Statistics and Computing (2000) **10,** 325–337

WinBUGS – A Bayesian modelling framework: Concepts, structure, and extensibility

DAVID J. LUNN*, ANDREW THOMAS*, NICKY BEST*
and DAVID SPIEGELHALTER†

*Department of Epidemiology and Public Health, Imperial College School of Medicine, Norfolk Place, London W2 1PG, UK
d.lunn@ic.ac.uk, a.thomas@ic.ac.uk, n.best@ic.ac.uk
†Medical Research Council Biostatistics Unit, Institute of Public Health, Robinson Way, Cambridge CB2 2SR, UK
david.spiegelhalter@mrc-bsu.cam.ac.uk

WinBUGS is a fully extensible modular framework for constructing and analysing Bayesian full probability models. Models may be specified either textually via the BUGS language or pictorially using a graphical interface called DoodleBUGS. WinBUGS processes the model specification and constructs an object-oriented representation of the model. The software offers a user-interface, based on dialogue boxes and menu commands, through which the model may then be analysed using Markov chain Monte Carlo techniques. In this paper we discuss how and why various modern computing concepts, such as object-orientation and run-time linking, feature in the software's design. We also discuss how the framework may be extended. It is possible to write specific applications that form an apparently seamless interface with WinBUGS for users with specialized requirements. It is also possible to interface with WinBUGS at a lower level by incorporating new object types that may be used by WinBUGS without knowledge of the modules in which they are implemented. Neither of these types of extension require access to, or even recompilation of, the WinBUGS source-code.

Keywords: WinBUGS, BUGS, Markov chain Monte Carlo, directed acyclic graphs, object-orientation, type extension, run-time linking

1. Introduction

WinBUGS is the current, windows-based, version of the BUGS software described in Spiegelhalter *et al.* (1996b). (BUGS is an acronym for Bayesian inference Using Gibbs Sampling.) It is a user-friendly, 'point-and-click' environment that makes accessible state-of-the-art statistical methodology (Markov chain Monte Carlo techniques) for analysis of a wide class of Bayesian full probability models.

The conceptual design of the software is based on constructing an internal representation of the probability model that is analogous to the way in which it may be visualized as a graphical model (e.g. Spiegelhalter 1998). In graphical modelling, each quantity in the model is represented by a node and nodes are connected by lines or arrows to show direct dependence. The details of distributional assumptions and deterministic relationships are 'hidden' to clarify the qualitative nature of the model. Many useful properties of the model can be derived simply from this abstract representation. This has led very naturally to an object-oriented approach to the software's design.

Object-oriented programming involves the construction of a hierarchical collection of *type* definitions, comprising a set of concrete types at the top with various levels of abstraction beneath. An object is an instance of a concrete type but it may be treated as being of a more basic type. Thus it is possible to write very general procedures that operate abstractly on all objects – the hierarchy is accessed at a level that is appropriate for the purposes of the operation. In WinBUGS objects are particularly useful for representing the various nodes in a graphical model.

WinBUGS has been designed primarily for handling directed acyclic graphs (DAGs; Lauritzen *et al.* 1990, Whittaker 1990) – graphical models where links between nodes are directed and

cycles are not permitted. DAGs represent a series of (realistic) conditional independence assumptions, which allow the full probability model to be factorized into a product of simple local components. Knowledge of each node's parents, i.e. the nodes upon which it directly depends, is sufficient to construct the full model, and so only this information need be incorporated into the internal representation of the model. This internal representation is therefore analogous to the graph itself and easily manageable however complex the model may be.

Statistical analysis of the model is conducted using various simulation methods known as Markov chain Monte Carlo (MCMC; see Gilks, Richardson and Spiegelhalter 1996, for example). The primary technique is Gibbs sampling (Geman and Geman 1984), in which at each iteration a new value for each unobserved stochastic node is sampled from the corresponding parameter's *full conditional distribution*, i.e. its distribution conditional upon all other model parameters and the data. The factorization alluded to above allows this distribution to be constructed merely from knowledge of the node's parents and children (i.e. nodes for which the node of interest is a parent). In this sense the Gibbs sampler is simply a sequence of local computations on the graph.

The structure of the WinBUGS source-code is also analogous to a graphical model, in that it comprises a network of locally communicating components – a component-oriented philosophy (Szyperski 1995) has been adopted. This novel software engineering approach aims to create fully extensible modular systems. Software consists of a number of components that are not linked together until load-time or even run-time. Each software component has a well-defined *interface* that describes the implemented entities that can be used in other components. The component interface is encoded in a machine readable format called a *symbol file*, the use of which allows consistency of the component interfaces to be checked both at compile-time and link-time, thus improving the reliability of the software. Inclusion of new methods and applications is achieved by writing extra components that simply either 'plug-in' to relevant slots in existing modules, or make use of existing modules, without requiring any part of the software to be recompiled.

The aim of this paper is to describe how WinBUGS works and how modern computing concepts, such as object-orientation, modular programming, and run-time linking, are exploited in the design of the software. We hope that it also encourages the reader to consider adopting similar approaches to the solution of complex statistical problems. The structure of the paper is as follows. In Section 2 we describe DAGs and the Bayesian statistical methodology that is particularly suited to their analysis. Section 3 outlines how WinBUGS satisfies the fundamental requirements of general MCMC software, and Section 4 describes briefly how models are specified. The basic concepts of object-oriented programming are described and illustrated in Section 5. In Section 6 we discuss how the software is organised into a hierarchical collection of *subsystems*, each of which has a specific set of responsibilities and comprises a module hierarchy. We pay particular attention to the Graph subsystem, which provides the objects used to construct an internal representation of the model. The various ways of extending the WinBUGS framework are described in Section 7 and a concluding discussion is given in Section 8. Technical details of the key methods bound to objects in the Graph subsystem are given in an Appendix, and full details of model specification and the user-interface can be found in the documentation provided with the software (`http://www.mrc-bsu.cam.ac.uk/bugs/`).

2. The setting

2.1. *Graphical models*

To illustrate the statistical concepts that WinBUGS exploits we consider a simple univariate linear regression model. Suppose we have observations y_i measured at $x_i, i = 1, \ldots, N$, where the x_i are design-points of the experiment, e.g. observation times, and are assumed known. If we suppose that a linear relationship exists then we might assume

$$y_i \sim N(\mu_i, \tau^{-1}), \mu_i = \alpha + \beta x_i,$$

for $i = 1, \ldots, N$. Here α and β are the unknown intercept and gradient parameters respectively, and τ is the *inverse* of the residual variance (also unknown). An alternative representation of this model is the *directed acyclic graph* (DAG; see, for example, Spiegelhalter 1998) shown in Fig. 1, where each quantity in the model corresponds to a *node* and links between nodes show direct dependence. The graph is *directed* because each link is an arrow; it is *acyclic* because by following the arrows it is not possible to return to a node after leaving it.

The notation is defined as follows. Rectangular nodes denote known constants. Elliptical nodes represent either deterministic relationships (i.e. functions) or stochastic quantities,

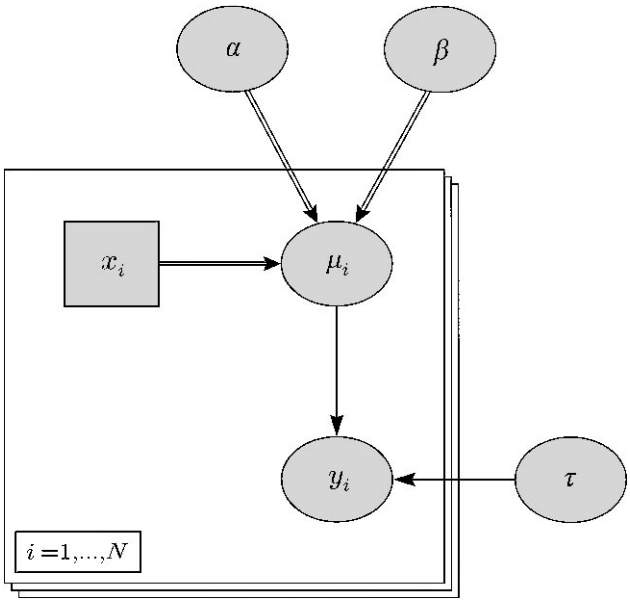

Fig. 1. *Directed acyclic graph for a simple linear regression*

i.e. quantities that require a distributional assumption. (Note that WinBUGS is designed for Bayesian models and so unknown parameters, such as α, β and τ in the above example, are stochastic, because a *prior* distribution must be assigned to each – see Section 2.2.1.) Stochastic dependence and functional dependence are denoted by single-edged arrows and double-edged arrows respectively. Repetitive structures, such as the loop from $i = 1$ to $i = N$, are represented by 'plates', which may be nested if the model is hierarchical.

A node v is said to be a *parent* of node w if an arrow emanating from v points to w; furthermore, w is then said to be a *child* (or *offspring*) of v. We are primarily interested in stochastic nodes, i.e. the unknown parameters and the data. When identifying probabilistic relationships between these, deterministic links are collapsed and constants are ignored. Thus the terms parent and child are usually reserved for the appropriate stochastic quantities. In the above example, the *stochastic parents* of each y_i are α, β and τ, whereas we refer to μ_i and τ as the *direct parents*.

DAGs can be used to describe pictorially a very wide class of statistical models. It is when these models become complicated that the benefits become obvious. DAGs communicate the essential structure of the model without recourse to a large set of equations. This is achieved by abstraction: the details of distributional assumptions and deterministic relationships are 'hidden'. This is conceptually similar to object-oriented programming, a subject discussed in Section 5.

In general, a DAG represents a series of *conditional independence* assumptions: for any node v, if the parents are known then no other nodes provide further information about v, except for descendants of v (the genetic analogy is clear). Thus

$$v \perp\!\!\!\perp \text{non-descendants}[v] \mid \text{parents}[v]$$

where $\perp\!\!\!\perp$ denotes 'is conditionally independent of'. In the above example $y_i \perp\!\!\!\perp y_j \mid \alpha, \beta, \tau$ for $j \neq i$. The conditional independencies expressed through DAGs allow properties of the model to be derived even though no specific probabilistic form has been specified (Lauritzen *et al.* 1990, Whittaker 1990, Spiegelhalter *et al.* 1993). In the following sub-section (2.2) we show how DAG representation greatly facilitates the analysis of arbitrarily complex full probability models.

2.2. *Methodology*

2.2.1. Bayesian statistics

Suppose we have observed data y and unknown parameters θ. The Bayesian approach to statistics is to treat all unknown quantities as random variables and assign a *prior* probability distribution to each. By also specifying a joint probability distribution for the data, i.e. a likelihood, we obtain a full probability model for all observable and unobservable quantities. In order to make inferences about θ we use Bayes' theorem to construct the *posterior* distribution, i.e. the joint distribution of all model parameters conditional on the observed data:

$$p(\theta \mid y) \propto p(y \mid \theta)p(\theta),$$

where, throughout, $p(.\mid.)$ and $p(.)$ denote conditional and marginal probability distributions respectively. Thus the posterior is proportional to the likelihood $p(y \mid \theta)$ multiplied by the prior $p(\theta)$. An excellent introduction to Bayesian data analysis is given by Gelman *et al.* (1995).

2.2.2. Markov chain Monte Carlo

In many realistic modelling situations the joint posterior distribution $p(\theta \mid y)$ is high-dimensional (e.g. $\dim(\theta) = 100$'s or 1000's), complex, and unavailable in closed form. Bayesian inference entails the evaluation of various summaries of the posterior, such as moments and quantiles. This requires integration, with respect to θ, of functions involving $p(\theta \mid y)$; it is these integrals that until recently have rendered Bayesian analysis problematic. Markov chain Monte Carlo (MCMC) methods (see, for example, Gilks, Richardson and Spiegelhalter 1996) alleviate these difficulties. Integrals are evaluated via Monte Carlo simulation from a Markov chain that is constructed so that its stationary distribution is the posterior.

Various algorithms exist for carrying out the required simulations, including Gibbs sampling (Geman and Geman 1984, Gelfand and Smith 1990), which is particularly useful for exploiting conditional independence assumptions (see Section 2.1). The algorithm proceeds by iterative simulation from the full conditional distributions of each unknown stochastic quantity given the current values of all other terms (nodes) in the model. Numerous applications of Gibbs sampling and other MCMC techniques can be found in the literature. (A number of these make use of either the BUGS or WinBUGS software.) The methodology has now been applied in many fields of research, such as medicine (Ayanian *et al.* 1998), financial economics (Pitt and Shephard 1999), spatial epidemiology (Wakefield and Morris 1999), and pharmacokinetics (Lunn and Aarons 1998).

2.2.3. DAG factorization

Let V denote the set of all nodes in a DAG. It can be shown (Lauritzen *et al.* 1990) that

$$p(\theta \mid y) \propto p(\theta, y) = p(V) = \prod_{v \in V} p(v \mid \text{parents}[v]). \qquad (1)$$

Not only is this factorization convenient in the sense that it enables arbitrarily complex full probability models to be specified in terms of simple *local* components, but it also makes identification of full conditional distributions straightforward. Let $V \backslash v$ denote 'all elements of V except v'. The full conditional $p(v \mid V \backslash v)$ is proportional to the product of terms in $p(V)$ that contain v:

$$p(v \mid V \backslash v) \propto p(v \mid \text{parents}[v]) \times \prod_{w \in \text{children}[v]} p(w \mid \text{parents}[w]). \qquad (2)$$

Table 1. *Sampling method hierarchy used by WinBUGS. Each method is only used if no previous method in the hierarchy is appropriate*

Target distribution	Sampling method
Discrete	Inversion of cumulative distribution function (trivial)
Closed form (conjugate)	Direct sampling using standard algorithms
Log-concave	Derivative-free adaptive rejection sampling (Gilks 1992)
Restricted range	Slice sampling (Neal 1997)
Unrestricted range	Metropolis-Hastings (Metropolis *et al.* 1953, Hastings 1970)

The first term, i.e. $p(v \mid \text{parents}[v])$, is referred to as the prior component and the second, i.e. the product over children$[v]$, is called the likelihood component. Often these are 'compatible' in the sense that the full conditional distribution can be derived analytically, in which case samples can be generated efficiently using the appropriate specialized random number generator (Ripley 1987). For example, a gamma prior for τ in Fig. 1 would combine with the normal likelihood to give a gamma full conditional for τ. (See Spiegelhalter *et al.* 1996b, pp. 17, 21, for tables of distributions and their so-called *conjugate* priors.) If for any node the full conditional distribution is not available in closed form then samples may be obtained by using (2) within a more general sampling method, such as adaptive rejection sampling (Gilks and Wild 1992) or a Metropolis-Hastings algorithm (Metropolis *et al.* 1953, Hastings 1970).

3. Requirements of MCMC software

Here we identify three fundamental requirements of general MCMC software and describe briefly how they are satisfied by WinBUGS.

3.1. *A wide class of models*

The factorization property in equation (1) allows arbitrary DAG structures to be specified simply by stating the relationship between each node and its direct parents. WinBUGS provides, via the BUGS language, a wide range of distributions and functions (easily incremented – see Section 3.3 below) that can be used to specify these relationships. From this model specification an easily manageable object-oriented representation of the model is constructed. This internal representation is also a collection of simple local components – it comprises an indexed list of objects, each of which corresponds to a particular node in the DAG and can access the objects that correspond to its direct parents.

3.2. *Efficient sampling*

WinBUGS is an 'expert system' (although no explanation facilities are required) that attempts to utilise the most appropriate sampling scheme for each stochastic node. In cases where a node's full conditional distribution is available in closed form WinBUGS can usually identify that closed form and implement the appropriate specialized sampling method. Where a node's full conditional is not (known to be) available in closed form the software examines the circumstances and chooses an appropriate general sampling method. Table 1 shows, in order of precedence, the five types of sampling method currently used by WinBUGS and the types of distribution for which they are employed.

The way in which object-orientation facilitates this 'expert system' behaviour is discussed in Section 6 and in the Appendix.

3.3. *Extensibility*

General software should be extensible, i.e. users should be able to add to its capabilities. Users of WinBUGS may extend the BUGS language by incorporating new distributions and/or new functions into the system. It is also possible to incorporate new MCMC sampling techniques and to write user-interfaces that deal efficiently with specific types of model. None of these types of software extension require access to, or even recompilation of, any part of the WinBUGS source-code. This is due to the component-oriented design of the software and the fact that it makes use of a powerful computing concept known as *run-time linking*, where software components are loaded *on demand*. Extensibility is further discussed in Section 7.

4. Specification and analysis of graphical models

In WinBUGS models may be specified either textually or graphically. (In both cases, data and initial values for the unknown parameters are specified either in an S-Plus-like format or in rectangular array format.) Textual specification is achieved using a declarative language known as the BUGS language. The following code defines the likelihood for the regression problem depicted in Fig. 1 (priors are omitted for brevity).

```
for (i in 1:N) {
    y[i] ~ dnorm(mu[i], tau)
    mu[i] <- alpha + beta * x[i]
}
```

As is customary the `~` notation denotes 'is distributed as'. This must always be followed by a distribution identifier, in this case `dnorm(.,.)`. (Note that in WinBUGS normal distributions are parameterised in terms of precisions rather than variances.) The `<-` notation is read 'gets' and identifies logical (or

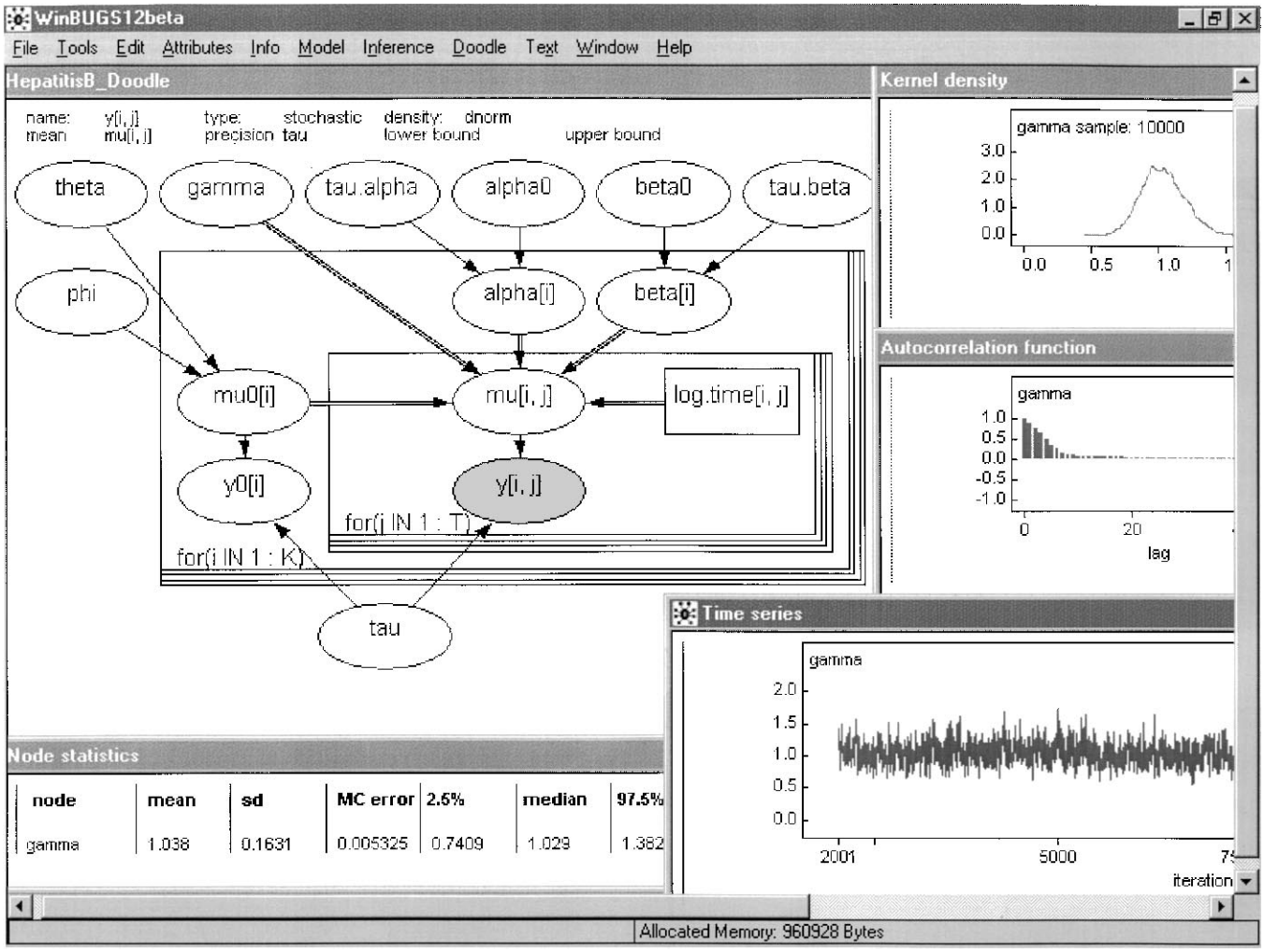

Fig. 2. *Screen dump from a WinBUGS session featuring a DoodleBUGS graph for the Hepatitis B immunization model discussed in Spiegelhalter et al. (1996a). The data,* `y[i, j]`*, are serial log-antibody-titre measurements obtained from (*$K = 106$*) Gambian infants after Hepatitis B immunization. The graph represents a hierarchical random effects model, with* `i` *indexing infants and* `j` *indexing repeated measures on each infant. The* `y[i, j]` *are assumed independent conditional on their mean* `mu[i, j]` *and on a parameter* `tau` *that represents the precision of the measurement process. Observed baseline log-titre values* `y0[i]` *are believed to be subject to the same sources of measurement error and are thus also dependent on* `tau`*. Each* `mu[i, j]` *is a deterministic function of* `log.time[i, j]`*, infant-specific intercept and gradient parameters (*`alpha[i]` *and* `beta[i]` *respectively), an 'errors-in-variables' covariate ('true' baseline log-titre:* `mu0[i]`*), and its associated coefficient* `gamma`*. Here a linear form is chosen (*`mu[i, j] <- alpha[i] + beta[i] * log.time[i, j] + gamma * mu0[i]`*), but linearity is by no means essential; a logical node's functional form is entered at the top of the graph when the node is selected. The* `alpha[i]`*,* `beta[i]`*, and* `mu0[i]` *are independently drawn from population distributions parameterised by:* `alpha0` *and* `tau.alpha`*;* `beta0` *and* `tau.beta`*; and* `theta` *and* `phi`*, respectively*

deterministic) relationships. The BUGS language is described in detail in Spiegelhalter *et al.* (1996b).

Graphical specification is achieved via the DoodleBUGS interface. Figure 2 shows the DoodleBUGS equivalent of the Hepatitis B immunization model discussed in Spiegelhalter *et al.* (1996a), which combines a random effects growth curve model for log-antibody-titre with measurement error on a covariate ('true' baseline log-titre). Nodes, directed links, and plates are drawn using simple mouse operations. Details of distributional assumptions or logical functions appear at the top of the graph when a node is selected (as shown for `y[i,j]`); these may be edited by the user.

Figure 2 also shows some results based on iterations 2001–12000 of an analysis using the depicted model: a 'time series' (or *trace*) plot, a kernel posterior density estimate, the autocorrelation function, and various summary statistics are shown for `gamma`, the coefficient associated with true baseline log-titre values. (Note that various other types of textual and graphical output are also available.) Such output can be generated at any time during the analysis – the relevant posterior samples are simply extracted from the appropriate *monitor object* (see Section 6.4) and manipulated accordingly.

From an abstract graphical representation of the model it is straightforward to 'read off' conditional independence

assumptions and thus determine the mechanism for constructing each full conditional distribution, irrespective of the underlying assumptions. For example, in Fig. 2 we can see that the full conditional for `gamma` is given by the prior for `gamma` multiplied by the density of each stochastic child, `y[i,j]` (`i` = 1, ..., `K`, `j` = 1, ..., `T`). (For the analysis shown `gamma` is assigned a normal prior and each `y[i,j]` is assumed to have arisen from a normal distribution with a mean `mu[i,j]` that is a *linear* function of `gamma` – hence the full conditional for `gamma` is normal.) WinBUGS uses object-orientation, which is discussed below, to exploit this fact.

5. Object-oriented programming

Detailed introductions to object-oriented programming can be found in Chapter 4 of Cornell and Horstmann (1997) and Chapter 12 of Reiser and Wirth (1992). Put simply the generic problem is to perform various operations on a collection of heterogeneous entities (or objects). Consider, for example, a simple graphics editor that draws a number of figures on the screen, e.g. rectangles, ellipses, lines. The first task is to identify a basic structure and functionality that is common to all objects and define an abstract *base type* that reflects this. Heterogeneity is handled by *extending* the base type.

An important feature of object-orientation is that types may have procedures *bound* to them – these are known as *methods*. For the graphics editor a base type named `Figure` could be defined. There would be no structure common to all objects but a common functionality could be imposed by binding various procedures to this base type. For example, each object of type `Figure` should be able to draw itself, via a method called `Draw`, say. For the base type this functionality is simply *declared* (abstractly): the details cannot yet be specified because, for example, rectangles and ellipses are drawn differently.

When a base type is extended, the extension inherits all of the base type's properties, i.e. its structure and methods. In addition, an extension may have *private data* added to it, such as 'fields' not contained in the inherited structure, details of inherited methods, and new methods. Extensions of the base type may also be extended, and so on. This results in a type hierarchy, which culminates in a set of *concrete* types, i.e. where all methods have been fully defined. For example, we could extend the base type `Figure` to `Rectangle`, `Ellipse`, and `Line`. These could be made concrete, by incorporating basic properties such as positional coordinates and dimensions into the structure as static fields, which would then be used to define the methods (e.g. `Draw`) for each extension. Alternatively, we may wish to extend `Line` to `Arrow`, etc.

An object is an *instance* (in the computer's memory) of a concrete type. It may be accessed via a variable of that type, or, alternatively, via any variable whose type the concrete type inherits from. For example, an object of type `Rectangle` may be assigned to a variable of type `Figure`; a call to this variable's `Draw` method would result in a rectangle being drawn even though the variable's type is abstract (private data are assigned *dynamically*). A principal benefit of this is that it enables the programmer to write very general procedures that operate abstractly on all objects without having to anticipate all the extensions.

6. Structure of WinBUGS

6.1. *Modules*

WinBUGS was developed using *BlackBox Component Builder* (Oberon microsystems, Inc., Zurich: `http://www.oberon.ch`), so-called because of its component-oriented nature (see Pfister 1997 for details). This is a *rapid application development* tool designed to extend the *BlackBox Component Framework*, which is an independently extensible class library. Programming is conducted using the Component Pascal language, which is a hybrid that combines procedural, object-oriented, and modular paradigms.

Software written using BlackBox is organised into *modules*. The module is the basic unit of compilation and typically contains one or more type definitions, including the details of type-bound procedures where appropriate. These types can be *exported* along with constants, variables, and procedures. Exported items constitute an *interface* via which different modules may interact. In BlackBox, interfaces play a very central role. Indeed, when a module is compiled a description of its interface is automatically created by the compiler. This can be compared to a contract (Meyer 1997, Chapter 11) between the module and its *clients*, i.e. other modules that make use of the exported items. A good contract should be clear, complete, and concise; it should not be ambiguous or lay down any irrelevant details, or one party may make false assumptions about the behaviour of the other. The interface summarises, accurately and concisely, the services that client modules can expect, without providing the details of how they will be carried out.

In short, a module is a 'black box' that interacts with its environment only through its interface. It shares its data structures with arbitrary other modules, about which it knows nothing. The most obvious benefit of modular programming is that by breaking down a large and complex problem into smaller problems that can be solved independently, the overall problem becomes substantially more manageable. However, the fact that interaction between modules only occurs through simple interfaces is also very important. So long as the interface remains the same a module can be replaced by a newer version without affecting the remainder of the software, which greatly facilitates evolution. This is the main reason for 'hiding' a module's implementation details: otherwise, changes may render assumptions based on an earlier version invalid.

6.2. *WinBUGS architecture*

A collection of modules that have related functionality is referred to as a *subsystem*. Figure 3 depicts the architecture of

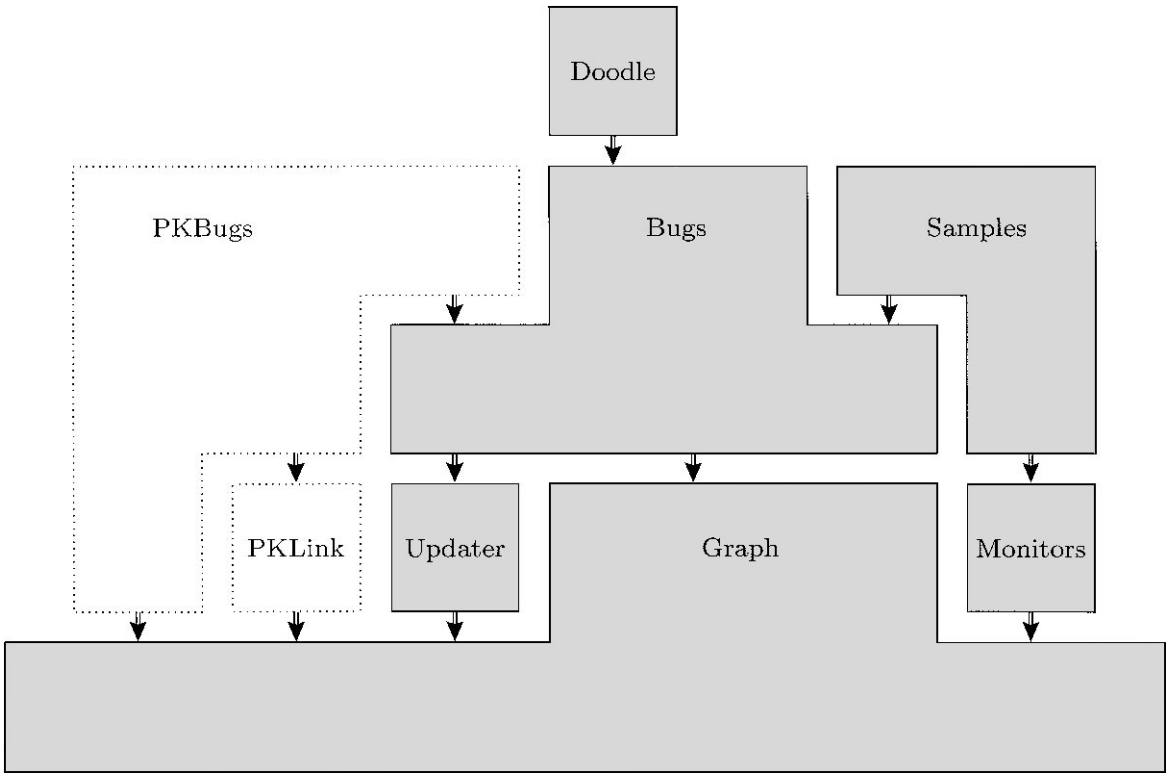

Fig. 3. *Architecture of WinBUGS. The shaded region shows the interaction between the six primary subsystems (Graph, Updater, Monitors, Bugs, Samples, and Doodle). The clear region shows how a specialized application, namely PKBugs, fits in. The depicted shapes and sizes of subsystems are meaningless – it is the hierarchical structure that is important. Arrows denote one-way communication between subsystems. If one subsystem points to another then the former makes use of the latter; moreover, the latter is unaware of the former's existence*

WinBUGS: it shows how the six primary subsystems interact. The arrows show *one-way* communication between subsystems. If one subsystem points to another then the former makes use of the latter. Moreover, the latter is unaware of the former's existence. Also shown are two subsystems of a specialized application, namely PKBugs (Lunn *et al.* 1998; `http://www.med.ic.ac.uk/df/dfhm/pkbugs_web/home.html`), designed for analysis of population pharmacokinetic data.

In this section we describe briefly the role of each WinBUGS subsystem.

(i) **Graph**. Modules that constitute the Graph subsystem collectively export a rich type hierarchy whose base type corresponds to a generic node in a graphical model. Objects of these types are used to construct an internal representation of the model. The Graph subsystem is entirely unaware of the other five subsystems – it merely provides 'building blocks' along with some rules as to how these building blocks should interact. The Graph subsystem is discussed in detail in the following sub-section (6.3).

(ii) **Updater**. The Updater subsystem provides objects that can 'update', via MCMC simulation, a node corresponding to an unknown parameter in the graphical model, i.e. they obtain a sample from the node's full conditional distribution.

(iii) **Monitors**. The Monitors subsystem defines the base types of objects that are responsible for storing the samples drawn from a node's full conditional distribution.

(iv) **Bugs**. Bugs has a number of responsibilities: it defines the 'grammar' for model specification, i.e. the BUGS language; parses the model specification; assembles (via Graph) and stores the internal representation of the model; provides a mapping between variables in the statistical model and the objects that represent them; and drives the MCMC algorithm.

(v) **Samples**. The Samples subsystem interacts with both Bugs, where details of the model are stored, and Monitors to produce textual and graphical output, such as summary statistics and posterior density plots.

(vi) **Doodle**. Doodle is a graphical interface that allows users to specify models via DAG diagrams. There is a one-to-one mapping between elements of the DAG and elements of the BUGS language, and so communication of the model from Doodle to Bugs is straightforward.

6.3. *The Graph subsystem*

6.3.1. Module and type hierarchy

By convention module names are prefixed by the name of the subsystem to which the module belongs. Thus all modules in the Graph subsystem begin with `Graph`, e.g. `GraphNodes`,

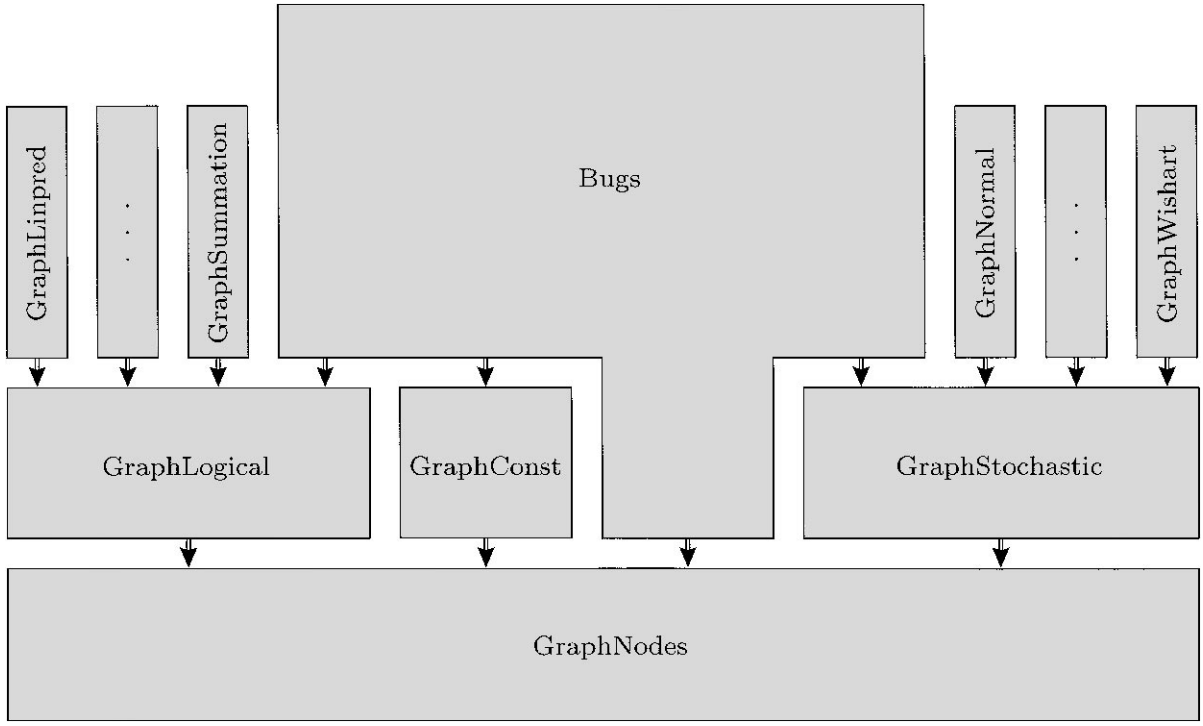

Fig. 4. *Module hierarchy of the Graph subsystem and how Bugs makes use of it. Notation as in Fig. 3*

`GraphStochastic`. Figure 4 shows the module hierarchy of the Graph subsystem and how the Bugs subsystem makes use of it.

The Graph subsystem defines a type hierarchy for representing nodes in a DAG. The base type `Node` is defined in module `GraphNodes` and is referred to as `GraphNodes.Node` (by convention). `GraphLogical.Node`, `GraphConst.Node`, and `GraphStochastic.Node` are extensions that represent logical, constant, and stochastic nodes respectively. Constant nodes are not extensible. Logical nodes are extended to particular classes of functions, such as linear predictors (module `GraphLinpred`) and sums of nodes (module `GraphSummation`). These are typically known to have convenient properties with respect to the derivation of (or, more generally, sampling from) full conditional distributions. They may, however, simply aid model specification. Stochastic nodes may be either univariate or multivariate (e.g. Wishart), and so `GraphStochastic.Node` is extended to `GraphStochastic.Univariate` and `GraphStochastic.Multivariate`. These are further extended, to concrete types, in numerous distribution-specific modules, such as `GraphNormal` and `GraphWishart`. Note from Fig. 4 that the Bugs subsystem is unaware of distribution-specific (and function-specific) modules and their respective types; it only requires knowledge of the abstract types.

6.3.2. Graph construction

Here we describe how the internal representation of the graphical model is constructed. We first introduce the `Name` data structure, which is defined in module `BugsNames`. For each variable in the statistical model, a variable of type `BugsNames.Name` is used to store its user-specified name, information regarding its dimensions, and the collection of node objects that represent it. This provides a mapping between the textual and internal representations of the model, so that objects corresponding to particular nodes in the graph can be located easily. (The user's model nomenclature is used to identify components of the internal representation because the arbitrary structure of the model cannot be anticipated.) Module `BugsIndex` stores a list of the `BugsNames.Name` variables used to represent the model.

To create node objects we make use of *factory objects*. Module `GraphNodes` exports an abstract type named `Factory`, which has a single method, `New`; a variable of this type is also exported. Each distribution-specific and function-specific module in the Graph subsystem exports a variable of a type that extends `GraphNodes.Factory`. This object's `New` method creates and returns (dynamically) an instance of the appropriate distribution-specific or function-specific node type. Thus a node object can be created simply via a call to the `New` method of the appropriate module's factory object, and so details of the node type, including the type itself, can be completely hidden so that the module is entirely 'black box' in nature.

Bugs uses these factory objects as follows. First, the user's model specification is parsed and a 'tree' representation of the model is created. The Bugs subsystem contains a resource file that defines the grammar for model specification. This is simply an editable text file containing identifiers for each available distribution and function, e.g. `dnorm(s,s)` denotes a normal

distribution, which should be specified using two scalar (`s`) parameters. Alongside each identifier is a procedure call with the syntax `Module.Install`, where `Module` is the name of the module in which the appropriate concrete node type is defined (e.g. `GraphNormal`) and `Install` is a simple procedure that sets the factory object in `GraphNodes` equal to the factory object in `Module`. Thus when the former's `New` method is called the appropriate node object is returned dynamically, and so Bugs requires no knowledge of distribution-specific or function-specific modules (which provides enormous scope for extending the software – see Section 7). Node objects are created simply by 'picking up' and executing commands in the resource file and by then using the factory object exported by `GraphNodes`.

Once all node objects have been created and incorporated into their respective `BugsNames.Name` variables, and once these have been stored within `BugsIndex`, it is necessary to define the directed links between nodes in the graph. This is achieved by each node object incorporating pointers to its direct parents. The names of direct parents are determined by referring back to the tree representation of the model created by the parser. These are used to locate parent objects, which are passed to the relevant node via its `Set` method (see Appendix).

6.3.3. Methods

A principal aim of any MCMC software should be to make use of sampling techniques that are both efficient and reliable. In WinBUGS, the methods attached to node objects allow the software to choose and employ the most appropriate sampling scheme for each full conditional distribution. They can be categorised as follows: Topology – methods that define or inform about the structure of the graph; Classification – methods that navigate the graph and provide information about how a stochastic node's full conditional distribution might be classified; and Updating – methods that evaluate functions of interest to *updater objects*, i.e. objects that obtain samples from the full conditional distributions. Some technical details of particularly important methods are provided in the Appendix.

6.4. *Updaters and monitors*

Updater objects, whose base type (`Updater`) is defined in module `UpdaterUpdaters`, are responsible for carrying out the MCMC simulation: they update the graph by calculating new values for nodes. There are updaters for specific distributions, i.e. when a node's full conditional can be expressed in closed form, and general updaters, such as Metropolis-Hastings samplers and 'slice' samplers (Neal 1997). The creation of specific updater objects is analogous to the creation of node objects. Once the appropriate node's full conditional distribution has been classified, an `Install` procedure is executed from a resource file. This sets the factory object in module `UpdaterUpdaters` equal to the appropriate updater-specific factory object; the former can thus be used to create specific updater objects without knowledge of the modules in which they are defined.

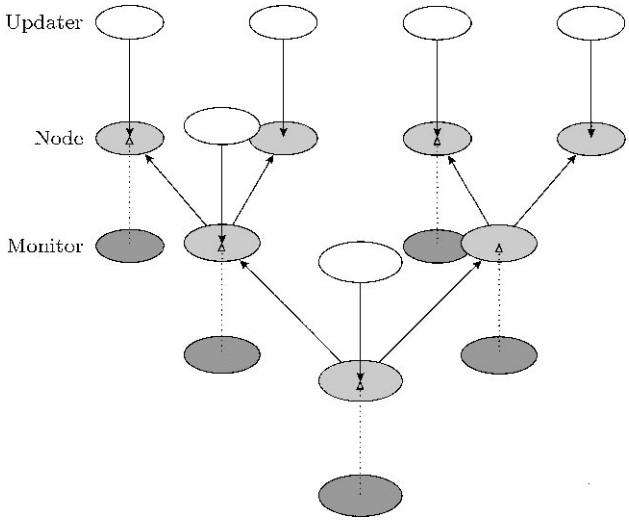

Fig. 5. *3-D representation of the graph (comprising node objects) and associated updater and monitor objects. There are three layers that extend into the page: updaters (white) form the top layer; nodes (light grey) form the second; and the third comprises monitors (dark grey). Arrows denote pointer variables that link the objects together. Node-to-offspring pointers are not shown. Monitor-to-node pointers are 'dotted' simply to give the impression of depth*

Each updater object incorporates a pointer to the node that it updates. Through this the updater can access relevant local information about the graph. To enable this coupling the updater has a method called `Init`, which receives the appropriate node object as an argument – this is analogous to a node's `Set` method.

The most important procedure bound to updater objects is the `MCMC` method. This draws one random sample from the node's full conditional distribution and updates the node's `value` field (see Appendix) accordingly. For general updaters the `MCMC` method is straightforward to define, because no special considerations are required. However, when the full conditional distribution can be expressed in closed form the parameters of that closed form must somehow be derived. This is done by making use of the `LikelihoodForm` method of each of the attached node's offspring (see Appendix).

Monitor objects (whose base types are defined in the Monitors subsystem) are also coupled to nodes via pointer variables. At the end of each iteration of the Gibbs sampler, each monitor object accesses the node attached to it and stores its current value in an array field. Whereas updaters are automatically created by the Bugs subsystem for every (unobserved) stochastic node in the graph, monitors are assigned to either stochastic or logical nodes by the user during run-time. Figure 5 depicts the internal representation of the graph, comprising node objects, along with associated updater and monitor objects.

7. Extensibility

The BlackBox Component Builder extends the BlackBox Component Framework to provide a user-interface that enables rapid

application development: it incorporates a powerful text editor, for example, and to some extent automates the programming of many of the essential features of modern software, e.g. *dialogue boxes*. The various subsystems that constitute the WinBUGS software are simply further extensions of this framework. The Graph subsystem provides building blocks that are generally useful for representing Bayesian models; Updater and Monitors provide building blocks that are generally useful for analysing these models via MCMC techniques, and so on. In this sense WinBUGS is a component framework in its own right.

The software has been designed so that users can extend it, with minimal effort, to meet their own requirements. All that is required is some familiarity with the Component Pascal language and the structure of the WinBUGS framework (this paper addresses the latter issue). There are three main ways of extending the framework: (i) new types of logical node; (ii) new types of stochastic node; and (iii) new MCMC updating algorithms.

In the first two cases, the user must write a single module that defines an extension of either `GraphLogical.Node` or `GraphStochastic.Node`. This module must also define an extension of `GraphNodes.Factory` and a simple `Install` procedure that sets the global factory object in module `GraphNodes` equal to an instance of the new factory object type. A call to the `Install` procedure should be included in the resource file that defines the grammar for model specification, along with an identifier for the new node type. This procedure call can be 'picked up' and executed by WinBUGS even though the software is unaware of the new module's existence; thus WinBUGS can make use of new node objects without having to anticipate them. This is made possible by the fact that BlackBox can load modules *on demand* during run-time – this is known as run-time linking.

New updater objects are integrated into the framework in an analogous fashion, by extending both `UpdaterUpdaters.Updater` and `UpdaterUpdaters.Factory`, exporting a simple `Install` procedure, and making modifications to the appropriate resource file.

Another way of extending WinBUGS would be to write an application (or user-interface) for users with specialized requirements, e.g. when models are best specified using approaches other than the BUGS language. Such an application could merely relieve the Bugs subsystem of its parsing and graph assembly responsibilities. The new parsing process would typically lead to a convenient (preliminary) internal representation of the model that could be transposed into WinBUGS objects.

The `BugsNames.Name` data structure provides a valuable medium for communicating models to the Bugs subsystem. Module `BugsIndex` can be instructed, by any application, to store a graphical model comprising `BugsNames.Name` variables. Thus, a specialized application may create node objects by making use of the Graph subsystem, organise them into `BugsNames.Name` variables, and then make Bugs aware of the graph by passing these to it. Once the directed links in the graph have been defined (via each node's `Set` method) the Bugs subsystem has all the information necessary to navigate the graph, build the required full conditional distributions, and allocate appropriate updater objects. Hence, WinBUGS' Bayesian modelling capabilities could then be fully exploited to conduct the remainder of the analysis.

To date, two such applications have been developed, namely DoodleBUGS (defined by the Doodle subsystem) and PKBugs (Lunn *et al.* 1998). The latter was designed for the analysis of population pharmacokinetic data. Population pharmacokinetic models are generally nonlinear and typically complex. Indeed, the complexity of dosing histories in observational studies is often such that model specification via the BUGS language would be infeasible. Instead, PKBugs allows the user to specify the model using an established shorthand notation for data entry (Beal and Sheiner 1992) along with a series of simple dialogue boxes and menu commands.

8. Discussion

This paper illustrates how modern computing concepts can be used to solve, elegantly and efficiently, extremely difficult problems. Object-orientation is the key to exploiting the general properties of DAGs so that the software can deal efficiently with arbitrarily complex models. The component-oriented design of the software improves its reliability and renders it easily extensible. The fact that modules can be loaded on demand (i.e. at run-time) means that new features can be added without having to recompile any part of the existing software.

Although WinBUGS has been designed primarily for DAGs, other types of graphical model, such as conditional autoregressive structures with undirected links (Besag *et al.* 1995; Spiegelhalter, Thomas and Best 1996), are also permissible. These have somewhat less convenient factorization properties but are still quite tractable.

Currently WinBUGS can only perform multivariate (or 'block') updating when it can derive the appropriate full conditional distribution in closed form, e.g. Wishart. Multivariate updating can greatly improve the overall efficiency of the MCMC simulation, and a future version of the software is intended to employ general multivariate sampling techniques, e.g. Metropolis-Hastings, for any suitable block of nodes. Apart from this modification, work on the *core* of the WinBUGS framework has now been completed. Future work will largely entail extension of this core and the (further) development of specialized applications, such as PKBugs and GeoBUGS; GeoBUGS will facilitate the specification of spatial models for disease mapping and related applications, and will provide a graphical interface for map display and interpretation of results. It is anticipated that Java versions of all WinBUGS-related software will be available in the near future, so that they may be run on Unix systems.

An educational version of WinBUGS is available at `http://www.mrc-bsu.cam.ac.uk/bugs/`. This can be used to analyse numerous example problems and general problems of limited size for evaluation purposes. All documentation and examples are packaged as part of the software. Currently, a full version can be obtained free of charge via registration. Table 2

Table 2. *Distribution of registered WinBUGS users (as at July 1999) by geographical location, type of use, and affiliation*

Location	Users	Type of use	Users	Affiliation	Users
North America	1064	Educational	1303	University	1436
Britain	336	Medical	616	Not-for-profit	279
Rest of Europe	481	Environmental	143	Company	277
Australia & New Zealand	119	Industrial	103	Personal	138
Rest of World	224	Financial	59	Other	94

shows the distribution (as at July 1999) of over 2000 registered users worldwide, by geographical location, type of use, and affiliation. The software is primarily used in English-speaking countries for educational purposes in university institutions, although over 40% of all usage is for 'real-life' applications.

Appendix: Methods bound to node objects

Of the 10 methods bound to `GraphNodes.Node`, three are particularly important. These are shown along with their categorisations in the top part of Table 3 and are described below.

(i) `Set`. Each distribution-specific and function-specific node object incorporates pointers to its direct parents. These define the local structure of the graph and are assigned using the `Set` method. For each node the Bugs subsystem locates the objects that correspond to its direct parents. These are then passed to the node as arguments to its `Set` method and the node incorporates them into its structure.

(ii) `Parents`. This method returns a list of the node's *stochastic* parents. Once the `Set` method has been called, a node has access to its direct parents. For each of these a *type test* is made. If the parent is stochastic (`GraphStochastic.Node`) then it is added to the list. If the parent is constant (`GraphConst.Node`) then it is ignored. If the parent is logical (`GraphLogical.Node`) then recursion is used: the parent's `Parents` method is called, and so on. Recursion in each branch ceases when a stochastic or constant node is found – this is added to the list if stochastic.

(iii) `Value`. The `Value` method returns the node's current value. For logical nodes this is evaluated using the parents' current values. For stochastic nodes the real number stored in the `value` field is returned (see below).

Table 3. *Important methods (and their categorisations) bound to the types `GraphNodes.Node` and `GraphStochastic.Node`*

Type {static fields}	Methods	Use
`GraphNodes.Node`	`Set`	Topology
{no static fields}	`Parents`	Topology
	`Value`	Updating
`GraphStochastic.Node`[a]	`BuildLikelihood`[b]	Topology
{ `value`: REAL;	`ClassLikelihood`	Classification
`offspring`: List of	`ClassPrior`	Classification
`GraphStochastic.Node`;	`LikelihoodForm`	Updating
`properties`: SET }	`Likelihood`	Updating
	`Prior`	Updating
	`Conditional`[b]	Updating

[a] `GraphStochastic.Node` inherits all of `GraphNodes.Node`'s methods and has numerous additional methods.
[b] Details of method specified at abstract level, in module `GraphStochastic`.

Extensions of `GraphNodes.Node` inherit all of its methods. `GraphLogical.Node` has two additional methods. One of these is `Mapping`, which plays an important role in the classification of full conditional distributions and is discussed below. `GraphStochastic.Node` has an additional 20 methods. Those of particular importance are shown in the lower part of Table 3 and are also discussed below. This type's static fields are described as follows: `value` stores the node's current value; `offspring` is a list of `GraphStochastic.Node` objects that correspond to the node's stochastic children; and `properties` is a set of properties of the node – any of a number of integer constants, such as `data` and `censored`, exported by module `GraphStochastic` can be included in the `properties` field.

(iv) `BuildLikelihood`. The `BuildLikelihood` method obtains the node's stochastic parents via the `Parents` method and adds the node to the `offspring` field of each. For example, a call to the `BuildLikelihood` method of y_i in Fig. 1 would result in y_i being added to the `offspring` fields of α, β, and τ. Module `BugsCompiler` exports a procedure that calls `BuildLikelihood` for each stochastic node in the graph, in order to fill the `offspring` fields of all stochastic nodes. (Only children that contribute to a node's likelihood component are added to its `offspring` field, i.e. nodes used for prediction are excluded.)

(v) `ClassLikelihood`. This method is instrumental in the classification of full conditional distributions. Suppose we are interested in the full conditional distribution of `node`. For each element of `node.offspring` (denoted by `child`) we call `child.ClassLikelihood(node)`, i.e. `node` is an argument to the `ClassLikelihood` method. This classifies the density of `child` *as a function of* `node`, e.g. the normal density for each y_i in Fig. 1 has the form of a gamma density when considered as a function of τ. Often, `child` is not immediately aware of how it is related to its stochastic parents (e.g. `node`) because

some, or all, of its direct parents are logical. In this case each logical parent's `Mapping` method is used to classify the parent as a function of `node`. This method works recursively up the graph (i.e. against the arrows), building a functional classification along the way, until either `node` or one of the other stochastic parents is located.

Some functions have convenient properties in this respect. For example, the density of each y_i in Fig. 1, when considered as a function of either α or β, is normal, because μ_i is *linear* in both α and β. (If we were to assign normal priors to α and β then their full conditionals would therefore also be normal.) Module `GraphRules` exports (as integer constants) the possible classifications for functions and densities, along with numerous rules for combining them. Density classifications include closed forms, such as `normal` and `gamma`, and more general forms, such as `logCon` (log-concave). This method is central to the 'expert system' described in Section 3.2, which enables WinBUGS to perform efficient sampling for arbitrary models.

(vi) `ClassPrior`. The `ClassPrior` method simply returns a classification for the node's assumed distribution in the full probability model. This is used in conjunction with `ClassLikelihood` (for each child) to classify the node's full conditional distribution.

(vii) `LikelihoodForm`. In cases where a stochastic node's full conditional distribution is available in closed form this method is used to derive the parameters of that closed form. It is called for each element of the node's `offspring` field and returns the child's contributions to the full conditional parameters. The form of these contributions depends on the type of full conditional distribution, e.g. normal, gamma; this information is passed to the `LikelihoodForm` method via its argument list.

If we were to assign a gamma prior, $Ga(a, b)$ say, to τ in Fig. 1 then its full conditional would be $Ga(a + \sum_{i=1}^{N} \frac{1}{2}, b + \sum_{i=1}^{N} \frac{1}{2}(y_i - \mu_i)^2)$. For the derivation of this full conditional, the `LikelihoodForm` method of each y_i (the offspring of τ) would return $\frac{1}{2}$ and the current value of $\frac{1}{2}(y_i - \mu_i)^2$ as the contributions of that child to the shape and inverse-scale parameters respectively. For the derivation of a normal full conditional, e.g. for α or β (assuming normal priors), the `LikelihoodForm` method of each y_i (also the offspring of α and β) would return (current values of) other appropriate functions. These contributions are combined, in a way that befits the circumstances, by updater objects (see Section 6.4).

(viii) `Likelihood`, `Prior` and `Conditional`. `Conditional` evaluates the natural logarithm of the node's full conditional density, up to proportionality. This is required to perform MCMC simulations in cases where the full conditional cannot be expressed in closed form. First the node's `Prior` method is called. This returns the node's log-density evaluated at the node's current value (with the parents also at their current values); for efficiency, only terms involving the node itself are used in the calculation. The `Likelihood` method is identical except that all terms are used in the calculation. This is called for each element of the node's `offspring` field. The log-full conditional (up to proportionality) is given by the sum of these density evaluations.

Methods `BuildLikelihood` and `Conditional` are identical for all stochastic nodes and are thus defined at an abstract level, in module `GraphStochastic`. The other methods in the lower part of Table 3 are distribution-specific.

Acknowledgments

This work was funded by: the Engineering and Physical Sciences Research Council (Award Number GR/L 10437); the Economic and Social Research Council (H519255036, H519255023); and the Medical Research Council (G9803841). We are grateful to Richard Arnold for his invaluable contribution to the WinBUGS project and for helpful discussions on this paper. We would also like to thank the users of WinBUGS for their patience and enthusiasm.

References

Ayanian J.Z., Landrum M.B., Normand S.-L.T., Guadagnoli E., and McNeil B.J. 1998. Rating the appropriateness of coronary angiography–Do practicing physicians agree with an expert panel and with each other?. New England Journal of Medicine 338: 1896–1904.

Beal S.L. and Sheiner L.B. 1992. NONMEM User's Guide, parts I-VII. NONMEM Project Group, San Francisco.

Besag J., Green P., Higdon D., and Mengersen K. 1995. Bayesian computation and stochastic systems. Statistical Science 10: 3–66.

Cornell G. and Horstmann C.S. 1997. Core Java, 2nd Edition. Prentice Hall, New Jersey.

Gelfand A.E. and Smith A.F.M. 1990. Sampling-based approaches to calculating marginal densities. Journal of the American Statistical Association 85: 398–409.

Gelman A., Carlin J.B., Stern H.S., and Rubin D.B. 1995. Bayesian Data Analysis. Chapman and Hall, London.

Geman S. and Geman D. 1984. Stochastic relaxation, Gibbs distributions and the Bayesian restoration of images. IEEE Transactions on Pattern Analysis and Machine Intelligence 6: 721–741.

Gilks W. 1992. Derivative-free adaptive rejection sampling for Gibbs sampling. In: Bernardo J.M., Berger J.O., Dawid A.P., and Smith A.F.M. (Eds.), Bayesian Statistics 4. Oxford University Press, Oxford, pp. 641–665.

Gilks W.R., Richardson S., and Spiegelhalter D.J. 1996. Markov Chain Monte Carlo in Practice. Chapman and Hall, London.

Gilks W.R. and Wild P. 1992. Adaptive rejection sampling for Gibbs sampling. Applied Statistics 41: 337–348.

Hastings W.K. 1970. Monte Carlo sampling-based methods using Markov chains and their applications. Biometrika 57: 97–109.

Lauritzen S.L., Dawid A.P., Larsen B.N., and Leimer H.G. 1990. Independence properties of directed Markov fields. Networks 20: 491–505.

Lunn D.J. and Aarons L. 1998. The pharmacokinetics of saquinavir: A Markov chain Monte Carlo population analysis. Journal of Pharmacokinetics and Biopharmaceutics 26: 47–74.

Lunn D.J., Wakefield J., Thomas A., Best N., and Spiegelhalter D. 1998. PKBugs User Guide. Dept. Epidemiology and Public Health, Imperial College School of Medicine, London.

Metropolis N., Rosenbluth A.W., Rosenbluth M.N., Teller A.H., and Teller E. 1953. Equations of state calculations by fast computing machines. Journal of Chemical Physics 21: 1087–1091.

Meyer B. 1997. Object Oriented Software Construction, 2nd Edition. Prentice Hall, New Jersey.

Neal R.M. 1997. Markov chain Monte Carlo methods based on 'slicing' the density function. Technical Report 9722, Dept. of Statistics, University of Toronto.

Pfister C. 1997. Component Software: A Case Study Using BlackBox Components. Oberon microsystems, Inc., Zurich.

Pitt M.K. and Shephard N. 1999. Time-varying covariances: A factor stochastic volatility approach. In Bernardo J.M., Berger J.O., Dawid A.P., and Smith A.F.M. (Eds.), Bayesian Statistics 6. Oxford University Press, Oxford, pp. 547–570.

Reiser M. and Wirth N. 1992. Programming in Oberon: Steps Beyond Pascal and Modula. ACM Press, New York.

Ripley B.D. 1987. Stochastic Simulation. Wiley, New York.

Spiegelhalter D.J. 1998. Bayesian graphical modelling: A case-study in monitoring health outcomes. Applied Statistics 47: 115–133.

Spiegelhalter D.J., Best N.G., Gilks W.R., and Inskip H. 1996a. Hepatitis B: A case study in MCMC methods. In Gilks W.R., Richardson S., and Spiegelhalter D.J. (Eds.), Markov Chain Monte Carlo in Practice. Chapman and Hall, London, pp. 21–43.

Spiegelhalter D.J., Dawid A.P., Lauritzen S.L., and Cowell R.G. 1993. Bayesian analysis in expert systems (with discussion). Statistical Science 8: 219–283.

Spiegelhalter D.J., Thomas A., and Best N.G. 1996. Computation on Bayesian graphical models. In Bernardo J.M., Berger J.O., Dawid A.P., and Smith A.F.M. (Eds.), Bayesian Statistics 5. Oxford University Press, Oxford, pp. 407–425.

Spiegelhalter D., Thomas A., Best N., and Gilks W. 1996b. BUGS 0.5: Bayesian inference Using Gibbs Sampling–Manual (version ii). Medical Research Council Biostatistics Unit, Cambridge.

Szyperski C. 1995. Component-oriented programming: A refined variation of object-oriented programming. The Oberon Tribune 1: 1–5.

Wakefield J. and Morris S. 1999. Spatial dependence and errors-in-variables in environmental epidemiology. In Bernardo J.M., Berger J.O., Dawid A.P., and Smith A.F.M. (Eds.), Bayesian Statistics 6. Oxford University Press, Oxford, pp. 657–684.

Whittaker J. 1990. Graphical Models in Applied Multivariate Analysis. Wiley, Chichester.