Conceptual data modelling for bioinformatics

Erich Bornberg-Bauer and Norman W. Paton

Date received (in revised form): 5th March 2002

Erich Bornberg-Bauer
is a Senior Lecturer in Bioinformatics in the School of Biological Sciences, University of Manchester, UK. His research interests past and present range from the theory of biopolymer evolution to modelling biochemical pathways, from the analysis of modular composition of genomes and proteins to the integration of biological data.

Norman W. Paton
is a Professor of Computer Science at the University of Manchester where he co-leads the Information Management Group. His research interests have principally been in databases, in particular active databases, spatial databases, deductive object-oriented databases and user interfaces to databases. He is currently working on parallel object databases, spatio-temporal databases, distributed information systems and information management for bioinformatics.

Keywords: *conceptual data model (CDM), database, entity–relationship (ER), unified modelling language (UML), biological data*

Erich Bornberg-Bauer,
School of Biological Sciences,
University of Manchester,
2.205 Stopford Building,
Oxford Road,
Manchester M13 9PT, UK

Tel: +44 (0) 161 275 7396
E-mail: ebb@bioinf.man.ac.uk

Abstract

Current research in the biosciences depends heavily on the effective exploitation of huge amounts of data. These are in disparate formats, remotely dispersed, and based on the different vocabularies of various disciplines. Furthermore, data are often stored or distributed using formats that leave implicit many important features relating to the structure and semantics of the data. Conceptual data modelling involves the development of implementation-independent models that capture and make explicit the principal structural properties of data. Entities such as a biopolymer or a reaction, and their relations, eg catalyses, can be formalised using a conceptual data model. Conceptual models are implementation-independent and can be transformed in systematic ways for implementation using different platforms, eg traditional database management systems. This paper describes the basics of the most widely used conceptual modelling notations, the ER (entity–relationship) model and the class diagrams of the UML (unified modelling language), and illustrates their use through several examples from bioinformatics. In particular, models are presented for protein structures and motifs, and for genomic sequences.

INTRODUCTION

Driven by genome projects and the recent development of other new techniques, such as proteomics or ligand screening, the amount of biological data is rapidly increasing (see Baxevanis[1] for an overview of current databases). However, not only are there ever-increasing quantities of data available in biology, the variety and complexity of the different information resources are also tending to increase with time.

Large centralised repositories for data, such as SWISS-PROT or Genbank, are carefully managed, often using modern data management techniques. However, the increasing prevalence of experimental techniques that generate large quantities of data means that ever more laboratories are faced with information management challenges. Managing large quantities of complex data in a systematic and efficient manner is not straightforward, and *ad-hoc* techniques that may have sufficed in the past will increasingly be a barrier to effective integration and analysis of experimental results in the future.

One important aspect of data management is coming to a clear understanding of the nature of the available data. What different kinds of data are generated by specific experimental techniques? How do these relate to other data produced in the laboratory or beyond? What information is derived from the primary data? What data need to be stored in perpetuity, and what can be summarised and then backed-up or discarded? What additional information is required to validate or analyse different data sets? What quantities of information are likely to be produced and will need to be stored as a result of an experimental activity? Obtaining answers to questions such as these in a systematic way is much more straightforward in the context of clear and comprehensive models of the relevant data. Conceptual data models make explicit the structural properties of data, and as such are useful for capturing, refining and communicating details about the data in a laboratory or a database.

Once a conceptual model of data has

The principal notions in conceptual modelling are: entity type and relationships

been produced, it tends to look like common sense. However, constructing conceptual models is a challenging, often iterative process. This paper seeks to increase the profile of conceptual modelling techniques in bioinformatics, and to collect together representative examples of conceptual models that can be used or built upon to support information management tasks in different parts of bioinformatics. The target audience is primarily (computational) biologists and bioinformaticians with minimal experience of data modelling and database design, who nevertheless find themselves involved with the development of biological databases. Thus we hope to improve the ease with which bioinformaticians can communicate with experts in data modelling, and then to judge independently if conceptual modelling is appropriate for their needs or not.

The paper is structured as follows. The next section provides some background material on conceptual models and design processes, and gives some definitions which are valid across most modelling notations. The section following on 'Entity–relationship modelling' gives an overview of ER modelling, and illustrates the approach using a model of sequence patterns. The section 'Unified modelling language' gives an overview of UML class diagrams, and illustrates such diagrams using models of protein structure and genome sequences. The final section discusses the role of conceptual modelling in bioinformatics.

CONTEXT

Definitions

A conceptual data model (CDM) provides a notation by which the structural properties of data (the structuring of data and their relationships) from a certain domain (a field of knowledge such as biochemistry or structural molecular biology) can be described in a precise but implementation-independent manner. The resulting model can then be more or less directly translated into the actual implementation, whether relational, object-oriented or semi-structured, and eventually populated, ie 'filled', with the actual data.

Many notations have been proposed for conceptual modelling, but most have two principal notions: entity types and relationships.

- **Entity types**: an entity type provides a description of the properties that are shared by a collection of entities in a domain. For example, Protein could be an entity type, with attributes including sequence, name, molecular weight, accession number and species. A single entity type is expected to have many *instances*, each of which gives values to the attributes specified in the corresponding type. For example, *human α-haemoglobin* and *whale myoglobin* are the names of two instances of the entity type Protein. The values of their attribute species are *human* and *whale* respectively.

- **Relationships**: a relationship represents an association between two (or more) entity types. For example, a Protein could interact with several other Proteins, or could be a member of a family. There may be different categories of relationship, which characterise the nature of the relationship. For example, there may be a notation to represent the fact that one entity type *is-part-of* another (eg a Beta Strand is part of a Sheet in the secondary structure of a Protein) or that one entity type *is-a-kind-of* another (eg an Enzyme is a kind of Protein).*

Once constructed, a conceptual data model of a domain describes the data in the domain, and can be seen to place constraints on the attributes and relationships that are valid within the domain. A conceptual data model is often developed in the context of an application

*(For the sake of simplicity we ignore the fact that RNA can be an enzyme.)

or a collection of tasks that is to be carried out. However, many biological databases contain data on a particular species or resulting from a particular type of experimental or analytical activity, and are thus not focused on particular uses of such data. In general, though, making clear and appropriate modelling decisions depends on the purpose for which the data are required. For example, in a protein database it may be necessary only to know the name of each species of each Protein. If so, the name of the species can be an attribute of Protein. However, if it is necessary to know more about the species (eg its taxonomic name as well as its common name, the areas in the world where it resides, etc.), then species is better modelled as an entity type, with attributes and relationships of its own.

There is no universally accepted standard for either the design process or the notation

The design process

A design activity involves a combination of a design process and a modelling language. A design process provides a sequence of steps through which the developers proceed when constructing a conceptual model. In the same way as there is no universally accepted notation for conceptual modelling, there is also no universally accepted design process.

Figure 1 outlines a possible design process based on one provided in Elmasri and Navathe.[2] This process depicts three tasks:

- requirements analysis – the identification of the needs of the application and the sources of information that the modelling activity seeks to support;
- conceptual design – the use of a conceptual data modelling notation to describe the data identified in requirements analysis; and
- logical design – the development of an

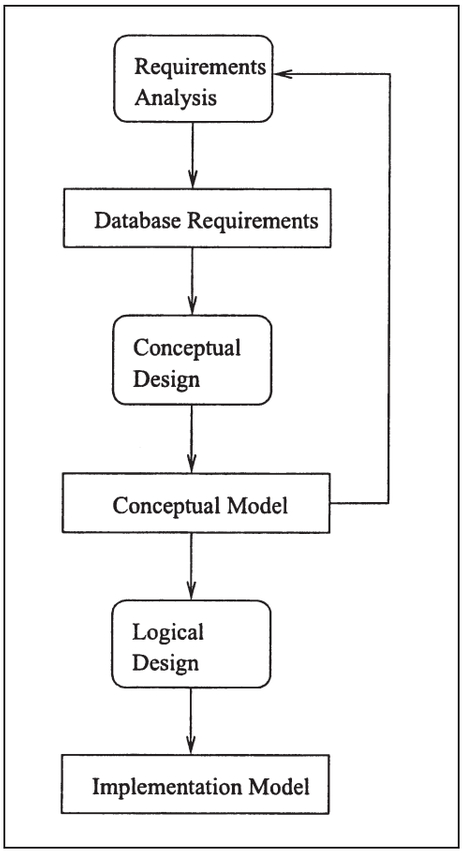

Figure 1: Example of a conceptual modelling process. Especially at the level of a conceptual model many inconsistencies and neccessary amendments may become obvious such that a revision at this stage may be necessary and useful to avoid more demanding revisions at a later stage. See text for more details

implementation data structure from the conceptual model, such as the creation of a collection of table definitions for use in a relational database system.

An important feature of the process illustrated in Figure 1 is that it is iterative – the construction of the conceptual model may raise issues that need to be clarified by revisiting the tasks to be supported or the sources of information to be described.

Conceptual data models allow precise statements

The role of the conceptual data model in the design process is to allow precise statements to be made about the data of interest in a manner that can be communicated to others. The comprehensibility of a conceptual model is important, as it is used both in discussions with subject experts whose understanding of the relevant data is to be described, and by the developers of software that makes use of the data. A general remark on conceptual models is that they are usually much easier to read than to construct.

In our experience, conceptual data modelling is often best conducted as a collaborative process, in which models are constructed incrementally, for example, on a whiteboard, so that different people can provide input on the features of the data being modelled. In general, we have been involved in modelling activities in bioinformatics in which computer scientists and biologists work together on models. This sort of joint approach is probably the most effective in practice, as experienced modellers should be able both to ask pertinent questions that guide the development of a model and avoid modelling errors. Developing models that describe all the data that are valid and relevant, while prohibiting the inclusion of data that are invalid or do not occur in practice is generally both a challenging and rewarding process.

Selection of conceptual data models

There are many different conceptual data modelling notations, although the most well-known families are the entity–relationship (ER) models[2,3] and the object-oriented models.[4–6] In our experience the success or failure of a conceptual modelling activity is not generally dependent on the conceptual modelling notation used. Thus the sorts of issues that can appropriately influence the selection of a CDM are local experience in the use of different techniques, availability of appropriate modelling tools, and likely implementation platform. In terms of the latter, ER models are targeted principally at database applications, and the mapping of such models onto relational database systems is well understood. However, if an implementation is likely to end up using object-oriented programming, middleware or database techniques, working from an object modelling language is likely to be most appropriate. Details on how to develop implementations from object models over several different platforms are provided in Blaha and Premerlani.[7]

ENTITY–RELATIONSHIP MODELLING

An ER schema consists of entity types, relationships and attributes. As described above in the section on 'Definitions', an entity type provides a description of the properties that are shared by a collection of instances in a domain. An entity type is drawn as a rectangle that encloses its name. For example, in Figure 2, DNA, Protein, Enzyme, Reaction and Biopolymer are all entity types. The instances of an entity type are known as entities. The attributes of an entity type indicate what values can be stored to identify or describe an instance of the type. The attributes of an entity type are depicted in ovals directly connected to the entity type. For example, the entity type Biopolymer in Figure 2 has attributes accno (for accession number), name, species and sequence.

One or more of the attributes of an entity type may be designated as the *key*, which is depicted by the name(s) of the

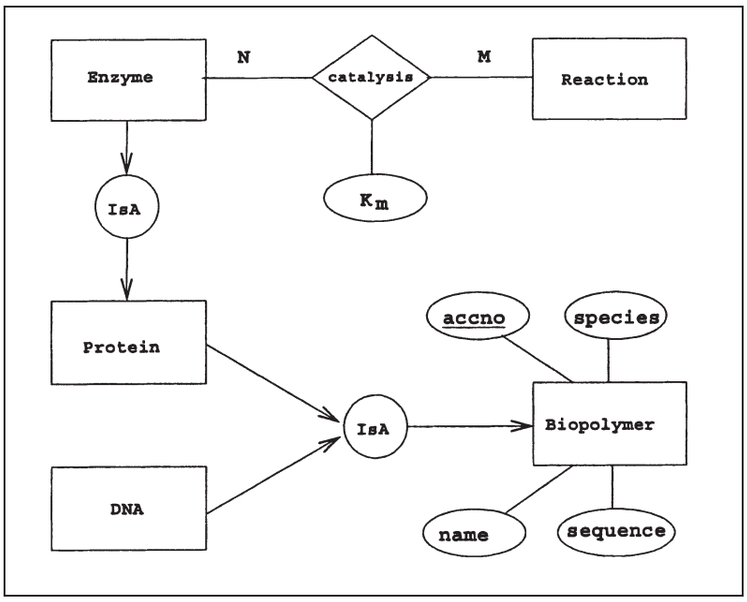

Figure 2: ER notation for some biological concepts. Reaction is related to Enzyme through a *many-to-many* relationship (see the section on 'Entity–relationship modelling' for more details). Each enzyme is a protein, which is depicted by the IsA relationship between Enzyme and Protein. Both DNA and Protein are kinds of Biopolymer as is depicted by the IsA relationship. This particular schema is not part of any published model, but has been designed for illustration purposes

attribute(s) being underlined in the diagram. For example, **accno** is the key of **Biopolymer** in Figure 2. This means that there can be at most a single instance of **Biopolymer** with a given **accno**. If no single attribute of an entity type can be used as the key, then it is possible that several attributes can be used together to uniquely identify the instances of the type, in which case that group of attributes can be underlined in the schema diagram. A key with several components is known as a *compound key*.

It is common for several entity types within a schema to share attributes and relationships. For example, both **Protein** and **DNA** have a **species**. Such sharing of attributes can be represented using *is-a-kind-of* relationships between entity types, which are depicted in Figure 2 by arrows from the more specialised type to the more general type through a circle containing **IsA**. For example, both **DNA** and **Protein** are kinds of **Biopolymer** in Figure 2, and **Enzyme** is a kind of **Protein**. **Biopolymer** is said to be the *supertype* of both **DNA** and **Protein**, and both **DNA** and **Protein** are *subtypes* of **Biopolymer**.

Such relationships have two principal roles. Firstly, the properties of a supertype are inherited by its subtypes, thereby leading to more concise models. For example, in Figure 2, the attributes associated with **Biopolymer** are all inherited by **Protein** and **DNA** through the **IsA** relationship. Secondly, the *is-a-kind-of* relationship makes explicit relationships between the collections of instances of different entity types. For example, from Figure 2 one can deduce that every instance of **Enzyme** is also an instance of **Protein**, and that every instance of **Protein** is also an instance of **Biopolymer**.

Any relationship other than the *is-a-kind-of* relationship between two types is depicted by a rhombus that encloses the name of the relationship, and which is linked to the related entity types. Although some ER models allow a single relationship to be between more than two entity types, it is often considered good practice to use only binary relationships.

The **catalysis** relationship in Figure 2 is an example of a binary relationship between the entity types **Enzyme** and **Reaction** that indicates which reactions are catalysed by which enzymes. A single **Enzyme** (eg peroxidase) may catalyse many reactions (as, for example, peroxidase acts on several substrates), and a **Reaction** (eg peroxide degradation) may

be catalysed by many different enzymes (eg peroxidase and catalse). Therefore, this relationship is a *many-to-many* relationship. This is depicted by the M and the N in the figure, which essentially denote arbitrary numbers of participants in the relationship. The cardinality specified for a relationship can be left open ended, using M or N, or can be specified to be a particular value. Where a specific value of 1 is used, this gives rise to one-to-one or one-to-many relationships, the latter in particular being very common (eg a Genome has many Chromosomes, but a Chromosome is specific to a single Genome).

Relationships can themselves have attributes. It is appropriate for a relationship to have an attribute if the value to be recorded describes the relationship, and is not an attribute of either of the participating entity types. For example, the Michaelis constant would be specific for the pair of the reactant and the enzyme involved. A relationship may relate an entity type to itself. For example, an interaction relationship could be used to indicate which Proteins are known to interact with each other.

Given the constructs described above, there are likely to be many plausible conceptual models for describing a data set. Such variety can derive from differences in the purpose to which the data is to be put, or the stylistic preferences of the modeller. As a case in point, relationships can be more or less precisely modelled. For example, in Figure 2, the catalysis relationship between the entity types Enzyme and Reaction models the role of a specific kind of Biopolymer in a Reaction. However, the fact that an Enzyme is related in some way to a Reaction could have been represented by a much more general relationship, with a name such as participatesIn between Biopolymer and Reaction. Which approach is most suitable depends on the nature of the data to be modelled and the purpose for which the resulting database is being developed.

ER model for fingerprints

Conceptualising the relationship between sequences, their similarity and function is essential to exploiting the predictive power of comparative functional inference. PRINTS is a database of *fingerprints*, grouped sequence patterns that are characteristic of specific protein families.[8] Its major advantage lies in the possibility of selecting levels of groupings for family definitions by choosing different collections of motifs from a fingerprint. This is important because the definition of a protein family may vary with the level of stringency one decides to choose. Patterns are derived from SWISS-PROT and TrEMBL.[9] PRINTS is a typical example of a database in bioinformatics that was first implemented using ASCII files, and then migrated to a relational platform for performance and functionality reasons. Part of the migration process involved the development of a conceptual model for the data. PRINTS currently contains 1,210 entries and 7,200 motifs.[8]

The schema consists of three basic entity types, namely Sequence, Fingerprint and Motif, as shown in Figure 3. Each Sequence may contain any number of Motifs, each of which must appear in one or more sequences (*many-to-many* relationship). Each Fingerprint is a collection of Motifs, but each Motif must appear in one and only one Fingerprint. This is because within PRINTS a Motif is defined as a pattern that appears in a specific Fingerprint. However, a mobile protein fragment could cause exactly the same pattern to appear in another Fingerprint, which is handled in this context by the definition of more than one Motif associated with the same pattern. The relationship between a Fingerprint and a particular Sequence represents a functional characterisation of the protein. Depending on the number of Motifs in a Fingerprint which match a particular sequence, this assignment is considered to be more or less reliable. If all motifs occur in the right order, the relationship is

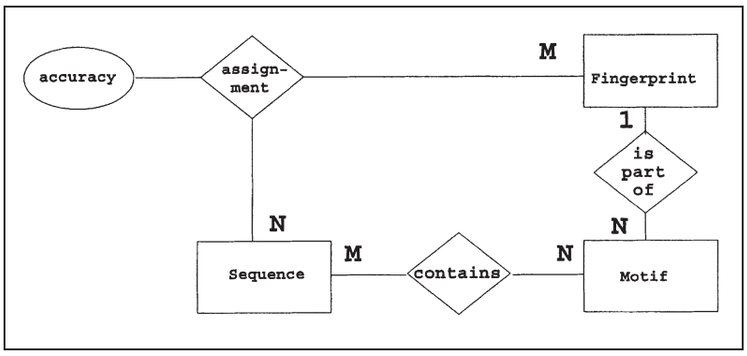

Figure 3: ER model for the PRINTS-S database[8]

considered to be a 'true positive'. If fewer motifs are matched in the sequence the possible values are 'true partial' or 'false partial'. This is denoted by an attribute (accuracy) which is associated with the Assignment relationship, which can take any of these three values. Thus accuracy is appropriately an attribute of the relationship, as the accuracy is a characteristic of the relationship between a Sequence and a Fingerprint, and not a characteristic of either of these entity types themselves.

Implementing from ER models

Although in principle the ER model is implementation platform-independent, it is most commonly used in conjunction with relational databases, and design environments supporting ER often include facilities for generating table definitions from an ER model. As the relational model provides different modelling facilities from the ER model, implementing an ER model using a relational database involves mapping the constructs of the ER model into those of the relational model.

Many constructs of the ER model can be mapped quite directly onto relational tables for implementation. For example, each entity type is represented by a table in the relational model. Thus in the relational implementation of PRINTS, there are Fingerprint, Sequence and Motif tables, as shown in Figure 4.

The attributes of an entity type are mapped to attributes of the corresponding table, and the key of an entity type generally becomes the key of its corresponding table.

The way a relationship in the ER model is mapped onto tables depends principally on the cardinality of the relationship. For example, a *one-to-many* relationship is represented by storing the key of the table at the *1* end of the relationship as an attribute of the table at the *many* end (a *foreign key*). For example, in Figure 3, the FingerprintAccession attribute of Motif is a foreign key of the Fingerprint table. By contrast, a

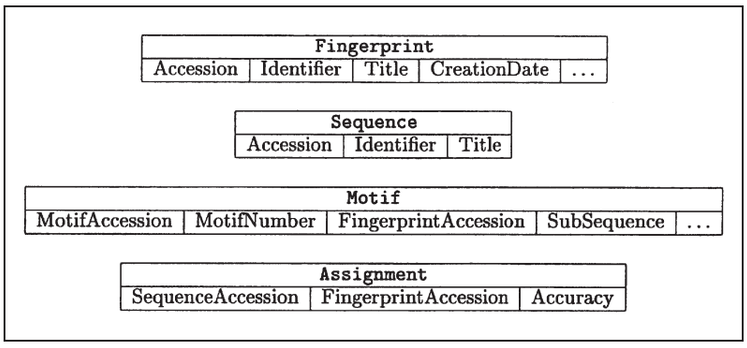

Figure 4: Examples of tables generated from the ER model of Figure 3. See section on 'Implementing from ER models' for further details

many-to-many relationship is represented by a table that has a compound key consisting of the keys of the related tables, as in the case of the table **Assignment**.

UNIFIED MODELLING LANGUAGE (UML)

UML is a standard object modelling language

UML[5] is the industry standard object modelling language. In this paper the focus is on class diagrams, which are used to model the structural aspects of data within UML. However, UML contains many different modelling notations, which also allow the behaviour of an application to be described in different ways, so UML can be seen as providing comprehensive application modelling facilities. For example, case diagrams can be used during requirements analysis to identify the principal categories of users making use of a system and the activities carried out by those user groups.

As an object-oriented modelling language, the central notation in UML is the class diagram. A *class* is a description of the attributes, operations and relationships of a set of objects. As such, a class in UML is analogous to an entity type in ER modelling. The instances of a class are referred to as objects, and are analogous to entities in ER modelling.

A class is depicted as a rectangle, within which is stated the name of the class. Optionally, the names and types of *attributes* and *operations* can also be depicted within the rectangle that represents the class. For example, Figure 5 includes the class **Protein**, which has attributes **name** and **accessionNumber**, both of which are of type **String**. The class **Protein** also has an operation **display()**, which can be expected to print or draw objects that are instances of **Protein**. By convention, the names of classes start with capital letters, and the names of attributes start with lower case letters. Where a name of a class or attribute is constructed from several words, the later words start with capital letters, as in **accessionNumber**. Unlike ER models, UML class diagrams do not support keys – this is probably because typical implementation platforms for UML classes do not support keys, whereas ER models are typically mapped to relational databases for implementation, where keys have a prominent role.

UML supports the description of several different kinds of relationships between classes. An *is-a-kind-of* relationship is depicted by a closed headed arrow, eg as from **Enzyme** to **Protein** in Figure 5. **Protein** is said to be a *superclass* of **Enzyme**, and **Enzyme** a *subclass* of **Protein**. All the attributes, relationships and operations of a superclass are inherited by its subclasses. A class may have zero, one or more superclasses or subclasses.

Relationships (other than the

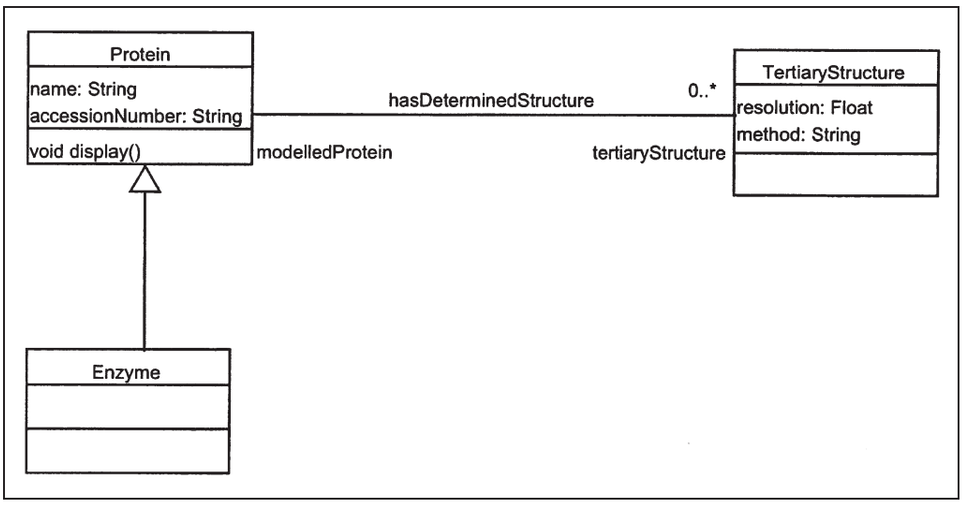

Figure 5: Example UML model. Three concepts (Enzyme, Protein, TertiaryStructure) from three domains (biochemistry, molecular biology and structural biology respectively) are integrated in one schema

generalisation relationship) between classes are known as associations in UML. For example, the fact that a protein may be associated with several experimentally determined tertiary structures can be represented by the association illustrated in Figure 5. The association hasDeterminedStructure links a Protein to information about its experimentally determined TertiaryStructures. It is possible to name associations in different ways. For example, when a class participates in a relationship it plays some role in the relationship; roles are the most common way of naming relationships in UML. In Figure 5, the class TertiaryStructure is fulfilling the role of the TertiaryStructure of the associated Protein. Furthermore, from the perspective of the TertiaryStructure, the class Protein is playing the role of a modelledProtein.

In analogy to the ER model, it is also possible to annotate a relationship with some information on its cardinality. For example, in Figure 5, a TertiaryStructure is the structure of a single protein (the default cardinality, and thus not shown explicitly), and a Protein may have several experimentally determined TertiaryStructures (as depicted by the 0^*, where * represents 'many').

As well as annotating an association with its cardinality, a further structure modelling feature of UML that is used in the subsequent examples is aggregation, which allows the representation of the *part of* relationship. An aggregation is depicted by ◊ at the end of the relationship that represents the *whole* in the *part–whole* relationship. Examples of aggregations are given in Figure 6, for example to indicate that a Chain is part of a Protein and that a SecondaryStructureElement is part of a Chain.

In this and subsequent sections, UML diagrams have been drawn using the UML editor ARGO, which can be downloaded.[10,11] It should be noted that UML class diagrams include more modelling facilities than are described here; for further details, see Booch *et al*.[5]

UML model for protein structure

This section describes a UML model for protein structure data, which is a simplification of the object-oriented data model provided in Gray *et al*.[12] The original model was implemented directly using an object database system. The UML diagram is given in Figure 6. The model includes primary, secondary and tertiary structure information. The topmost class in Figure 6 is Protein, of which all other classes are either directly or indirectly components. All the relationships between classes in Figure 6 are either aggregation or generalisation relationships.

A Protein has several attributes, which indicate its name, the code by which it is identified in PDB, and its molecularWeight. Each Protein consists of one or more Chains. A Chain can be seen as consisting of a collection of Residues or as a collection of SecondaryStructureElements.

The class Residue provides both primary structure (the name of the amino acid at a particular position within the Chain) and tertiary structure information (the coordinates of the residue within the model). The x, y and z coordinates of each atom associated with the Residue are modelled using the class Coordinates, which associates the position of each atom with the name of the atom (eg *ca* could be used to refer to the α-carbon).

The class SecondaryStructureElement is an *abstract* class – this is depicted in the diagram by the fact that its name is in Courier.* An abstract class is one for which no direct instance objects are ever created, but which can play a useful organisational role in the diagram. In this case, SecondaryStructureElement is the superclass of Loop, Helix and Strand, all of which can have direct instances.

Two of the subclasses of SecondaryStructureElement have

*(This should really be depicted in italics, but ARGO generates a Courier font in its Postscript generator.)

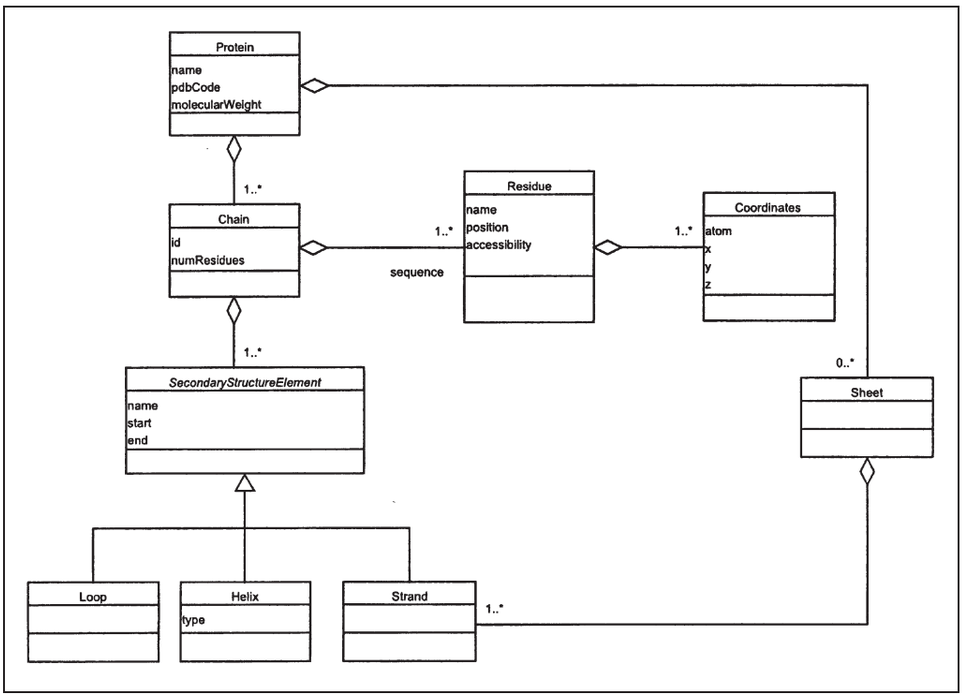

Figure 6: UML model for protein structure. See the section on 'UML model for protein structure' for further details

additional properties that are not shared by all SecondaryStructureElements. In particular, Helix has the attribute type which could, for example, be α or threeten, and each Strand is related to the Sheet of which it is part.

The modelling of SecondaryStructureElements illustrates a common dilemma in modelling – whether to use the generalisation hierarchy or an attribute to categorise the members of a collection. For example, the addition of an attribute type on SecondaryStructureElement could be seen as removing the need for the subclasses Loop, Helix and Strand – the type attribute could then take on a value that indicates whether a specific SecondaryStructureElement is a Loop, a Helix or a Strand. A criterion that encourages the use of a generalisation hierarchy in this sort of situation is the presence of attributes or relationships in the subclasses that are not applicable to the superclass (eg the relationship between Strand and Sheet). As there are no attributes or relationships specific to different kinds of helices in Figure 6, the different kinds of Helix are distinguished between using the attribute type, rather than through the introduction of additional subclasses.

In terms of modelling practice, UML is relaxed in many things. For example, in Figure 6 not all relationships are given names or role names – it is hoped that the use of aggregation will allow the user to infer names such as consistsOf and isPartOf rather than these having to be given explicitly; classes are given attributes, but the types of the attributes are not specified in the diagram – in general it is good practice when modelling at least to identify the attributes associated with different classes, as this is important in clarifying exactly what data each class actually models; and no classes are given operations – the original schema was produced to describe the data and not the way the data are used, so the emphasis was not on the behaviour associated with the classes.

UML model for genome sequences

The UML model provided in Figure 7, which is originally from Paton *et al.*,[13] can be used to describe sequence information

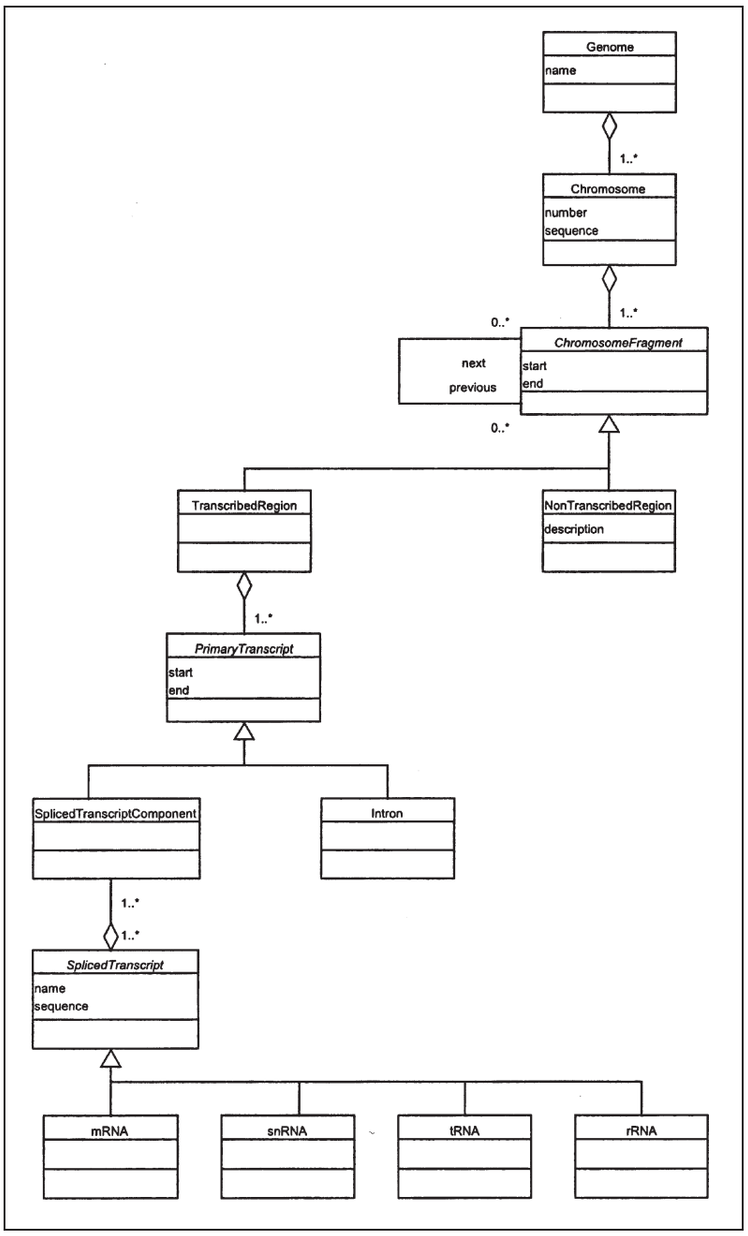

Figure 7: UML model for eukaryote genome sequence. See the section on 'UML model for genome sequences' for further details

for fully sequenced eukaryotic genomes. The model has been used as the basis for an implementation using an object database, and has been populated with sequence data from *Saccharomyces cerevisiae*. A Genome consists of one or more Chromosomes, each of which has a number and a sequence. These Chromosomes in turn consist of a collection of potentially overlapping ChromosomeFragments, each of which represents a TranscribedRegion or a NonTranscribedRegion within the chromosome. The next/previous relationship on ChromosomeFragment is an example of a recursive relationship, which is used in this context to provide an ordering to the ChromosomeFragments. In fact, UML allows a constraint to be specified for a relationship, to the effect that the elements at one end of the association are {ordered}. This can be written at the end of the relationship where the ordering

exists, but it may additionally be useful to have a **next/previous** relationship to directly associate related regions.

The class **NonTranscribedRegion** is used to represent features such as promoters, centromeres and telomeres, which are only distinguished between in the description attribute – these could generally benefit from more detailed modelling than is provided here. The DNA sequence associated with each **ChromosomeFragment** is modelled by recording the **start** and **end** positions of the sequence of each fragment in **ChromosomeFragment**. These start and end positions can be used to obtain the actual sequence of a **ChromosomeFragment** by looking up the relevant range in the sequence attribute of **Chromosome**.

The modelling of transcribed regions is somewhat involved, in that it is necessary to be able to capture alternative splicing.[13] Figure 8 illustrates the relationship between several of the classes in the model. The top part of the figure represents the **PrimaryTranscript**s associated with a single **TranscribedRegion**, and the bottom part of the figure illustrates two **SplicedTranscript**s that have resulted from alternative splicing. The **Intron**s, which are shown shaded, do not contribute to the **SplicedTranscript**s, but several of the **SplicedTranscriptComponent**s can potentially end up as part of differently constituted **SplicedTranscript**s. The *many-to-many* relationship between **SplicedTranscriptComponent** and **SplicedTranscript** represents the fact that a **SplicedTranscript** is commonly composed of more than one **SplicedTranscriptComponent**, and that on occasion there may be several alternative **SplicedTranscript**s that can result from a collection of **SplicedTranscriptComponent**s.

This model for genome sequences illustrates the extent to which the purpose of the model influences what is to be modelled. The model is explicitly targeted at fully sequenced genomes, and there is no attempt to describe the experimental data generated during the sequencing. An example of a conceptual model for genome mapping is given in Hu *et al.*[14]

Implementing from UML models

In principle, UML models are independent of the implementation platform to be used. In practice, mapping UML models, including class diagrams, onto object-oriented implementation

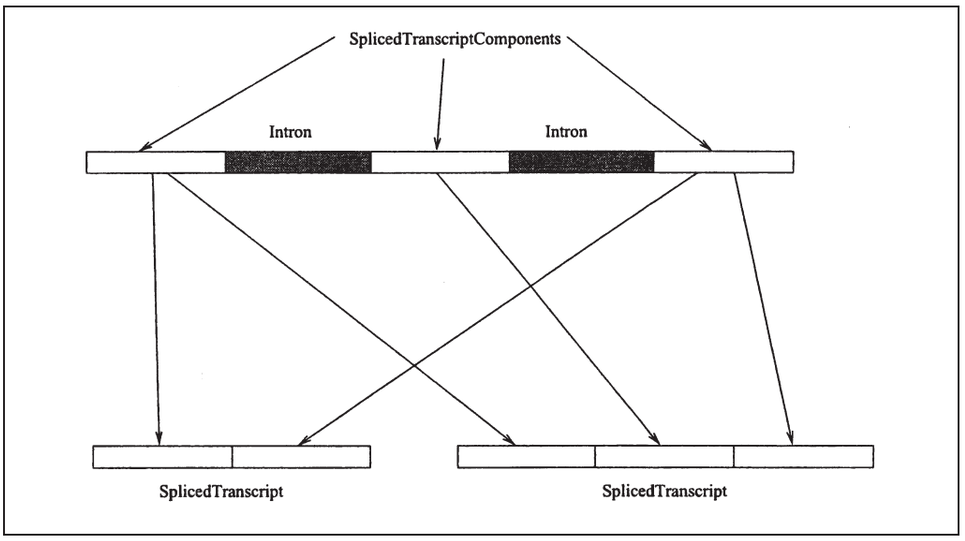

Figure 8: Illustration of alternative splicing. See the section on 'UML model for genome sequences' for further details

platforms is more straightforward and intuitive than mapping onto non-object-based platforms. This is not to say, however, that it cannot be done. In the same way as the section on 'Implementing from ER models' provided some rules to guide the mapping of ER models to the relational model, Blaha and Premerlani[7] provide a comprehensive description of how to map class diagrams onto relational tables. This process is along the same lines as the process for ER models, but the absence of keys in UML models and the tendency for inheritance to be used more widely in object models, often makes the mapping process more involved.

Here, the process of implementing from a UML model is illustrated using the genome schema from Figure 7, which has been implemented using the object database POET.[15]

While POET provides more than one way of implementing database classes, one makes use of Java class definitions, which are preprocessed by the POET system. In essence, POET then allows objects that are instances of the preprocessed classes to be stored in the database. A schema fragment in POET Java is illustrated in Figure 9.

A class from the UML diagram maps into a class in Java. Each attribute from the UML class maps to an attribute in Java. Relationships in UML map onto attributes in one (or both) of the related classes. A complication here is that neither Java nor POET supports bidirectional relationships automatically, so if the implementer wants both roles of a relationship to be represented directly in the Java classes, some effort is required to keep these roles consistent with each other. For example, in Figure 7, if the chromosomes attribute of Genome is to be consistent with the fromGenome attribute of Chromosome, application programs or class methods must be programmed to support such consistency. Figure 9 also illustrates the use of the POET collection class SetOfObject, which is one of several collection classes provided with POET, following the industry standard for object databases.[16]

A further immediate change in perspective tends to take place when mapping from UML class diagrams onto the constructs of object-oriented programming languages. Although UML class diagrams place quite a substantial emphasis on structural aspects of the data (such as attributes and relationships), it is generally considered good practice to make the structural properties of a program class *private*, and to provide access to such properties only through methods (see, for example, the get and set methods in Figure 9). Languages such as Java have well-defined conventions for defining and naming methods used for accessing structural properties.

DISCUSSION

Conceptual data models are a proven technology, in that they have been in widespread use in numerous software projects for many years. However, they have not been used particularly widely with biological and biochemical data, despite the increasing number and complexity of information resources in these areas. This paper seeks to increase awareness of the role that conceptual models can play in describing and understanding the structure of biological data, and has provided example models using the two most widely used conceptual modelling notations.

UML *v*. ER and other implementational issues

While ER was developed earlier on and for relational databases, UML supports object-oriented design better

The conceptual modelling notations illustrated in this paper, namely the ER model and UML class diagrams, were developed at different times for somewhat different purposes. The ER model was developed to support the design of database schemas, in particular schemas for relational databases. As such, ER models have reasonably direct and systematic mappings onto relational databases for implementation,[2] and support modelling notions that are familiar from the relational model (eg

```
// Import statements provide access to existing functionality.  The OGMG
// imports give access to database functionality relating to the ODMG
// Object Database standard
import org.odmg.DCollection.*;
import com.poet.odmg.*;
import com.poet.odmg.util.*;
...

public class Genome
{
    private    String name;
    private    String organism;
    private    String source;
    private    double size;
    private    SetOfObject chromosomes;

    // The constructor function is used to create new instances and to provide
    // initial values to attributes
    public Genome(String a_name, String an_organism, String a_source, double a_size)
    {
        name = a_name;
        organism = an_organism;
        source = a_source;
        size = a_size;
        chromosomes = new SetOfObject();
    }

    // The following accessor methods allow reading or updating of stored
    // properties of the class.
    public String getName()
    {
        return name;
    }

    public void setName(String a_name)
    {
        name = a_name;
    }

    public SetOfObject getChromosomes()
    {
        return chromosomes;
    }

    public void setChromosomes(SetOfObject a_chromosomelist)
    {
        chromosomes = a_chromosomelist;
    }

    public void addChromosome(Chromosome a_chromosome)
    {
        chromosomes.add(a_chromosome);
    }
}

public class Chromosome
{
    private String number;
    private int size;
    private String sequence;
    private Genome fromGenome;
    private SetOfObject ChromFragments;
    ...
}
```

Figure 9: Mapping of Genome and Chromosome to POET Java classes

keys). By contrast, UML class diagrams are only one of a collection of diagrams that together support the object-oriented design of complete applications. As class diagrams are not targeted at any specific category of application, mapping of these diagrams onto implementation platforms can be less direct or systematic than in the

narrower context within which ER is used, but it is often straightforward to map class diagrams onto object-oriented implementation platforms.

While there is now a standard definition of UML, there are many ER proposals, which themselves differ significantly, eg in terms of the ways that inheritance is supported. In this paper we have thus taken the view that it is appropriate to choose and describe a specific ER proposal, but not to elaborate on the ways this specific proposal differs from UML. We have also deliberately avoided detailed discussion of how behaviour is modelled in UML, as this is a large area for which there is no room in the paper.[5]

Outlook

Developing conceptual models is not straightforward, which in turn reflects the fact that obtaining a clear understanding of the semantics of a piece of data is not always easy. Obtaining a clear and agreed view of the data in a domain is challenging, as different people will see things in different ways, use terminology differently and emphasise different features. The situation is further complicated since biosciences are non-axiomatic and the views on the same or similar concepts vary strongly among different, although closely related, communities. However, conceptual models can be helpful in developing, making explicit and communicating clear and detailed descriptions of data that is available or that is about to be produced.

It is hoped that this paper can extend the use of conceptual models within bioinformatics, and thus ease the currently growing problems with managing and sharing biological data.

Acknowledgements

We are pleased to acknowledge the support of the BBSRC/EPSRC Bioinformatics Initiative in funding work on genome information management at Manchester.

References

1. Baxevanis, A. D. (2000), 'The molecular biology database collection: an online compilation of relevant database resources', *Nucl. Acids Res.*, Vol. 28, pp. 1–7.
2. Elmasri, R. and Navathe, S, (2000), 'Fundamentals of Database Systems', 3rd edn, Addison-Wesley, Reading, MA.
3. Chen, P. (1976), 'The entity relationship model: Toward a unified view of data', *ACM Trans. Database Systems*, Vol. 1, pp. 9–36.
4. Booch, G. (1991), 'Object-oriented Design with Applications', Benjamin/Cummings, Redwood City, CA.
5. Booch, G., Rumbaugh, J. and Jacobson, I., Eds (1999), 'The Unified Modelling Language User Guide', Addison-Wesley, Reading, MA.
6. Rumbaugh, J., Blaha, M., Premerlani, W. *et al.*, Eds (1991), 'Object-oriented Design and Modelling', Prentice-Hall, Englewood Cliffs, NJ.
7. Blaha, M. and Premerlani, W. Eds (1998), 'Object-oriented Modelling and Design for Database Applications', Prentice-Hall, Englewood Cliffs, NJ.
8. Attwood, T., Croning, M., Flower, D. *et al.* (2000), 'Prints-s: The database formerly known as prints', *Nucleic Acids Res.*, Vol. 28, pp. 225–227.
9. Bairoch, A. and Appweiler. R. (2000), 'The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000', *Nucleic Acids Res.*, Vol. 28, pp. 45–48.
10. URL: http://argouml.tigris.org
11. Robbins, J., Hilbert, D. and Redmiles, D. (1997), 'ARGO: A design environment for evolving software architectures', in 'Proc. ICSE', ACM Press, pp. 600–601.
12. Gray, P., Paton, N., Kemp, G. and Fothergill, J. (1990), 'An object-oriented database for protein structure analysis', *Protein Eng.*, Vol. 4(3), pp. 235–243.
13. Paton, N., Khan, S., Hayes, A. *et al.* (2000), 'Conceptual modelling of genomic information', *Bioinformatics*, Vol. 16(6), pp. 548–557.
14. Hu, J., Mungall, C., Nicholson, D. and Archibald, A. (1998), 'Design and implementation of a corba-based genome mapping system prototype', *Bioinformatics*, Vol. 14(2), pp. 112–120.
15. URL: http://www.poet.com
16. Cattell, R. and Barry, D. (2000), 'The Object Database Standard: ODMG 3.0', Morgan Kaufmann, San Diego, CA.