information model* [19] 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 provide abstractions similar to protocol expansion or constructs similar to the input/output and protocol connections described in this paper.

Procedural Modeling in Office Information Systems. The modeling of *protocol classes* in OPM is related to the procedural modeling in office information systems [17, 25] in the sense that both specify 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, OPM is designed mainly for scientific applications in which protocols can involve alternative, sequences of, and optional subprotocol steps, the inputs and outputs of protocols can be represented by abstract object classes, and users can be interested in any level of protocol abstraction.

Object-Oriented Views. Object-oriented views have been explored in several papers, such as [1, 29, 30]. These papers deal only with derived object classes, and use query constructs (e.g., `SELECT` statements) for their specification. In addition to derived object classes, OPM provides derived protocol class constructs. Furthermore, the form of base and derived class constructs in OPM is similar.

Browsers. The OPM Browsing and Querying tool provides browsing, querying, and updating in a unified framework. Several browsing and querying systems and prototypes have been previously proposed [2, 24, 21, 23, 28]. Some of these systems support read-only access to databases, and do not provide modification capabilities [24, 23]. Furthermore, systems such as [23, 28] are also limited by their underlying model. For example, the models underlying the data browser of [28] or `KIVIEW` [23] do not support tuple attributes or associating attributes with union of value classes.

Relational Translation. Translating semantic schemas (without derived attributes) to relational schemas has been investigated in several papers (e.g., [20, 22, 32]). These translations are mainly concerned with representing semantic data structures using relational tables and constraints; little or no attention is paid to the operational (i.e., implementing methods) part of the translation. The OPM translation includes the generation of relational procedures implementing the OPM data retrieval and update methods.

Acknowledgements — 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.

Ernest Szeto implemented the current version of the OPM Schema Editor and the OPM Browsing and Querying tool. Tony Chen, Jun Wang, Francis Pang, Yiru Zhang, and Ofer Ben-Shachar contributed to the implementation of early versions of the OPM data management tools.

Tim Hunkapiller and Qing Zhang helped us understand the large scale sequencing application in their laboratory at Caltech. Sue Borchardt, Chris Brunn, Kenneth H. Fasman, Stan Letovsky, Peter Li, David Marquette, Krishnaveni Palaniappan, and Karen Phipps of GDB, suggested improvements of the OPM tools and shared with us their knowledge of genomic database applications.

The editors and referees provided suggestions that helped improving this paper.

REFERENCES

[1] S. Abiteboul and A. Bonner. Objects and views. In *Proc. ACM SIGMOD Conference* (1991).

[2] R. Agrawal, R.H. Gehani, and J. Srinivasav. Odeview: The graphical interface to ode. In *Proc. ACM SIGMOD Conference* (1990).

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

[4] 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) (1992).

[5] I.A. Chen and V.M. Markowitz. Mapping object-protocol schemas into relational database schemas and queries. Technical Report LBL-33048, Lawrence Berkeley Laboratory (1994).

[6] I.A. Chen and V.M. Markowitz. The object-protocol model view and query language. Technical Report LBL-33706, Lawrence Berkeley Laboratory, working document (1994).

[7] I.A. Chen and V.M. Markowitz. Opm schema translator. Technical Report LBL-35582, Lawrence Berkeley Laboratory (1994).

[8] I.A. Chen and V.M. Markowitz. Modeling scientific experiments with an object data model. In *Proc. 11th Int. Conf. on Data Engineering* (1995).

[9] I.A. Chen, V.M. Markowitz, and T. Hunkapiller. Modeling a large sequencing application with the object-protocol model. Technical Report LBL-35867, Lawrence Berkeley Laboratory (1994).

[10] S. Dar, N.H. Gehani, and H.V. Jagadish. Cql++: A sql for the ode object-oriented dbms. In A. Pirotte & al., editor, *Proc. Advances in Database Technology - EDBT'92*. Springer-Verlag, published as Lecture Notes in Computer Science, vol. 580 (1992).

[11] C.J. Date. Referential integrity. In *Relational Database - Selected Writings*. Addison-Wesley (1986).

[12] K.A. Frenkel. The human genome project and informatics. *Communications of ACM*, **34**(11) (1991).

[13] N. Goodman. An object-oriented dbms war story: Developing a genome mapping database in c++. In W. Kim, editor, *Modern Database Management: Object-Oriented and Multidatabase Techniques*. ACM Press (1994).

[14] M. Hammer and D. McLeod. Database description with sdm: A semantic database model. *ACM Transactions on Database Systems*, **6**(3) (1981).

[15] R. Hull and R. King. Semantic database modeling: Survey, applications, and research issues. *Computing Surveys*, **19**(3) (1987).

[16] Y.E. Ioannidis and M. Livny. Moose: Modeling objects in a simulation environment. In G.X. Ritter, editor, *Information Processing 89*. Elsevier Science Publishers B.V. (1989).

[17] R. King and D. McLeod. A database design methodology and tool for information systems. *ACM Transactions on Office Information Systems*, **3**(1) (1985).

[18] E.S. Lander, R. Langridge, and D.M. Saccocio. Mapping and interpreting biological information. *Communications of ACM*, **34**(11) (1991).

[19] A.J. Lee, E.A. Rundensteiner, S. Thomas, and S. Lafortune. An information model for genome map representation and assembly. In *Proc. 2nd Int. Conf. on Information and Knowledge Management* (1993).

[20] P. Lyngbaek and V. Vianu. Mapping a semantic database model to the relational model. In *Proc. ACM SIGMOD Conference* (1987).

[21] D. Maier, P. Nordquist, and M. Grossman. Displaying database objects. In *Proc. 1st Int. Conf. on Expert Database Systems* (1985).

[22] V.M. Markowitz and A. Shoshani. Representing extended entity-relationship structures in relational databases: A modular approach. *ACM Transactions on Database Systems*, **17**(3) (1992).

[23] A. Motro, A. D'Atri, and L. Tarantino. The design of kiview: An object-oriented browser. In *Proc. 2nd Int. Conf. on Expert Database Systems* (1989).

[24] J.L. de Oliveira and R. de Oliveira Anido. Browsing and querying in object-oriented databases. In *Proc. 2nd Int. Conf. on Information and Knowledge Management* (1993).

[25] B. Pernici et. al. C-todos: An automatic tool for office system conceptual design. *ACM Transactions on Information Systems*, **7**(4) (1989).

[26] J.M. Pratt and M. Cohen. A process-oriented scientific database model. *ACM SIGMOD Record*, **21**(3) (1992).

[27] R.S. Pressman. *Software engineering. A practitioner's approach*. McGraw-Hill (1987).

[28] T.R. Rogers and R.G.G. Cattell. Entity-relationship database user interfaces. In S.T. March, editor, *Proc. 6th Int. Conference on Entity-Relationship Approach*. Elsevier Science Publishing Company (1988).

[29] E.A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proc. 18th VLDB Conference* (1992).

[30] M.H. Scholl, C. Laasch, and M. Tresch. Updatable views in object-oriented databases. In C. Delobel & al., editor, *Deductive and Object-Oriented Databases*. Springer-Verlag, published as Lecture Notes in Computer Science, vol. 566 (1991).

[31] E. Szeto, I.A. Chen, and V.M. Markowitz. Opm browsing and querying tool. Technical Report LBL-35907, Lawrence Berkeley Laboratory (1994).

[32] S. Tsur and C. Zaniolo. An implementation of gem – supporting a semantic data model on a relational back-end. In *Proc. ACM SIGMOD Conference* (1984).

[33] J. Wiener and Y. Ioannidis. A moose and a fox can aid scientists with data management problems. In *Proc. 4th International Workshop on Database Programming Languages* (1993).

APPENDIX. AN OPM SCHEMA EXAMPLE

The following OPM schema models the application described in section 2.

```
OBJECT CLASS Contig_Map
   ID: contig_id
   ATTRIBUTE contig_id: INTEGER                not null
   ATTRIBUTE (fragment, position): set-of (Fragment, INTEGER)
   ATTRIBUTE owner: Person                     inverse of Person.owns
OBJECT CLASS Fragment
   ID: fragment_id
   ATTRIBUTE fragment_id: INTEGER              not null
   ATTRIBUTE sequence: VARCHAR(750)
   ATTRIBUTE length: INTEGER
   ATTRIBUTE owner: Person                     inverse of Person.owns
OBJECT CLASS Person
   ID: person_id
   ATTRIBUTE person_id: INTEGER                not null
   ATTRIBUTE name: CHAR(80)
   ATTRIBUTE owns: set-of  Contig_Map or Fragment
   ATTRIBUTE contig_map    DERIVATION: owns [Fragment] !fragment [Contig_Map]
OBJECT CLASS Connection
   ID: connection_id
   OUTPUT_OF DELETE CASCADES
   ATTRIBUTE connection_id: INTEGER            not null
   ATTRIBUTE left_fragment: Fragment
   ATTRIBUTE right_fragment:  Fragment
   ATTRIBUTE distance: INTEGER
PROTOCOL CLASS Construct
   ID: construct_id
   EXPANSION: (Overlap or Constraint), Assemble
   SUBPROTOCOLS DELETE NULLIFIES
   ATTRIBUTE construct_id: INTEGER             not null
   ATTRIBUTE fragments: set-of Fragment        not null  input
   ATTRIBUTE contig_map: Contig_Map            not null  output
PROTOCOL CLASS Overlap
   ID: overlap_id
   ATTRIBUTE overlap_id: INTEGER               not null
   ATTRIBUTE fragments: set-of Fragment        not null  input isa Construct.fragments
   ATTRIBUTE connections: set-of Connection    not null  output
   ATTRIBUTE program (name, version): (CHAR(40), CHAR(6))
PROTOCOL CLASS Constraint
   ID: constraint_id
   ATTRIBUTE constraint_id: INTEGER            not null
   ATTRIBUTE fragments: set-of Fragment        not null  input isa Construct.fragments
   ATTRIBUTE connections: set-of Connection    not null  output
   ATTRIBUTE constraint_type: CHAR(80)
PROTOCOL CLASS Assemble
   ID: assemble_id
   PREDECESSOR_PROTOCOL DELETE RESTRICTED
   ATTRIBUTE assemble_id: INTEGER              not null
   ATTRIBUTE connections: set-of Connection    not null  input from Overlap via connections
                                                             or Constraint via connections
   ATTRIBUTE contig_map: Contig_Map            not null  output isa Construct.contig_map
```