Viewing Genome Data as Objects for Application Development

Ellen R. Bergeman, Mark Graves, Charles B. Lawrence

Department of Cell Biology, Baylor College of Medicine,
One Baylor Plaza, Houston, TX 77030
email: {erb,mgraves,chas}@bcm.tmc.edu

Abstract

Genomics is becoming a data-intensive science, and an increasing number of laboratories are generating data which swamps storage in traditional paper-and-ink notebooks. Capturing the data flow requires large systems with multiple applications manipulating the same or similar data. Large systems often have conflicting requirements for data representation. Consistency across applications is a prime consideration, and appropriate data representation is an important issue in developing practical systems for molecular biologists. Graphs are a natural representation for describing genome data, while objects are good for modeling the behavior necessary for laboratory applications. We present a method for translating graph descriptions of genome data into objects using objects as views on graphs. Graph representations describe genome concepts while objects capture individual views for application development insuring consistency across genome applications.

Introduction

Representing genome data within databases and computer applications is an important topic within the genome research community. Genome data consists of real world entities and human-defined concepts, which must be accurately and flexibly represented in systems supporting ongoing research. Accurately defining these entities and concepts is a vital step in producing systems which can be used by biologists.

Each researcher has a "mind's-eye" view of the information with which he works. When listening to two biologists talk, one notices that these views may overlap but are seldom identical. When speaking, these ambiguities can be ignored or redefined through discussion, but in databases or applications, the same ambiguity must be eliminated or specified clearly. Often the choice is made to define a view which represents a minimum definition of a concept rather than deal with representing multiple definitions. Our approach to the problem of needing an "all things to all people" description of concepts which can be broken apart into individual views is to use graphs to describe the concepts and objects to represent the views.

We use graphs to capture the domain of genetics because genetic concepts and relationships connect to form a graph-like structure. A graph representation captures the concepts and relationships of genetics in an unrestricted though formal description. We have developed a representation for data which is based on the graph-theoretic definition of a graph as a collection of vertices and edges (Graves, Bergeman & Lawrence 1995). From this unrestricted description of a concept, individual views can be abstracted. Since all views originate from the same representation, consistency and accuracy are maintained.

Objects make viable views on graphs and are useful for application development because they provide modularity in design and inheritance of behavior. An object encapsulates a subgraph of the data graph and provides a means of introducing application-specific constraints on the data.

Graphs and Objects

Graphs are a mathematical formalism for describing complex structures. They consist of a collection of vertices and edges and often include labels on the nodes or edges. Data is represented as a graph by the relationships captured in the edges. Edges represent binary relationships, where two nodes are related by the edge between them.

Objects are a mechanism for encapsulating the behavior of computational processes. They typically include mechanisms for describing data abstractly, for hiding the behavior of object operations (or methods), and for making the object's internal structure and operations available to a restricted class of objects which inherit them. In a system, they may be designed around the data which they store or around the behavior which they provide.

We have combined graphs and objects in a novel approach so that objects provide interactive views on the database. A biologist uses a microscope to view a small area of interest in a cell or smear, increasing and decreasing magnification or moving the slide to change the view in order to discover information of interest. In much the same way, a user may move through a database to examine a part of the graph containing all data. Objects focus the attention to relevant details and provide behavior which can aid the researcher in finding information of interest.

Why Use Objects

An object-oriented approach is useful for application development because objects succinctly define a set of behaviors. Objects provide data abstraction, encapsulation, and inheritance of structure and behavior. Data abstraction allows objects to vary in complexity and to present data in a less implementation-specific manner. Encapsulation hides the implementation details which encourages modularity, promotes reuse, and simplifies future modification of code; all of which are important in a rapidly changing environment such as molecular biology research. Inheritance of structure and behavior facilitates reuse of software components and simplifies the task of meeting diverging needs in a changing environment.

Other view mechanisms on graphs are also possible. A more restricted technique would be to implement views as abstract data types, which do not allow for inheritance. A more general technique would be to use constructs from constructive type theory as views (Graves 1993a), which would also allow for type constructors. Using objects as views on graphs is a practical approach which makes graph representations available within readily available programming environments.

Object-oriented Design

Object-oriented design is the process of describing domain concepts as objects which have specific data and behavior. A requirements specification or other description of the application domain is used as a basis for discovery of objects. When design is completed, the object descriptions are used to implement the application.

The three aspects of object design are domain modeling, application modeling, and user interface modeling. Domain modeling concentrates on the concepts and relationships which are central to the domain. In genome applications, these are usually biological concepts. Application modeling concentrates on the objects which perform the computation-intensive tasks. User interface modeling concentrates on describing a reasonable user interface and any objects necessary for the user interface to function.

There are many object-oriented design methodologies which are used in industry. We have used several and found that for genome application development, no one methodology is effective in producing good object-oriented designs. We discuss some commonly used design methodologies in the next section.

Object-Oriented Software Engineering Methodologies

Within the object-oriented software engineering community, there are several paradigms or methodologies for software development. The methodologies which we have studied include Booch (Booch 1991), Class-Responsibility-Collaboration (Wirfs-Brock, Wilkerson & Wiener 1990), Object Modeling Technique (Rumbaugh et al. 1991), Objectory (Jacobsen et al. 1992), and Fusion (Coleman et al. 1994). Each methodology supports the framework of encapsulation, abstraction, modularity and hierarchical organization but have different descriptions of the analysis and design process.

The Booch methodology is one of the earliest object-oriented software engineering methodologies. It emphasizes the modeling of classes and objects: identifying them, their semantics, and relationships. The other methodologies use Booch as a foundation, emphasizing and extending various aspects. The Class-Responsibility-Collaboration (CRC) methodology defines classes, their responsibilities and collaborators. CRC is useful for object modeling, but does not provide enough support for discovery of the operational needs of the application. Rumbaugh's methodology, Object Modeling Technique (OMT), provides object, functional and dynamic modeling, emphasizing the operations of the application but not sufficiently supporting modeling of the domain.

The Fusion methodology, which is a second generation object-oriented software engineering methodology, combines aspects of Booch, CRC, and OMT in a common framework. It places equal emphasis on object modeling and modeling system interaction. Objectory also emphasizes the need to understand the process of the application as well as the objects. The requirements model of Objectory describes the system using use-cases, sequences of interactions between the user and the system.

We have found that no methodology will work when taken directly from a textbook. Although we use Fusion notation, our methodology is unique to genome application needs. For example, we discovered that the use cases of Objectory are more useful in analyzing the requirements for the user interface than the system interaction modeling of Fusion. We also found that a weakness in all the methodologies is capturing the complex genomic information. We incorporate graphs in design to address this weakness.

Graph Representation

Modeling genome data as graphs

Graphs provide a mechanism to represent genome data in a flexible framework which can be easily extended, which is important because genetics is advancing at a rapid rate. In addition, graphs are a natural model for genome data.

Some genome data has an intrinsic graph-like structure. Graphs have been used to describe mapping relationships (Cinkosky et al. 1992; Graves 1993b), gene regulation (Thieffry & Thomas 1994), metabolic pathways (Ochs 1994; Hofestaedt 1994; Karp and Paley 1994), and phylogenetic trees (Mirkin & Rodin 1984).

Experimental data also has a graph-like structure. Genetics concepts are defined in terms of their relationships to other concepts. These relationships are determined by experimental evidence. In molecular biology, most of the science is based on indirect observations, and the results of these experiments are relationships between the reagents used in the experiment and the result found, i.e., colony filter hybridization results, genetic map order information or gene function. There are few primitive concepts in genetics, other than the biochemistry, so most of the results are relationships between relationships between relationships, etc. The nesting of relationships has a graph-like structure.

Because most of the aspects of genomic relationships are independent of each other, it is useful to decompose (fully normalize) the relationships into binary relations. Binary relations are relations restricted to arity two. The binary relations can be combined with other binary relations to form concepts which may not have been considered when the database was originally developed. For example, an oligonucleotide may have originally been intended to be a PCR primer, but now may be considered to be a STS or non-polymorphic marker within the database.

Representation Languages

A representation language must be able to naturally express the data in the domain. The constructs in the formalism should closely correspond to the domain and capturing small changes in domain concepts should require only small changes in the representation. We have incorporated aspects of conceptual modeling, software engineering and knowledge representation into our graph representation language.

Conceptual modeling is the process of describing the concepts and relationships of a domain that are to be stored in a database (Brodie, Mylopoulos & Schmidt 1984). The process takes place within a theoretical framework called a conceptual model. A conceptual model is a data model which formalizes the representation and manipulation of concepts and relationships. A conceptual model captures the essential concepts in a domain without making decisions about the relative importance of each concept, which depends upon the specific database or application being developed. For example, a person has an name, address, and employer. A person also creates experiments. Each of the concepts: name, address, employer and experiment should be modeled as having a relationship with a person. A schema in a conceptual model can be translated into a database schema for a relational, object-oriented or other kind of database.

Domain analysis in software engineering (Prieto-Diaz & Arango 1991) is an aspect of software reuse. The emphasis of domain analysis is on discovering aspects of the domain which can be used in many different software applications. Although domain analysis does not provide the tools necessary for describing complex genome data, it does suggest a variety of approaches which might be taken. Neighbors (1980) suggests that a domain is a collection of objects, operations on the objects and relations between the objects. Prieto-Diaz (1987) suggests that rules of usage and multiple classifications into groups and abstractions are also useful. Greenspan, Mylopoulos & Borgida (1982) define domains (for requirement modeling) as objects, activities, and assertions which are organized using different abstraction mechanisms. Domain analysis also suggests mechanisms of formalizing domain information in terms of an algebra, with sorts, operations, and axioms (Srinivas 1990) or as semantic theories (Goguen 1986).

Knowledge representation provides graph representation languages which can be used to capture genome data. Using graphs for knowledge representation originated in semantic networks. Semantic networks were one of the first representation languages to attempt to capture structural relationships in a domain (Quillian 1968), and they are the precursor of current attribute value formalisms, conceptual modeling, and semantic databases. Semantic networks were not a good foundation for reasoning systems, but did demonstrate their ability to represent static association and structural information in domains (Winston 1970; Schank 1972; Schubert, Goebel & Cercone 1979; Brachman 1979). Semantic networks continued to evolve as a representation formalism (Lehmann 1992) and formed the basis of attribute value formalisms, such as feature structures (Kasper & Rounds 1986; Carpenter 1992), ψ-types (Ait-Kaci 1984), and terminological subsumption languages (Brachman & Schmolze 1985). Semantic networks also influenced development in databases leading to the creation of semantic databases (Hull & King 1987; Peckham & Maryanski 1988) and schema design tools for relational databases (Chen 1976). The graph-based modeling mechanism we propose is based on a type of attribute value formalism and uses graphs to model the concepts and their relations.

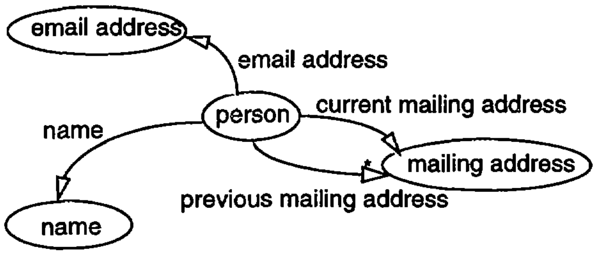
Graph Model

The graph representation which we use to capture genome concepts consists of three extensions to the basic definition of a graph. The first extension is the definition of a concept as a vertex of a graph. Each vertex is labeled with the name of a concept. The type of link between two genomic concepts is also important, thus the second addition to the definition of a graph is a collection of edge labels. The edge labels specify the characteristic of one of the objects, a relation between two vertices or the role that one of them has with respect to the other. For example, valid relations between a cosmid and a YAC would include "hybridizesTo", "generatedFrom", or "sharesSTS", or a plasmid might be in relationship to the plasmid's sequence or map location. The third extension is the addition of cardinality constraints to the edges.

There are four data types in our graph representation: concepts, edges, edge labels, and cardinalities. A graph is a collection of concepts, link names, cardinalities, and edges where:

- Concepts are the nodes of the graph and model the simple concepts and n-ary relations of the domain.
- Link names on the edges describe the relation which holds between the two vertices and are uniquely named.
- Cardinalities are either a positive integer or "many".
- Edges connect two vertices. There can be multiple edges between two vertices (with different link names). There is a cardinality for each vertex. Thus, an edge is a relation between two concepts, one link name, and two cardinalities

For example, a person may be represented as having a name, email address, a current address and multiple previous addresses as shown in the figure below:



There are an additional two extensions to the graph representation which are useful in practice: views and constraints. A graph representation may be broken up into several diagrams, or views, which represent the various configurations in which the links may occur. A diagram should be annotated with constraints which must hold between the concepts and links. Constraints augment the diagram by providing additional information on the concepts and links. For example, a constraint could state that the length of each cosmid sequence is between 10K and 100K base pairs.

An example graph representation is shown in Figure 1. Nodes and edges in bold on the diagram represent the view which will be modeled as objects in a later section. The graph describes a filter hybridization experiment for the laboratory process which acted as the application domain for this project. The project is concerned with identifying cosmid clones with regions homologous to cDNA clones. The process consists of a series of hybridization experiments between cDNA and cosmid clones. Within the process, multiple experiments are completed which work together to isolate a relationship between an individual cDNA clone and cosmid clone. Because each experiment can produce multiple results, the researchers needed a tool which would help them eliminate redundant experiments, record experimental data, and release final results to other researchers.

Domain Object Design

Genome application development requires accurately translation of genome concepts into useful domain objects. For an application to meet the needs of the user, it must be based on the correct definition of the concepts it represents. By defining a graph representation as the initial step in development, the domain expert and computer scientist can work together to capture all concepts in the genome domain, without restricting their thinking to a single application. To create an application, a developer must select subgraphs from the graph representation which describe the needed concepts and convert them into objects. In this section, we describe the process of creating the graph representation and translating views on the representation into an object model.

Modeling

A biologist and informatician work together to describe the domain using a graph representation. The biologist has the knowledge of the domain and the computer scientist has the ability to develop systems which use the domain knowledge. Definition of the concepts and relationships should be the responsibility of a domain expert. That person has a better understanding of the domain than a computer scientist could because he works within the domain on a daily basis.

The four steps to develop a graph representation of the domain are: listing the domain concepts, creating simple sentences which describe relationships in the domain, drawing major concepts as nodes in a graph, adding relationships as edges in a graph.

The first step in creating a graph representation is to list the concepts in the domain. Concepts are real world objects, relationships, and events, such as Experiment, YAC, or Hybridization.

The second step is to list simple sentences containing two domain concepts and a linking word or phrase. Linking phrases are descriptions of the interaction of the two domain concepts which describe the relationship clearly. They include: "has a name", "contains as an element", "hybridizes to", "probes". If the phrase describes a complex relationship, the relationship should be treated as a concept. For example, "YAC hybridizes to an STS" is a complex con-

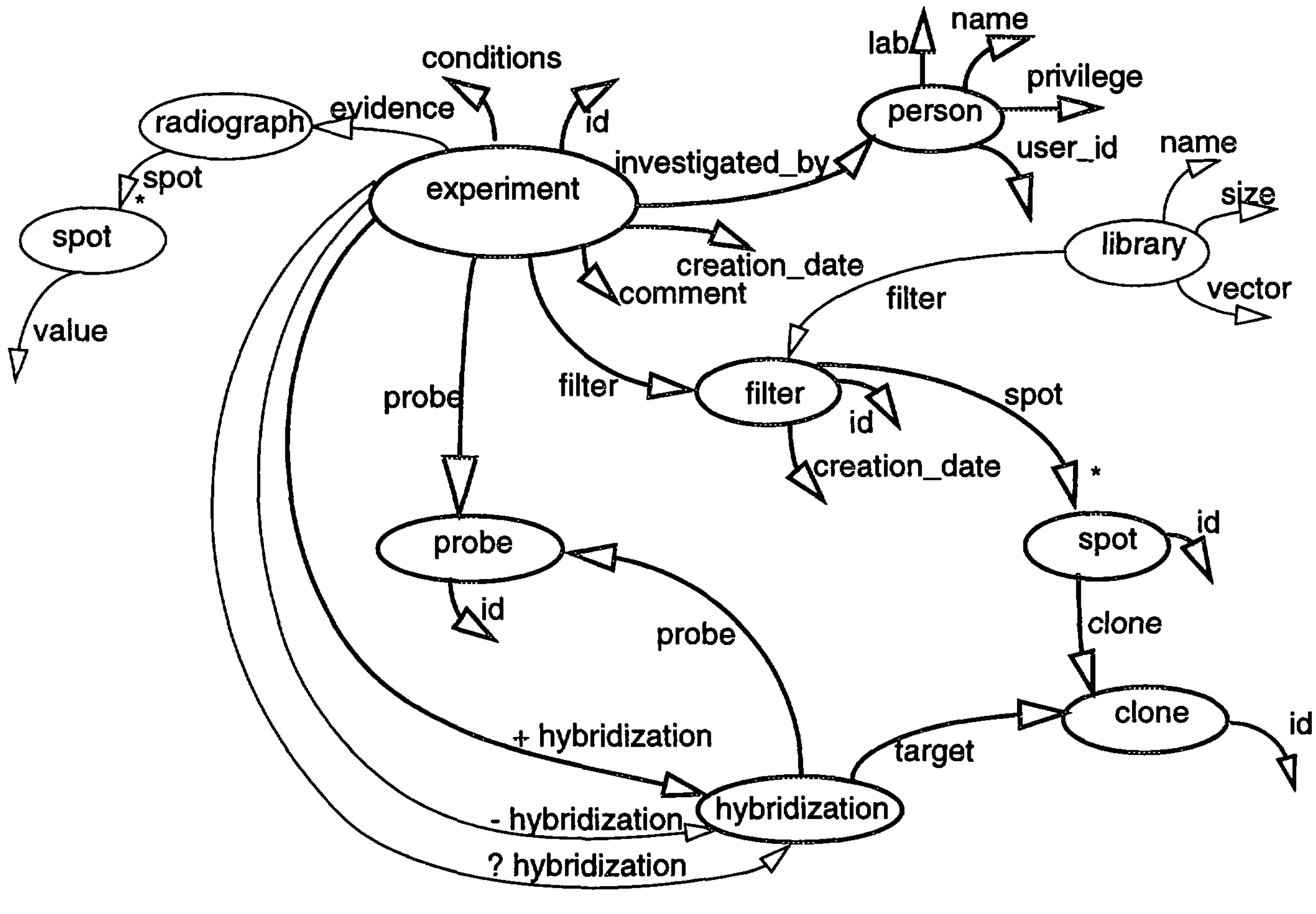


Figure 1: Graph schema for a Filter Hybridization Experiment

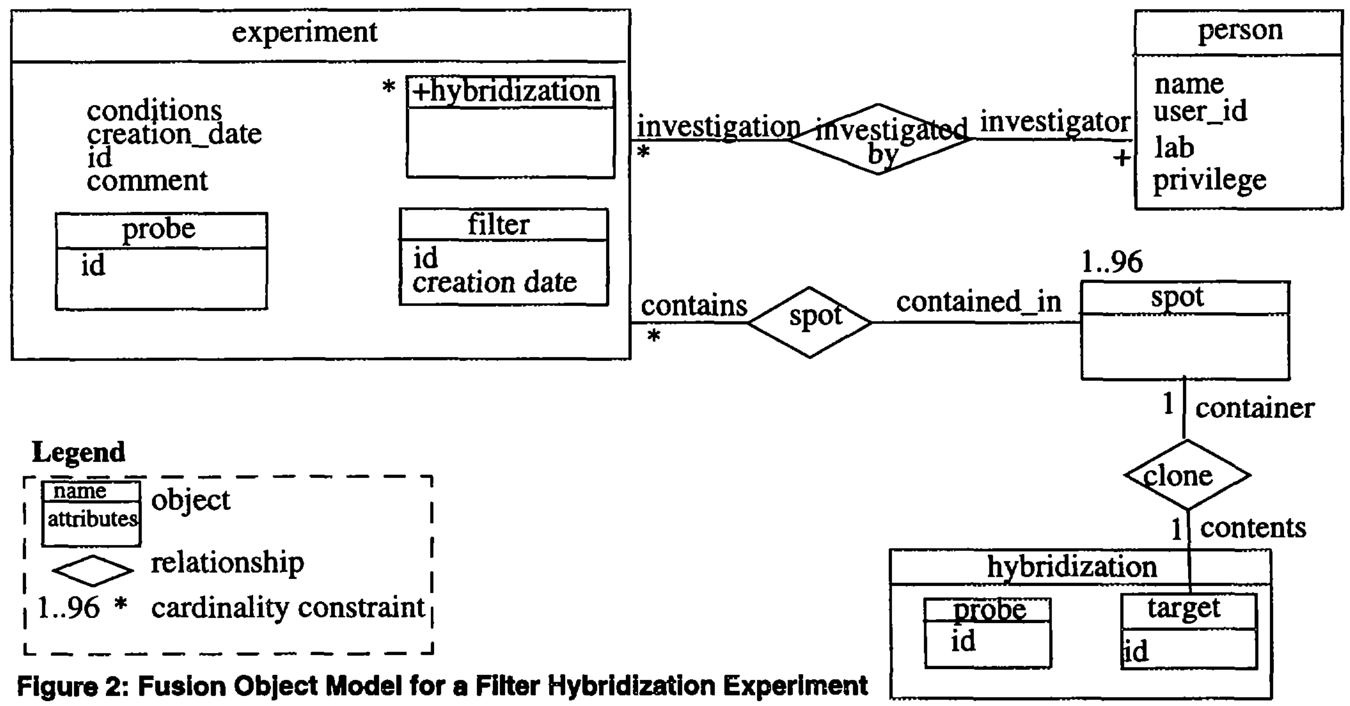


Figure 2: Fusion Object Model for a Filter Hybridization Experiment

cept which is important in the experimental physical mapping domain. It is necessary to create the concept of Hybridization and create the sentences "YAC is target in a Hybridization" and "STS is probe in a Hybridization" and "Hybridization has Experimental Evidence" instead of the single sentence "YAC hybridizes to STS".

After the concepts have been linked using simple sentences, the third step is to select major concepts from the concept list and draw them as nodes in a graph. The major concepts should be selected based on each concept's relative importance in the domain.

The fourth step is to add edges between the nodes which represent the linking phrases of step two. The direction of the edge should be the same as the sentence. To make the graph more readable, we have chosen to drop "is a" and "has a" parts of the linking phrase from the label which results in some edges having the same label as the 'receiving' node.

After domain concepts have been refined, the graph representation can be used to define objects which are used during application development.

Domain Object View

A graph representation should capture all concepts and relationships in the domain. When developing an application, it is necessary to select the appropriate concepts and relationships and eliminate all others. A domain object view is a graph which contains only concepts and relationships to be used in a specific application. The nodes and edges drawn in bold in Figure 1 are an example of a view as a subgraph of a graph model.

Domain Object Diagram

A domain object diagram is a graph which contains the concepts which should be implemented as objects in the application. When creating the domain object diagram, the developer and domain expert decide which concepts in the domain object view have primary roles in the application and which are secondary. The concepts with primary roles will be modeled as objects in the application.

A heuristic which we have found useful for creating the domain object diagram is to consider all nodes which have no outgoing arcs to play secondary roles. This is based on the realization that these nodes were only in the view because they helped to describe a primary concept and did not need relationships to describe themselves. While this heuristic is reasonable, it is important to not make the assumption that every such node can be eliminated. It is possible that a primary concept could be simple enough that it has no outgoing edges though it is vital to the application domain.

Eliminating the secondary nodes from the graph leaves the domain objects. The domain object diagram is basically a subgraph of the entire domain object view. No new concepts should be introduced into this diagram. If the diagram does not include all concepts which the domain expert knows to be important to the application, the domain object view should be reevaluated. No decision is made in this step other than eliminating secondary nodes.

Aggregation Diagram

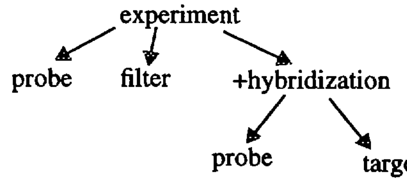
The aggregation diagram describes the hierarchical relationship of domain objects. From the domain object diagram, an aggregation diagram can be developed. The aggregation diagram captures parent/child relationships between the concepts of the domain.

Aggregation is a parent/child relationship between an object and other objects in the domain. When one object depends upon another for its existence in the domain or the database, that relationship should be captured in an aggregation diagram. The relationship is not one of relative importance but essentiality. For example, in a database, it is essential that key objects be added to the database along with, or before, non-key object can be added. That type of constraint should be captured in the aggregation diagram.

An aggregation diagram describes domain objects which play an attribute role but must be modeled as separate objects because they have their own attributes. For example, an experiment has a relationship with a person, filter, and probe. Person, filter, and probe all have their own attributes and are objects in the domain object diagram. According to the domain expert, an experiment should never exist in the application or database without being in a relationship with both a filter and a probe. Therefore an experiment is an aggregation of probe and filter. Within the domain, the same experiment could exist without knowing who completed the experiment, so it is not necessary for person to be a part of the aggregation.

An aggregation diagram has a tree-like structure, reflecting the hierarchical relationships within the domain. There can be multiple disconnected trees, with each tree describing one hierarchy. Any nodes in the domain object diagram which are essential for the existence of another concept should be a part of an aggregation diagram.

An aggregation diagram for the filter hybridization experiment would be drawn as:



Object Model

Aggregation diagrams, domain object diagrams, and domain object views are used together to create an object model. An object model should incorporate all concepts from the domain object view, describe all objects from the domain object diagram individually, and reflect the aggregation relationships described in the aggregation diagrams.

Creation of an object model should be a straightforward process of translating the information from the three previous diagrams into a standard object oriented software engineering methodology notation. All decisions concerning the objects should have been made at a previous step. If problems appear, the previous diagrams should be reevaluated before the object model is completed.

The process of creating the object model is as follows:

Using some object-oriented software engineering methodology notation, each node in the domain object diagram is drawn as an object. All nodes in the domain object view which were dropped from the domain object diagram because they were secondary are drawn as attributes of their primary objects. Parent objects in the aggregation diagrams are drawn as aggregations of their children. Non-aggregation relationships which are included in the domain object view are added as relationships between objects in the object model.

We use Fusion notation (Coleman et al. 1994), because concepts of multiplicity in relationships, aggregation of objects and attributes are combined into a single set of diagrams representing all domain objects in the application, as shown in Figure 2.

Implementation of Views

Development of object-oriented applications or databases can proceed directly from the object model described in the previous section using standard object-oriented software engineering methods. However, the real strength of our graph-based design of domain objects becomes more apparent when developing object-oriented applications on a graph database. Objects can be implemented directly from the domain object model to access graphs stored in a graph database. We have implemented an object-oriented data entry application based on this paradigm in the object-oriented programming language Smalltalk (Goldberg & Robson 1989). The application interacts with a graph database (Graves, Bergeman & Lawrence 1995), and all domain objects are implemented as views on the data graphs stored in the graph database.

In pure object-oriented systems, domain objects are not usually responsible for providing behavior to the application but instead are used to encapsulate a collection of data in a meaningful way. The data is stored in the object's internal structure and the object is responsible for providing access methods which can be used by other objects to access the data.

Instead of storing the data as part of the internal object structure, objects can have the ability to access the database for the data. By adding this behavior, domain objects become the interface between application and database, providing a clean interface between the two. The objects interact with the DBMS to store and retrieve data as needed by the application.

In addition, objects can be restricted to access only part of the graph database. When only application-specific data can be added to or retrieved from the database, domain objects may be considered as views on a graph database. Each domain object in the application represents and provides an interface to a part of the graph database. The object provides accessing methods which set and retrieve information in the graph much as other objects set and retrieve information from the object's own internal structure. For example, a "person" object in a traditional object-oriented implementation might contain accessing methods to retrieve the phone number, address, and email address of an individual. This information would be stored in the instance variables of an object. When objects are used as views, that information would be retrieved from the graph.

Accessing the database must be done through interaction with the database management system. The database must provide operations for data entry and access. These operations are:

1. Creating a new subgraph in the database;
2. Retrieving a subgraph which contains key values;
3. Adding new edges to the graph;
4. Querying the database for edges within a subgraph.

An object uses DBMS-specific commands to invoke operations on the database as part of its behavior. In addition, an object enforces application and database constraints as needed. These behaviors are implemented within attribute accessing methods. Accessing methods provide a public interface to the object for object-oriented development, while hiding DBMS-specific code.

Database interaction behavior can be naturally distributed between instance and class. The class creates new graphs and queries for graphs, acting as a template for any subgraphs which are to be added to the database. The class also can support constraints on any subgraph which it creates. Instances add and retrieve edges from a subgraph which exists within the database. An instance also enforces constraints on data which will be added to the graph.

Because accessing methods must provide a variety of behavior, we categorize attributes into three distinct groups which we call key, collection, and secondary attributes. Key attributes uniquely identify an object within the application and must not be changed. They are used in the process of creating or querying for a subgraph within the database. All other attributes are categorized as secondary or collection attributes based on whether their cardinality can be greater than one. Secondary attributes will always have zero or one values. Collection attributes may have any number of values. Accessing methods must provide behavior based on the attribute category. The process of selecting which attributes belong in each category is entirely application dependent and not a part of the database design process.

Key attributes uniquely identify the part of the graph which the object views. These attributes must be given values when an instance is created and should not be changed by the application. In essence, they are "read-only" to other objects in the application. Key attributes are defined by the needs of the application. For example, in the filter hybridization experiment application, we determined that a user should not define an experiment without specifying both the target and probe.

Collection attributes represent multiple edges of the same type in a subgraph. Accessing methods on these attributes are implemented to handle adding an edge to the graph and querying the subgraph for all edges of the same type. Positive hybridizations in an experiment is an example of a collection attribute. When the experiment subgraph is created in the database, the collection is empty but as results are entered, the content of the collection changes.

Constraints on multiplicity are also implemented within collection attributes. In the filter hybridization experiment, a filter was constrained to contain only 96 probes. Error checking was implemented as a part of accessing method behavior to notify the user if he was trying to add too many probes to a filter.

Secondary attributes represent data which are not essential to the existence of the domain object and which are constrained to one value, for example, "comment" and "creation date". These attributes are implemented to support adding and querying single edges. Depending on application needs, these attributes can also support changing or replacing data in the database.

Because objects are views on data graphs, it is possible to implement objects in the same application which access the same graph but provide different functionality and constraints. Inheritance naturally follows. In the same way, objects can be reused in other applications which have the same view requirements.

Conclusion

Graph representations are a natural and flexible means of describing complex genome concepts, capturing the overall view. Individual views can be represented as objects. Objects are good for developing genome applications because they encapsulate views for specific applications. Using the same graph representation for multiple object definitions provides consistency across applications.

We have found that:

1. Treating objects as views on graphs simplifies genome application development.
2. Object structure can be derived directly from a graph representation.
3. Using one graph to derive multiple object views helps insure consistency across genome applications.

Although more investigation is needed into developing the most appropriate representation for genome data, our findings indicate that both graphs and objects are useful for genome application development.

Acknowledgments

The authors thank Andy Arenson, Bob Cottingham, Pamela Culpepper, Dan Davison, Cheng Chi Lee, Wayne Parrott, Joanna Power, Randy Smith and Manfred Werhnert for frequent discussions of the ideas in this paper. This work was supported by the W.M. Keck Center for Computational Biology, the Baylor Human Genome Center funded by the NIH National Center for Human Genome Research, a grant to C.B.L. from the Department of Energy, and fellowships to M.G. from the National Library of Medicine and from the Department of Energy.

References

Ait-Kaci, H. 1984. *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures.* Ph.D. diss., Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA.

Booch, G. 1991. *Object Oriented Design with Applications.* Benjamin/Cummings.

Brachman, R.J. 1979. On the epistemological status of semantic networks. In N. V. Findler, ed., *Associative Networks - The Representation and Use of Knowledge by Computers.* Academic Press, New York. Also BBN Report 3807, April 1978.

Brachman, R. J. and Schmolze, J. G. 1985. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, pp. 17-216, August.

Brodie, M., Mylopoulos, J., and Schmidt, J., eds. 1984. *On conceptual modelling: Perspectives from artificial intelligence, databases, and programming languages.* Springer-Verlag, New York.

Carpenter, B. 1992. *The Logic of Typed Feature Structures.* Cambridge University Press.

Chen, P P. 1976. "The entity-relationship model: toward a unified view of data," *ACM Transactions on Database Systems* 1:1, pp. 9-36.

Cinkosky, M.J., Fickett, J.W., Barber, W.M., Bridgers, M.A., and Troup, C.D. 1992. SIGMA: A system for integrated genome map assembly. *Los Alamos Science* 20: 267-269.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. 1994. *Object-Oriented Development the Fusion Method.* Prentice Hall.

Goguen, J. 1986. Reusing and interconnecting software components. *IEEE Computer,* 19(2):16-28.

Goldberg, A., and Robson, D. 1989. *Smalltalk-80: The Language.* Addison-Wesley, Reading, MA.

Graves, M. 1993a. *Theories and tools for designing application-specific knowledge base data models.* Ph.D. diss., Dept. of Electrical Engineering and Computer Science, University of Michigan. University Microfilms, Inc.

Graves, M. 1993b. Integrating order and distance relationships from heterogeneous maps. In L. Hunter, D. Searls and J. Shavlik, eds., *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology (ISMB-93),* AAAI/MIT Press, Menlo Park, CA, July.

Graves, M., Bergeman, E.R. and Lawrence, C.B. 1995. A Graph-Theoretic Data Model for Genome Mapping Databases. In the Proceedings of the Hawaii International Conference on System Sciences-28, Biotechnology Computing Track. January.

Greenspan, S., Mylopoulos, J., and Borgida, A. 1982. Capturing more world knowledge in the requirements specification. In *Proc of the 6yth International Conference on Software Engineering*, pp. 225-234.

Hofestaedt, R. 1994. Modeling and Visualization of Metabolic Bottlenecks. In H. Lim, ed., *Proceedings of The Third International Conference on Bioinformatics and Genome Research.* World Scientific Publishing, June.

Hull, R., and King, R. 1987. Semantic database modeling: survey, applications, and research issues. *ACM Computing Surveys* 19:201-260, September.

Jacobsen, I., Christerson, M., Jonsson, P., and Overgaard, G. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach.* ACM Press.

Karp, P., and Paley, S. 1994. Automated drawing of metabolic pathways. In H. Lim, ed., *Proceedings of The Third International Conference on Bioinformatics and Genome Research.* World Scientific Publishing. June.

Kasper, R. T., and Rounds, W. C. 1986. A logical semantics for feature structures. In *Proceedings of the 24th Annual Conference of the Association for Computational Linguistics,* pp. 235-242.

Lehmann, F. 1992. *Semantic networks in artificial intelligence,* Pergamon Press.

Mirkin, B.G. and Rodin, S.N. 1984. *Graphs and Genes,* volume 11 of *Biomathematics.* Spreinger-Verlag.

Neighbors, J. M. 1980. *Software Construction Using Components.* Ph.D. diss, Dept. of Information and Computer Science, Univ of California, Irvine.

Ochs, R. 1994. A relational database model for metabolic information. In H. Lim, ed., *Proceedings of The Third International Conference on Bioinformatics and Genome Research.* World Scientific Publishing, June.

Peckman, J., and Maryanski, F. 1988. Semantic data models. *ACM Computing Surveys,* 20:153-189, September.

Prieto-Diaz, R. 1987. Domain Analysis for reusability. In *Proc of COMPSAC 87: The Eleventh Annual International Computer Software and Applications Conference,* pages 23-29. IEEE, October.

Prieto-Diaz, R., and Arango, G. eds. 1991. *Domain Analysis and Software Systems Modeling.* IEEE.

Quillian, M.R. 1968. Semantic memory. In M. Minsky, ed., *Semantic Information Processing.* The MIT Press, Cambridge, MA. Also Ph.D. Thesis, Carnegie Institute of Technology, 1967.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Loernsen, W. 1991. *Object-Oriented Modeling and Design.* Prentice Hall, New Jersey.

Schank. R.C. 1972. Conceptual dependency: A theory of natural language understanding. *Cognitive Psychology,* 3(4):552-631.

Schubert, L.K., Goebel, R.G., and Cercone, N.J. 1979. The structure and organization of a semantic net for comprehension and inference. In Findler, ed., *Associative Networks - The representation and use of knowledge in computers,* pp. 121-175. Academic Press, New York.

Srinivas, Y. 1990. Algebraic specification for domains. Technical Report, ASE-RPT-102, Dept. of Information and Computer Science, Univ of California, Irvine.

Thieffry, D., and Thomas, R. 1994. Logical Analysis of Genetic Regulatory Networks. In H. Lim, ed., *Proceedings of The Third International Conference on Bioinformatics and Genome Research.* World Scientific Publishing. June.

Winston, P.H. 1970. Learning structural descriptions from examples. Technical Report, MIT AI-TR-231, MIT, Cambridge, MA, September.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L. 1990. *Designing Object-Oriented Software.* Prentice Hall.