# 4

# The geodatabase

## 4.1 Introduction

This chapter describes the way that spatial and attribute data are structured and stored for use within a GIS. It provides the necessary information about data models and database design to enable archaeologists unfamiliar with computer databases to make appropriate decisions about how best to construct a system that will work well and efficiently.

A database is a collection of information that is structured and recorded in a consistent manner. A card catalogue that records information about archaeological sites, such as their location and date, is as much a database as a full-fledged web-searchable digital sites and monuments record. Digital databases differ from their paper counterparts mainly in that they are dependent on database software for searching and retrieving records. The complexity of the data structure will also be increased as digital databases are often broken into several different related files. This reduces the amount of duplicated information in a database, improves access speed and also enables the retrieval of small subsets of data rather than complete records. Software that is used to store, manage and manipulate data is referred to as a *Database Management System* (DBMS). The objectives of a DBMS are to store and retrieve data records in the most efficient way possible, from both the perspective of the overall size of the database and also the speed at which that data can be accessed.

The technology of DBMS is a major research focus in computer science. There is consequently a vast and growing literature on database method and theory, but unless one is interested in designing and writing database programs from scratch it is possible to remain largely ignorant of these details. More usefully we can briefly note the four specific functions that a DBMS should provide (cf. Burrough and McDonnell 1998, p. 50), namely: (i) quick access to, and the ability to select subsets of data, potentially by several users at the same time; (ii) a facility for inputting, editing and updating data; (iii) the ability to define and enforce rules to ensure data accuracy and consistency; (iv) the ability to protect data against unintentional or malicious destruction.

A GIS needs to store, manage and retrieve both geographical and attribute data (the combination of which is collectively referred to as a 'geodatabase'). Many GIS programs provide tools for the management of attribute as well as geographical data, although it is also common to use a separate DBMS program, such as Microsoft Access or MySQL, to manage attribute data. The first part of this chapter reviews

Table 4.1 *A flat-file database*

| sherd_id | sherd_class | temper | diameter (mm) | site_id | earliest date (BP) | latest date (BP) | area ($m^2$) |
|---|---|---|---|---|---|---|---|
| 1 | rim | fine sand | 88 | 1001 | 1200 | 1100 | 8045 |
| 2 | body | fine sand | | 1001 | 1200 | 1100 | 8045 |
| 3 | base | grog | | 1001 | 1200 | 1100 | 8045 |
| 4 | body | vegetable | | 1004 | 900 | 700 | 410 |
| 5 | rim | vegetable | 47 | 1004 | 900 | 700 | 410 |
| 6 | body | fine sand | | 1006 | 1000 | 800 | 8900 |
| 7 | base | coarse sand | | 1007 | 800 | 400 | 7980 |
| 8 | rim | shell | 140 | 1007 | 800 | 400 | 7980 |
| 9 | rim | grog | 134 | 1009 | 200 | 100 | 1200 |

the ways that attribute data may be stored and managed in a DBMS, and the second part examines the storage methods for raster and vector datasets.

### 4.1.1 *Data models: from flat file to relational*

The simplest form of database consists of a single table of data, in which each column contains a field and each row a record. Tables such as these are referred to as *flat-file databases* because there is no depth to the data. All the data are stored in one location and there are no linkages between these records and those in other data tables. Flat-file systems do have some advantages over other logical models as they are very easy to maintain, the data are readily available, and search and find operations are computationally simple. However, they have a major disadvantage which precludes their use for all but the simplest databases, which is that they have no facility for dealing with data that may be redundant. For example, many records in the pottery sherd database shown in Table 4.1 contain duplicated information about the site from which the sherd comes.

This introduces a high level of duplication in the data. Furthermore, some data cells are empty – for example, the field *diameter* is only filled in for rim sherds. The result is both an unnecessarily large and redundant data structure in which a large number of cells either contain repeated data or are null. Flat-file tables are thus only appropriate for the simplest databases, as they soon become unwieldy when information about more than one entity needs to be recorded, as in the above example. An elegant solution to this problem, first defined by E. F. Codd (1970) in what became known as the *Relational Model*, is to break the database into separate tables that each contain a coherent package of information. For example, creating three new tables each containing information on *sites*, *pottery* and *rims* reduces the amount of repeated data and blank entries. The disadvantage is that the system must then manage three tables instead of one, so some mechanism is then needed to extract information from all tables to answer questions such as 'what different types of temper are found on rim sherds from sites with dates before 1000 BP?'.

The advantages of the relational model are its simple tabular structure, uncomplicated relationships, and its associated powerful and versatile query language, called SQL (pronounced 'sequel'). Although the relational model may seem complex to the uninitiated, in practice it is a simple and versatile way of managing large datasets.

The principal component of a *relational database* is a table of data, referred to as a *relation*, which consists of a set of records that contain a number of different fields shared by all records. Each record must be distinct from every other via a unique identifier (such as an id number), referred to as the *primary key*. If no single field contains a unique entry, then two or more fields can be used to define a primary key, in which case it is called a *composite key*. Tables must also meet certain conditions defined by the relational concept of *normalisation* (for examples, see Beynon-Davies 1992, pp. 30–42). This means that records with the same repeating values should be placed into separate tables and be defined by their own primary key. The process of breaking a set of data into separate tables and defining the links between them is referred to as *normalisation*.

For example, in Table 4.1, values for *site_id*, *early date*, *late date* and *area* always occur in groups and should therefore be recorded in a separate table (called 'sites'). The rules of normalisation also require that dependent values in the remaining fields are separated; and so, in our example, *diameter* must also be placed into a different table ('rim measurements') than the *sherd_id*, *sherd_class* and *temper* fields, because it is dependent on a *sherd_class* value of 'rim'. In this case, the rims table needs to inherit the *sherd_id* data so that it is possible to determine which sherd each rim measurement belongs to. The normalised tables and their primary keys are shown in Table 4.2.

Relationships between tables are then created by defining links between a primary key and its equivalent field in another table, referred to as a *foreign key*. In our example, we need therefore to create a new field in the pottery table called *site_ id* that contains the id number of the site where that sherd was found. This field then acts as the foreign key and allows the primary key (*site_id*) in the site table to establish a relation with the pottery table. The variable *sherd_id*, which is present in both the pottery and rim tables, can be used to link those two tables together. The links tables can take the form of either a *one-to-one*, *one-to-many* or *many-to-many* relationship, depending on the number of records referred to by the primary–foreign key link. In this example, the relationship between 'sites' and 'pottery' is one-to-many (as one site may have many sherds) and the relationship between 'pottery' and 'rim measurements' is one-to-one (as no sherd will have more than one rim record – note we are not considering the possibility of conjoining sherds or rims here!). For reasons of storage efficiency and relational coherence, many-to-many relationships need to be turned into one-to-many relationships via the construction of intermediary tables. Normalisation is covered in greater detail in a number of accessible publications, including the University of Texas at Austin's Information Technology Service Data Modelling Pages[1] and Hernandez (2003).

[1] www.utexas.edu/its/windows/database/datamodeling/rm/overview.html.

Table 4.2 *The three normalised tables: sites (upper), pottery (middle) and rim measurements (lower). Primary keys are defined by a †, the foreign key by a ‡*

| site_id† | early date (BP) | late date (BP) | area ($m^2$) |
|---|---|---|---|
| 1001 | 1200 | 1100 | 8045 |
| 1004 | 900 | 700 | 410 |
| 1006 | 1000 | 800 | 8900 |
| 1007 | 800 | 400 | 7980 |
| 1009 | 200 | 100 | 1200 |

| sherd_id† | sherd_class | temper | site_id‡ |
|---|---|---|---|
| 1 | rim | fine sand | 1001 |
| 2 | body | fine sand | 1001 |
| 3 | base | grog | 1001 |
| 4 | body | vegetable | 1004 |
| 5 | rim | vegetable | 1004 |
| 6 | body | fine sand | 1006 |
| 7 | base | coarse sand | 1007 |
| 8 | rim | shell | 1007 |
| 9 | rim | grog | 1009 |

| sherd_id† | diameter (mm) |
|---|---|
| 1 | 88 |
| 5 | 47 |
| 8 | 140 |
| 9 | 134 |

The retrieval of data from a relational system using SQL involves defining the relations between entities and the conditions under which certain attributes should be selected. This is discussed in further detail in Chapter 7, where spatial data query methods are also reviewed. Some GIS programs, such as ESRI's ArcInfo and ArcView, have embedded DBMSs that allow basic relationships between tables to be defined and queried. While embedded management systems offer good functionality for data retrieval from databases with simple structures, linking a GIS to a database such as PostgreSQL, MySQL, Oracle or Microsoft's SQL Server, permits the full use of the relational model and SQL for managing and querying attribute data.

Even if the GIS software does provide basic DBMS facilities for managing attribute data there are three good reasons for using an external DBMS instead: (i) an external DBMS will be able to cope with more complex multi-table data relations; (ii) data retrieval will be faster with large volumes of data; (iii) it will permit more complex queries than the basic search and retrieval functions provided

in many integrated DBMS found in GIS programs. Nearly all GIS offer facilities for linking spatial data objects to attribute data held in an external relational database, often through SQL and database technologies such as Open DataBase Connectivity (ODBC).[2] SQL is covered in more detail in Chapter 7, although it is worth noting here that many database systems provide graphical query tools so that the user is not required to write raw SQL.

### Beyond the relational model: object-orientated systems

Recent post-relational database theory has seen the development of *object-orientated* (or OO) databases. Object-orientated (OO) databases share some of the philosophy behind OO programming languages such as Java. This includes concepts such as packaging data and functionality (behaviour) together into modular units called objects; the *encapsulation* of objects so that they can only change their own state, not the state of others in the system; and *inheritance*, such that new objects may inherit the properties of other existing objects. Object-orientated databases thus differ significantly from relational databases by including information about the behavioural characteristics of entities, which are referred to as its *methods*. The combined table and attributes, including rules defining their behaviour, is referred to as an *object*, and similar objects are grouped together in *object classes*. For example, as applied to GIS, 'burial mounds' and 'causeway enclosures' may be modelled as objects in an OO data model rather than simply entities in a layer. In the OO model, information about the entities are encapsulated within the object, whereas in the relational model, information is obtained via linked records in an attribute table. Embedding the relevant information within the object itself removes the need to search for properties in additional tables, and also provides the possibility of including rules that stipulate the behaviour of the object. The process of retrieving data is similar to the relational model, although in OO databases a derivative of SQL called *Object Query Language* (OQL) is used.

A fully OO GIS that permits a more flexible set of object rules and behaviours is not yet available for the desktop, and has not made a significant impact in archaeology. Commercial OO GIS software systems include Smallworld,[3] which has been designed for organisations with significant asset-management requirements. Note, however, that the geodatabase structure used in ArcGIS is partially OO because vector entities do have defined behaviours and rules to, for example, control the form of topological relationships between vector objects.

## 4.2 Designing a relational database for attribute data

The users of GIS most often become involved in database theory when they need to design a database to provide storage and access to a set of attribute data. For example, if you have been collecting data from an archaeological survey, you may

[2] http://en.wikipedia.org/wiki/ODBC.
[3] www.gepower.com/prod_serv/products/gis_software/en/core_spatial.htm.

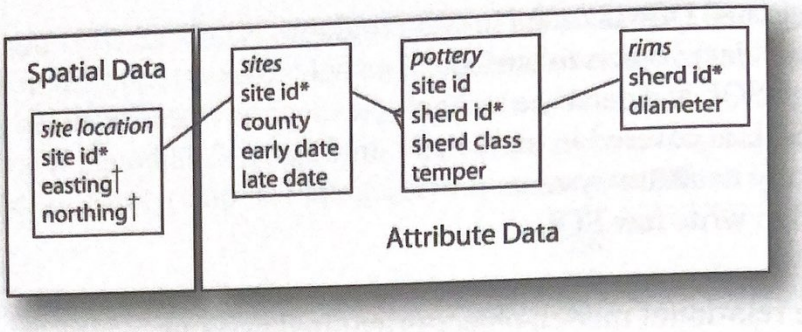



Fig. 4.1 An entity–relationship (E–R) diagram (boxes = entities and attributes, arrows = relationships) for a geodatabase. Asterisks indicate unique values (primary keys). † indicates a spatial attribute used for GIS implementation.

need to create a system to store information about the types of environmental and archaeological data that were collected that can then be linked to a GIS that contains information about the spatial location of, for example, testpits, survey areas, transects and archaeological sites.

The design of a database typically consists of four stages: designing a conceptual model, implementing the model, constructing an external interface (or 'front end'), and deciding on the system for the physical storage of data.

**Conceptual model** Database design often first begins at the level of the *conceptual model*. This is the most abstract and arguably the most important level of database design, as it concerns the entities (such as sites or artefacts) and their attributes (such as size or type), and their relationships with other entities (such as between artefacts and sites). The term 'entity' in this sense refers to any phenomenon, physical or abstract, that can be described and distinguished from other phenomena (Jones 1997, p. 164). For example, 'artefacts' and 'archaeological sites' are two different types of entities. When designing an attribute database, a useful first step is to make a list of entities and their attributes and the nature of the relationships between them. This can then be modelled using an entity–relationship (E–R) diagram (Chen 1976). The example provided in Fig. 4.1 shows the relationship between the entities discussed previously: 'sites', 'pottery', 'rim measurements'. Note that in this instance a fourth relation has been added that contains the spatial locations of the sites, which would usually be stored in a GIS.

**Implementation** Following the conceptual design, the next stage is typically to populate the database with a sample of information to ensure that the relations function in the way that is intended, and that it is possible to extract information in the combinations required for subsequent analysis (e.g. using the same example as above, is it possible to extract the locations of sites that possess sand-tempered pottery?).

**External interface** Assuming the database works, a common next step is to create the external interface, although with smaller single-user systems this may not be necessary. In larger multiuser systems having a 'front-end' is desirable as it provides users with a set of tools to enter and search for data. In addition, as there may be several different categories of user, the external model can be customised to present only the necessary subsets of data to a particular user. For example, the external level of a sites and monuments record database could be designed so that the data

seen by archaeologists and municipal planners are organised differently to the data made available to members of the public. It is at this level that an entire database, or sections of it, can be 'locked' so that data cannot be seen or modified by non-administrative users. Facilities for speeding up data entry and reducing input error can also be introduced at the external level, which might include drop-down menus, or 'radio buttons' offering simple yes/no options for data entry. Some database packages provide tools for the design of front-end 'forms', and many provide tools to enable the use of web browsers to view, query and edit data.

**Physical storage** Some decisions can be made by users regarding where and how information should be stored on hard-disks. GIS and database programmers expend considerable energy on increasing the efficiency of their programs by designing them so that the information they store can be quickly retrieved by the computer. For database end-users, the specifics of the physical model (such as how often a database writes to a disk, how it allocates virtual memory, and how it keeps track of changes to a database) may remain hidden, but the speed of the database and its ability to process requests for information efficiently is ultimately linked to how the database deals with the physical storage of data. A small desktop GIS will probably store data on the same physical device (hard-disk) as the GIS software; but data for large, multiuser, networked systems will likely be spread over many disks and, to prevent data loss in the event of disk failure, duplicated in what is known as a redundant array of inexpensive disks (RAID). If you are designing a database for more than one user, then it is worth investigating network storage and RAID systems in more detail – an afternoon's research on the Internet will provide ample information about available options.

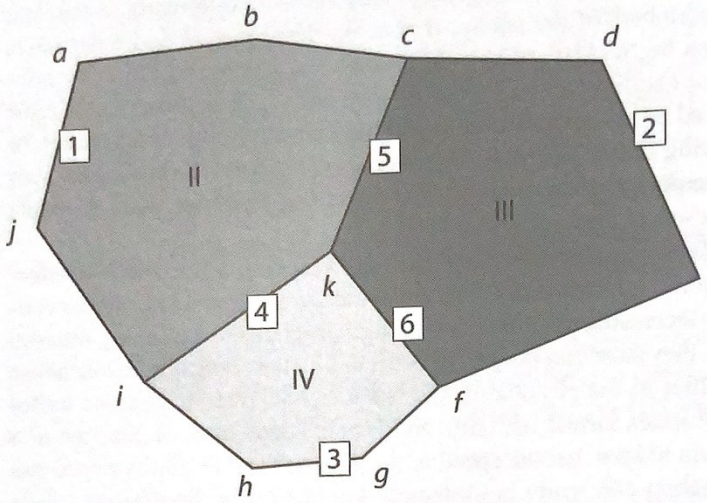
## 4.3 Spatial data storage and management

As with attribute data, a GIS also requires a database for the storage and manipulation of spatial objects. However, as the data structures for spatial objects are embedded into the GIS program and cannot themselves be manipulated, the spatial object DBMS remains largely 'behind the scenes'. Although it will do its job without complaint or need of user intervention, it is nevertheless worth briefly reviewing the different data structures used to store and manipulate vector and raster datasets in order to reinforce the primary importance of the DBMS in GIS.

### 4.3.1 *Vector data storage*

In the vector data structure, all objects, whether points, lines or polygons, are represented using one or more $x$, $y$-coordinate pairs. A set of points, for example, is recorded as a list of coordinate pairs, each of which is given an identification number. Polylines may be stored as a sequence of vertices terminated by nodes and are also given identification numbers. Polygons can be represented in the same manner as polylines and their final node must be the same as their first node in order to close the loop.

This method of data storage is used by CAD and some early GIS programs (e.g. ArcView 3.2), but it is not very efficient because it does not take into account the topological relationships between vector objects. When two polygons share a common boundary, for example, the vertices of the common boundary need to be recorded for each polygon, which results in a very data-hungry system. Furthermore,




| vertices | location |
|---|---|
| a | x,y |
| b | x,y |
| c | x,y |
| d | x,y |
| e | x,y |
| f | x,y |
| g | x,y |
| h | x,y |
| i | x,y |
| j | x,y |
| k | x,y |

| polygons | arcs |
|---|---|
| I | 1,2,3 |
| II | 1,5,4 |
| III | 5,2,6 |
| IV | 4,6,3 |

| arcs | start | end | vertices | l-poly | r-poly |
|---|---|---|---|---|---|
| 1 | i | c | j,a,b | I | II |
| 2 | c | f | d,e | I | III |
| 3 | i | f | h,g | IV | I |
| 4 | i | k | | II | IV |
| 5 | k | c | | II | III |
| 6 | k | f | | III | IV |

Fig. 4.2 Three topologically related polygons and a simple arc-node storage structure.

there is no way to store relationships – such as adjacency or overlap – between vector objects, so they must be computed at need. A more efficient solution uses a relational model for data organisation and is referred to as *arc-node data structure* (Fig. 4.2). This method structures, stores and references data in a relational DBMS so that points (nodes) construct polylines (arcs) and polylines construct polygons. The directionality of the arcs is recorded in this storage method by a field that defines the beginning and end node plus the vertices in between. Arcs that are used to define polygon boundaries also record which polygons lie on the left and right of the arc. This enables adjacency queries on polygons to be quickly answered with reference to the arc table rather than using a computationally more complex spatial query method.

Any spatial queries performed on these objects can then be answered by reference to these tables. For example, questions such as 'what is the area of polygon III?' can be answered by using the vertices that define arcs 5, 2 and 6 to build triangles, for which areas can be calculated, the sum of which provides the polygon's area. Topological queries, such as 'which polygons are adjacent to polygon IV?' can be answered by finding the arcs that define polygon IV, then using the left and right polygon field for those arcs to define adjacency to other polygons (i.e. polygons II and III).

Although the arc-node structure is arguably more efficient it can limit flexibility, so some recent GIS databases, such as that used in ArcGIS, have moved to a different topological structure managed by what is called a Spatial Database Engine. In this system, the arc-node structure for each object is stored separately in what is termed a Feature Class object. Topological relationships are thus computed at-need rather than stored as an inherent part of a spatial database, although some forms of relationships can be predefined and stored in Feature Datasets, so that, for example, changes in the boundary of one object will automatically result in

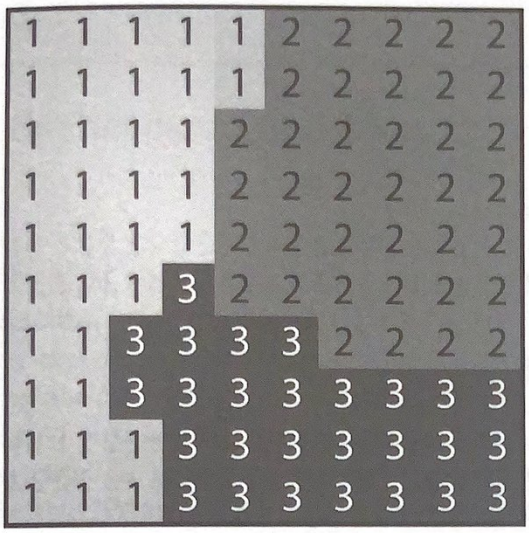



Fig. 4.3 A 10 × 10 raster grid with cell values overlay.

changes to those others that share the same boundary. This permits a high degree of flexibility for defining relationships between spatial entities, but is also computationally more intensive and requires additional memory for recording the structures and relationships between spatial objects.

In Chapter 7 we discuss the mechanics of a broader range of spatial queries that depend on topological relationships.

### 4.3.2 *Raster data storage*

Consider the raster map in Fig. 4.3. A simple method for storing such a grid of $n$ rows by $m$ columns is to record the $x$, $y$-coordinate pair of each cell together with its associated attribute value. This, for obvious reasons, is neither very practical nor efficient: there is no mechanism for recording the size of cells and large matrices will have massive memory overheads because every pixel requires three pieces of data (i.e. its $x$, $y$-coordinate and its value). Advances in raster storage systems in the 1980s and early 1990s saw the development of much more efficient methods of storing raster data (Burrough and McDonnell 1998, pp. 51–57). All but the most basic raster storage systems now possess a *header*, which defines the number of rows and columns, the cell size, the geodetic datum, the projection and reference system, and the location of a corner pixel followed by a sequence of cell values (Fig. 4.4).

In some GIS programs, such as Idrisi, the header information is stored as a separate file that shares a name with the file that stores cell values. Using a header results in more efficient data storage because the cell coordinates need not be stored, but large files can still occur with data consisting of long real number strings, as is the case with elevation data (Burrough and McDonnell 1998, p. 51). One method to reduce file size that works well for data that have some degree of positive spatial autocorrelation (Chapter 6) is to record the differences between one cell and the

columns: 10
rows: 10
cell size: 20
minimum *x*: 320
minimum *y*: 153
reference system: metres
1 1 1 1 1 2 2 2 2 2 1 1 1 1 1 2 2 2 2 2 1 1 1 1 2 2 2 2 2 2 1 1 1 1 2 2 2
2 2 2 1 1 1 1 2 2 2 2 2 2 1 1 1 3 2 2 2 2 2 2 1 1 3 3 3 3 2 2 2 2 1 1 3 3
3 3 3 3 3 3 1 1 1 3 3 3 3 3 3 3 1 1 1 3 3 3 3 3 3 3

Fig. 4.4 A typical raster storage file.

1 1 1 1 1 2 2 2 2 2 1 1 1 1 1 2 2 2 2 2 1 1 1 1 2 2 2 2 2 2 1 1 1 1 2 2 2
2 2 2 1 1 1 1 2 2 2 2 2 2 1 1 1 3 2 2 2 2 2 2 1 1 3 3 3 3 2 2 2 2 1 1 3 3
3 3 3 3 3 3 1 1 1 3 3 3 3 3 3 3 1 1 1 3 3 3 3 3 3 3

*becomes:*

1 5 2 5 1 5 2 5 1 4 2 6 1 4 2 6 1 4 2 6 1 3 3 1 2 6 1 2 3 4 2 4 1 2 3 8 1
3 3 7 1 3 3 7

Fig. 4.5 An example of raster file compression using the RLC method.

next. In this way an elevation matrix that contained the sequence {1021.34, 1022.01, 1023.00} could be recorded as {1021.34, 0.67, 0.99} and would thus reduce the number of stored digits in the raster file. Methods that rely on a form of *data compression* may also offer solutions for particular types of data. For example, an array of numbers can be compressed using a system of *run length compression* or RLC. This involves substituting repeating numbers with an integer that defines how many times the number occurs in sequence. RLC would reduce the data array in Fig. 4.4 to the second array shown in Fig. 4.5.

Other forms of compression use regular shapes such as squares, rectangles, triangles or hexagons that *tessellate* in a regular pattern and these are then assigned to data of uniform value, so that only the size and location of the shape and the shared cell value need to be recorded. When square blocks are used the process is often referred to as *quadtree* data encoding. Like the more common RLC method, each may offer some space savings at the expense of computational time.

In practice, a user need not be overly concerned with the specifics of the raster compression system used in any given GIS, as most are able to both import and export non-compressed raster data. The specific format and headers used to read non-compressed data are often particular to an individual GIS, although these details will be included in the software's documentation. Readers interested in additional information on raster data storage and compression are referred to Burrough and McDonnell (1998, pp. 51–57).