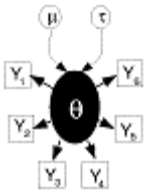



# OpenBUGS Developer Manual

Version 5.0.0, August 2020

Andrew Thomas
MRC Biostatistics Unit
Cambridge

March 2022

e-mail: helsinkiant@gmail.com

## Contents

# Introduction

The BUGS software is a popular statistical package for fitting Bayesian probability models. The software can be thought of as a compiler for the BUGS language. A model is described using the BUGS language then the compiler creates data structures that are able to make statistical inference for the model. There are two aspects to understanding the BUGS software: firstly the syntax and symantics of the BUGS language and secondly the source code that implements the BUGS compiler and inference engine. The design objectives of the BUGS language are discussed in the first chapter, subsequent chapters discuss the tools used to develop the software and the structure of the software.

The BUGS software is writen in the Component Pascal language. The software has been developed using the integrated development enviroment BlackBox Component Builder. Component Pascal is actually a development of the Oberon language. I am not the only person who writes software in Component Pascal, however the nearest main stream language to Component Pascal is Go. The main difference between the two languages is that Component Pascal uses Pascal style syntax while Go uses C style syntax. Both Component Pascal and Go are modular programming languages with well defined interfaces. Component Pascal uses a simple mark up of the source code model to define the module's interface -- the export marks * and -.

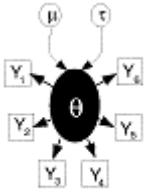
The source code of the BUGS software consists of several hundred modules organized into about one dozen software subsystems. Each module has a compound name where the first part of the name gives the subsystem to which the module belongs and the second part the file name of where the module is stored. For example the module BugsNames is in subsystem Bugs and is stored in file ./Bugs/Mod/Names.odc. When a source code module is compiled two files are created: the executable code file with the file extension ocf and the interface definition file with file extension osf. Both these files have the same file name as the source code file but are stored in directories ./SubsystemName/Code and ./SubsystemName/Sym respectively. Both files are binary files but the development enviroment contains tools to read them. Each module should have a fourth file assocciated with it: the documentation file. This file should contain a brief description of all the quantites listed in the module's interface definition file. Thisdocumentation file is a text file and stored in ./SubsystemName/Docu.

Each software subsystem has two summary files: a documentation file and an subsystem map file. The documentation file contains a description of the functionality provided by the subsystem. The subsystem map file contains hypertext links to the documentation files for each module in the

subsystem.These files can be accessed from the hypertext file ./Developer/SoftDocu. All the modules in the BUGS software are listed in the make file ./Developer/Make. They are listed in the order in which they should be compiled. Information about a module can be easily found by highlighting its name (in any document) and then right clicking into the highlight to open a pop up menu. If a module name is followed by a period and then another name this is called a qualified name. A qualified name can be highlighted and then right clicked to access information about that particular quantity in a module. For example the module BugsNames contains a procedure New so highlighting BugsNames.New and right clicking will give information on the New procedure in the module BugsNames.

The BUGS software contains several tools for inspecting the graphical model produced by the BUGS compiler. The most probing of these tools, the 'Node inspector' tool, is accessed from the Info menu. This tools allows the user to inspect the data structure in the graphical model that corresponds to a name in the BUGS language description of the statistical model. The tool also allows the user to inspect the sampling algorithm, if any, which acts on a particular name in the BUGS language model.

In the BUGS software's Info menu there is a verbose option which yield more information about the compilation of a particular BUGS language model. If the verbose option is checked then the source code generated for each type of logical relation in the model is displayed. Also displayed is a detailed list of the substeps in the compilation process plus the time taken in each substep. This list contains pairs of qualified names and times. Highlighting the qualified name and right clicking gives a quick way of viewing the source code for each step of the compilation process. The Info menu contains a decode tool which displays the data structure the BUGS compiler produces in a human readable form. Finally there is an option in the Info menu to display a representation of a compiled model in a form that is closer to the notation that the STAN or Turing packages use




# The Design of the BUGS PPL

Andrew Thomas
MRC Biostatistics Unit
Cambridge

August 2021

## Introduction

The BUGS software is a popular statistical package for fitting Bayesian probability models. It has evolved from being able to fit small simple models to being able to fit large complex models. In its latest version called MultiBUGS the software can use multi core computing to speed up the fitting of larger models. The software has always been distinctive in having a programming language to describe Bayesian statistical models. Such languages are now known as Probabilistic Programming Languages or PPLs.

## History

The BUGS software project has been running since 1989 when David Spiegelhalter of the MRC Biostatistics Unit recruited me to develop software to perform Gibbs Sampling for a wide range of statistical models. The initial idea for the software came from work David had recently done with Steffen Lauritzen on using probability to handle uncertainty in expert systems. Steffen had led a group which developed the Hugin package to implement evidence propagation in probabilistic expert systems based on message-passing in directed graphical models, and David wanted to have some software which extended these ideas by using Gibbs Sampling instead of exact Message Propagation as the inference algorithm. This new software became the BUGS project: BUGS standing for **B**ayesial inference **U**sing **G**ibbs **S**ampling The first release of the software was shown off at the Fourth Bayesian Statistics Conference in Valencia Spain in April 1991. The first version of the software could run two models the famous Rats example using normal gamma conjugate sampling and Asia a simple probabilistic expert system.

From the earliest version of BUGS the software had a programming language to describe statistical models. This language used to describe models in the BUGS software became known as the BUGS language. The BUGS language has remained very stable over a period of thirty

years with only minor changes which have usually been simplifications. The capabilities of the BUGS software have grown greatly over this period. The BUGS software can now use a wide range of sampling algorithms, as well as Gibbs Sampling, such as Random Walk Metropolis and Hamiltonian Monte Carlo. Happily the Metropolis algorithm was less popular when the BUGS project started or we might have had a BUMS project which would not have been so respectable.

## Other PPLs

In recent years more software systems have been built to perform inference for Bayesian statistical models. These systems often come with a language to describe the statistical model that is to be fitted. The language these systems use are called Probabilistic Programming Languages or PPLs. The basic idea of a PPL is to write a description of a Bayesian probability model, combine it with a data source and automatically produce software to make inference.

Python and TensorFlow are much used by the machine learning community but less so by the statistical community. Many PPLs have been built on top of these two languages / frameworks. However neither of them were around or popular when the BUGS project was started and so did not influence the design of the BUGS language. It is possible but uncertain that the BUGS language has had a minor influence on Python / TensorFlow based PPLs.

The BUGS language has had more influence on the design of some other PPL with the influence being strongest in JAGS, STAN and SlicSTAN family. Both JAGS and STAN PPLs have a block structure that was also present to some extent in the earliest version of the BUGS language. These block structures make the implementation of the PPL simpler but also make the use of the PPL more pernickety. The BUGS language evolved to loose its block structure and to soley describe the Bayesian statistical model. JAGS and STAN languages moved in the opposite direction and gained more block structure.

The major evolution of the BUGS language can be seen in these two versions of the program for the Rats model: the first version from around 1990 and the second version from around 2000.

```
model Rats;
   data in "c:\bugs\dat\rats.dat";
   inits in "c:\bugs\in\rats.in";
const
   N= 30;    # number of rats
   T = 5;    # number of time points
var
   x[T], mu[T, N], Y[N, T], alpha[N], beta[N], alpha_c,
   tau_alpha, beta_c, tau_beta, tau_c, alpha_0, x_bar;
{
   alpha_c ~ Normal(0.0,1.0E-6)
   beta_c ~ Normal(0.0,1.0E-6)
   tau_c ~ Gamma(0.001,0.001)
   tau_alpha ~ Gamma(0.001,0.001)
```

```
    tau_beta ~ Gamma(0.001,0.001)
    for( i in 1 : N ) {
        alpha[i] ~ Normal(alpha_c,alpha.tau)
        beta[i] ~ Normal(beta_c,beta.tau)
        for( j in 1 : T ) {
            Y[i , j] ~ Normal(mu[i , j],tau_c)
            mu[i , j] <- alpha[i] + beta[i] * (x[j] - x_bar)
        }
    }
    sigma <- 1 / sqrt(tau_c)
    alpha_0 <- alpha_c - x_bar * beta_c
}


model
{
    for( i in 1 : N ) {
        for( j in 1 : T ) {
            Y[i , j] ~ dnorm(mu[i , j],tau.c)
            mu[i , j] <- alpha[i] + beta[i] * (x[j] - xbar)
        }
        alpha[i] ~ dnorm(alpha.c,alpha.tau)
        beta[i] ~ dnorm(beta.c,beta.tau)
    }
    tau.c ~ dgamma(0.001,0.001)
    sigma <- 1 / sqrt(tau.c)
    alpha.c ~ dnorm(0.0,1.0E-6)
    alpha.tau ~ dgamma(0.001,0.001)
    beta.c ~ dnorm(0.0,1.0E-6)
    beta.tau ~ dgamma(0.001,0.001)
    alpha0 <- alpha.c - xbar * beta.c
}
```

Note in particular the later versions of the model syntax do not mention the source of the data. This is specified seperately outside the BUGS language -- the probability model is independant of any data. Also note that the second version does not have any constant or variable declarations.

Most PPLs have two types of relation: the "takes the value" relation (denoted by <- in the BUGS language) and the "is distributed as" relation (denoted by ~ in the BUGS language). It is the "is distributed as" relation that distinguishes PPLs from other types of programming languages and heavily influences their semantics and how they are implemented. In the following sections we will describe what information the block structure makes manifest and how it is also implied in the simplified modern version of the BUGS PPL.

## Lexical Issues

The BUGS language consists of a stream of characters: the latin and greek letters (except lower case terminal sigma), digits, punctuation marks and special characters. The lexical analyzer groups this stream of characters into a stream of tokens. The parser takes this stream of tokens and builds a syntax tree to represent the model. The two most important type of token are names and numerical literals. The BUGS language has the convention that if a name is followed imediately by a round bracket, that is by a "(", then the names is a reserved name in the BUGS language and does not represent a variable in the model. For example in the Rats model above "dnorm" and "for" are reserved names while "Y" and "mu" are variables in the model. By scanning the stream of tokens that constitute a BUGS language model the names of all the variables in the model can be found. This removes one of the functions of the var block in the old version of the BUGS language.

## Names in the BUGS Language

Once the names of the variables in the BUGS language model have been found they can be entered into a table of names. The var block provides two other sorts of information about names in the model: the shape of the name and the dimensions of the name. Names in the model can either be scalars or tensors with tensors of rank one usually being called vectors and tensors of rank two called matrices. Tensors have a square open bracket "[" after their name and then one or more indices sperated by commas and a closing square bracket "]". The parser uses this information to add the rank of each name to the name table with scalar having rank zero. The BUGS compiler checks that the shape of each tensor is the same in each place it is used in a BUGS language model.

By convention indices start at one in the BUGS language. The BUGS language compiler expands all the for loops in the model and records the value of the indices of each use of a tensor on the left hand side of each relation. The range of each indice, for a tensor, is set at the maximum value observed value of the index and added to the name table. There is one exception to this procedure for finding index bounds: names that are data, that is in the data source, have the ranges of their indices fixed in the data source. Each scalar and each component of a tensor used on the right hand side of a relation must occur either on the left hand side of a relation and or in a data source. To be able to expand all the for loops in the model the compiler needs the value of the loop bounds. These are 1, N and T in our Rats model. This is one use of the constant block in the early version of the BUGS language and is discussed further below. For each use of a component of a tensor on the right hand side of a relation the values of the indices are checked to ensure that they are within the inferred array bounds. For example in the second Rats model programme the implied upper bound for the index i of the vector quantity alpha[i] is N from expanding the for loop that contains the statement alpha[i] ~ dnorm(alpha.c, alpha.tau). The alpha vector occurs on the right hand side of the statement mu[i, j] <- alpha[i] + beta[j] * (x[j] - xbar) and the compiler checks that i does not take a value that is greater than N.

# Statement Types

The BUGS language has three types of statement: stochastic assignments, logical assignments and repetition loops. There is a fourth type of statement, a static if statement, but this can be thought of as a special case of the repetition statement where the statements within the loop occurs either zero times or once in the model.

# Stochastic Assignment

The stochastic assignment relation consists of a scalar, a tensor component or a tensor on the left hand side of the "~" operator and a righthand side. The righthand side is the name of a distribution followed by a comma seperated list of arguments enclosed in a pair of round brackets. Each argument of the distribution can be a scalar, a tensor component or a tensor, if it is a scalar it can be a numerical literal. Most stochastic relations in the BUGS language invove scalars on the left hand side such as the normal distribution or the binomial distribution. For example

```
y ~ dnorm(mu, tau)

alpha.c ~ dnorm(0.0, tau)

r ~ dbin(p, n)
```

There are some tensor valued stochastic relations involving, for example, the multivariate normal distribution or the wishart distribution. Tensor valued stochastic assignments have the size of the tensor explicitly specified on their left hand sides. For example

```
beta[i, 1:2] ~ dnorm(mu.beta[], R[ , ])

R[1 : 2 , 1 : 2] ~ dwish(Omega[ , ], 2)
```

The size of tensors on the right hand side can be inferred from the distribution name on the right hand side and the number of components on the left hand side in most cases. For the first example above mu.beta has two components and R four (2 by 2). While for the second example Omega has four components (2 by 2). The size of tensors occurring on right hand sides can be made explicit in which case the sizes given are checked against what is expected.

# Logical Assignment

The logical assignment relation consists of a scalar, a tensor component or a tensor on the left hand side of the "<-" operator and an expression on the righthand side. Most logical relations

are scalar valued and can have arbitary expressions involving operands that are scalars, tensor components or numerical literals and the basic arithmetic operators plus a number of built in functions such as the exponential function. These built in functions can also have arguments that are expressions. There are a few external functions that take a vector argument and return a scalar value such as the mean function. These external functions have arguments that are scalars, tensors or components of tensors but not expressions. A few logical assignments can be tensor valued such as matrix inverse. These tensor valued relations can only have a single external function on their right hand side.

## Repetition

The BUGS language has one construct for repetition: the for loop. The statement has the syntax

```
for(i in lower : upper) {   XXXX    }
```

where XXXX is a sequence of statements in the BUGS language which can include other for statements. The left hand sides of the XXXX statements usually depend on the loop control variable i, the righthand sides of the XXXX can but does not have to depend on the loop control variable i. The loop control variable i takes values that are greater than or equal to the value of the quantity lower and less than or equal to the value of the quantity upper. The bounds lower and upper in a for loop must take fixed integer values. They could for example be integer numerical literals such as 1. Another way of specifying these loops bounds is in the data source. A final way of specifying loop bounds is by a simple logical assignment, involving scalars, in the BUGS language. We could for example write T <-5 , N<- 30 in a model. This is similar to how the constant block worked in early versions of the BUGS language. The loop control varaibles have implicit scope of the for loop they are defined in and can not be used as names in the BUGS language model.

This is allowed `for(i in 1 : 10) { x[i] <- i }`

while this is not `for(i in 1 : 10) { i ~ dnorm(0,0, 1.0) }`

but we could write `for(i in 1 : 10) { x[i] ~ dnorm(0.0, 1.0) x[i] <- i }`

An interesting special case of the for loop is

```
for(i in 1 : 1) { XXXX }
```

where the statement XXXX occurs once in the model. Another interesting special case is

```
for(i in 1 : 0) { YYYY }
```

where the stayements YYYY will not occur in the model. We can combine these two ideas to give a sort of if statement

```
for(i in 1 : boolean) { XXXX }
```

or

```
for(i in 1 : 1 - boolean) { YYYY }
```

where boolean is either zero or one. If boolean was one then the statments XXXX would be in the model while if boolean was zero statements YYYY would be in the model. We have introduced the if statement as shorthand for this special use of the for loop. Using the if statement we would write

```
if(boolean) { XXXX}

if(1 - boolean) { YYYY }
```

An example of using the if statment in a model would be survival analysis where for each unit in a set there is an observed time at which the unit either fails or is censored and an indicator variable to inform whether the observed time is a failure or censoring event:

```
for(i in 1 : M) {
   # event = 1 for failure, event = 0 for censoring
   if(event[i]) { t[i] ~ dweib(r, mu) }
   if(1 - event[i]) { pred.fail[i] ~ dweib(r, mu)C(t[i],) }
}
```

## Branching

The BUGS language has a construct for branching, something like a case statement. We call this statement the mixture statement because it was first used to describe mixture models in the BUGS language. Consider the code snippet:

```
mu <- lambda[T]
T ~ dcat(P[1 : n])
```

this says that mu can have one of several values from the vector lambda depending on the value of the discrete variable T. We call the T variable an allocation variable. As T changes the value of mu switches. Note that T is part of the Bayesian model and will usually have a likelihood. It will have a distribution different from its prior distribution. For example consider the mixture model example (Eyes model included in the BUGS distribution):

```
model
{
   for( i in 1 : N ) {
      y[i] ~ dnorm(mu[i], tau)
      mu[i] <- lambda[T[i]]
```

```
        T[i] ~ dcat(P[])
    }
    P[1 : 2] ~ ddirich(alpha[])
    theta ~ dunif(0.0, 1000)
    lambda[2] <- lambda[1] + theta
    lambda[1] ~ dnorm(0.0, 1.0E-6)
    tau ~ dgamma(0.001, 0.001) sigma <- 1 / sqrt(tau)
}
```

The y[i] will modify the inferred values of the T[i] from its prior value a two state catagorical variable with prior probabilities P[1] and P[2] given by a dirichlet distribution.

Note that the alternatives choosen by the mixture statement need not all be stochastic or all logical assignments. Consider, for example, the cure model (Hearts model included in the BUGS distribution):

```
model
{
    for (i in 1 : N) {
        y[i] ~ dbin(q[i], t[i])
        q[i] <- P[state1[i]]
        state1[i] <- state[i] + 1
        state[i] ~ dbern(theta)
        t[i] <- x[i] + y[i]
    }
    P[1] <- p
    P[2] <- 0
    logit(p) <- alpha
    alpha ~ dnorm(0,1.0E-4)
    beta <- exp(alpha)
    logit(theta) <- delta
    delta ~ dnorm(0, 1.0E-4)
}
```

The variable q[i] can take two values: P[1] the stochastic variable p modeled by a logistic regression or P[2] the constant value zero.

It is sometimes possible to 'integrate' the allocation variables out of the model. This can make inference for these models more efficient.

## Data Transformations

If the compiler can prove that a logical assignment can be evaluated to a constant then the assignment is called a data transformation. This occurs if an assignment's right hand side does not depend on any variable quantities. The BUGS language has a general rule that there must

only be one assignment statement for each scalar or component of a tensor. This rule is slightly relaxed for data transformations. The language allows a logical assignment and a stochastic assignment to the same scalar or tensor component if and only if the logical assignment is a data transformation. Both JAGS and STAN have a seperate block in their programmes for data transformations.

## Generated Quantities

A BUGS language model can contain logical assignment relations for quantities that are not used in evaluating the joint probability distribution. They only need to be evaluated after the inference algorithm has finished its task. In the Rats model the statements

```
sigma <- 1 / sqrt(tau.c)

alpha0 <- alpha.c - xbar * beta.c
```

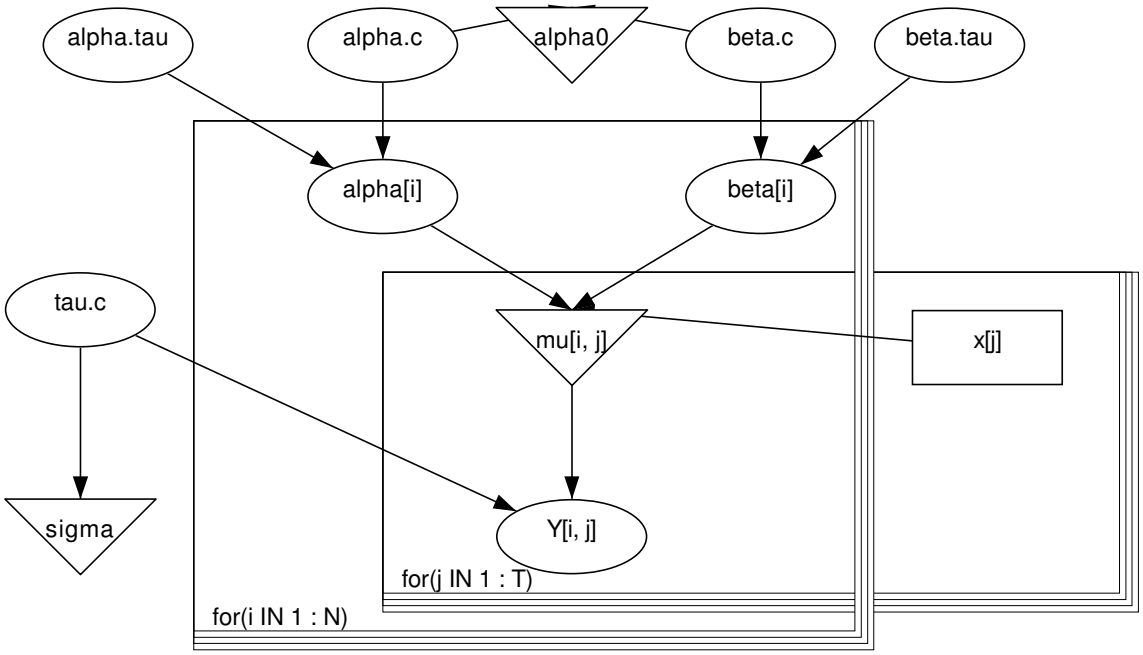
are not part of the statistical model. STAN has a special block for such parameters but I think that JAGS does not. The BUGS language compiler is able to tag the left hand sides of such assignments for special treatment.

Sometimes a model contains a stochastic relation where the left hand side quantity is not data and does not occur on the right hand side of any stochastic relation. We call such relations forward sampling relations and use a special sampling technique called forward sampling to generate values for such relations. STAN has a special notation to describe such nodes that allows the relation to be placed in the generated quantities block. The BUGScompiler is able to reconize such relations automatically.

## Statement Order

The BUGS language is used to describes a Bayesian probability model. It does not describe how the model should be fitted or what computations should be done with the model. The statements in a BUGS language model are not executed. Because the language is purely declarative the order of statements in a BUGS language model does not matter. The BUGS compiler produces three data structures from a BUGS language model: the name table, the parse tree and a directed acyclic graph of objects. These three data structures do not depend on the order of statements found in the BUGS language programme.

The BUGS sofware contains a graphics editor called DoodleBUGS that allows the user to draw graphs to represent simple Bayesian models. These graphs can be used as input to the BUGS compiler. Here is the graph for the Rats example. The Doodle is parsed to the same tree representation as the equivalent BUGS language model.




## Computing on the Compiler Output

The compiler can in addition analyze the object graph and build the data structures needed for an inference algorithm for that particular model from within a class of inference algorithms requested by the user. The BUGS software can produce data structures for component-wise MCMC (including Gibbs Sampling), Hamiltonian Monte Carlo, maximum likelihood estimation and Bayesian Variational Inference.

The object graph allows the simple calculation of conditional distributions used in component-wise MCMC. It also allows the calculation of the derivatative of these conditional distribution with respect to their prior parameter. These derivatives of the conditional distributions are identical with the derivatives of the joint probability distribution. The graph also allows the calculation of the joint probability distribution as required in Hamiltonian Monte Carlo.

There are two approaches to proccessing logical relations in the BUGS language: they can either be evaluated as needed or they can be evaluated before they are needed and their values stored for future use. The latest version of the BUGS software has adopted the second approach. All the nodes in the graphical model representing logical relations are placed into an array and sorted by their nesting level with the first array entries only depending on quantities defined by stochastic relations. Traversing this array and evaluating nodes gives up to date values to all logical relations. In Hamiltonian Monte Carlo all the stochastic values are changed in one block after which all the logical nodes need re-evalauting. The case for component-wise MCMC is less clear which is why earlier version of the BUGS software used an evaluate as needed approach. In the latest version of the software each variable stochastic node in the

model has its own array of logical dependent nodes which can be re-evaluated when the stochastic node's value changes.

Since the development of the MultiBUGS version of the software the BUGS compiler now contains tools that can split the graph data structure into shards where each shard runs on a seperate computer core. The data structures used in component wise MCMC and Hamiltonian Monte Carlo can also be split between multiple cores. This speeds up the running of large models.

## The Type System

The BUGS compiler uses the properties of the distribution on the right hand side of a stochastic assignment statement to make deductions about the variable on the left hand side. For example r ~ dbin(p, n) implies that r is integer valued while x ~ dnorm(mu, tau) implies that x is real valued. Some distributions are real valued but have support on a restricted range of the reals. For example p ~ dbeta(a, b) implies that p is real valued with support on unit interval while x ~ dgamma(r, lambda) implies that x is real valued but with support on the posative real line.

There are two multivariate distributions in the BUGS language, the Dirichlet and the Wishart that have support of a complex subspace of the reals. The Dirichlet has support on the unit symplex while the Wishart has support on the symetric posative definite matrices.

The BUGS compiler tries to infer if logical relations return an integer value by looking at whether their parents are integer valued and the operators that combine the values of their parents into the return value. For example in the cure model example above the logical relation state1[i] <- state[i] + 1 is integer valued because state[i] is a benoulli variable and therefore integer, the literal 1 is integer and the sum of two integers is an integer.

When the BUGS system reads in data from a data source it is able to tag whether the number read is an integer or a real and propagate this information to logical relations. Again using the cure model as an example the statement t[i] <- x[i] + y[i] is integer values because both x and y are data and are given as integers in the data source.

One special type of data is constants: that is just numbers with no associated distribution. Constants have many uses in BUGS language models but one of the most important is as covariates. A model can contain a large number of constants that are used as covariates. Because of the possible large numbers of these covariate type constants they are given special treatment by the BUGS compiler. If a name read in from a data source is only used on the right hand side of logical relations no nodes in the graphical model are created to hold its values they are directly incorporated in the objects that represent the right hand sides of the logical relations. For example the large Methadone model contains the regression

```
mu.indexed[i] <- beta[1] * x1[i] +
    beta[2] * x2[i] +
    beta[3] * x3[i] +
    beta[4] * x4[i] +
```

```
region.effect[region.indexed[i]] +
source.effect[region.indexed[i]] * source.indexed[i] +
person.effect[person.indexed[i]]
```

where i ranges from 1 to 240776. Not having to create node in the graphical model to represent x1, x2, x3, x4, region.indexed, source.index and persion.indexed saves a large amount of space.

In the BUGS language the type information is fine grained: each component of a tensor can have different type information. This is quite distinct from the situation in STAN and can make it much easier to specify a statistical model. One common case is where some components of a tensor have been observed while other components need to be estimated. The STAN documentation suggests workarounds for these situations but these are somewhat complex.

## Tools

The BUGS software contains a number of tools for examining the data structures produced by the compiler. The software also contains tools for examining how the various inference algorithms that BUGS uses are performing. These tools mostly work in terms of names in the BUGS language. The user can enquire about a name in the model (or some components of a tensor valued name) and receive output of various kinds -- for example the current value of the name's components, the types of node representing the name in the model graph or their average values.

A simple pretty printing tool can display the model's parse tree in a standard form. This pretty priting tools also works with models specified as DoodleBUGS graphs. A slight extension to this tool, which works on the complied model, produces a representation of the BUGS model closer to how the model would be specified in STAN code. This tools is called tentatively the STAN tool. BUGS models can contain names that are partially data and partially parameters which need special handling in STAN. I have not handled this case in the STAN tool that I have developed to translate BUGS language models into STAN language models.

For example the complied Rats BUGS model becomes in STAN like notation:

```
data{
        int N;
        int T;
        real Y[30,5];
        real x[5];
        real xbar;
}
parameters{
        real alpha[30];
        real alpha.c;
        real<lower = 0>  alpha.tau;
        real beta[30];
        real beta.c;
        real<lower = 0>  beta.tau;
        real<lower = 0>  tau.c;
```

```
}
transformed parameters{
        real mu[30,5];

        for( i in 1 : N ) {
            for( j in 1 : T ) {
                mu[i , j] = alpha[i] + beta[i] * (x[j] - xbar)
            }
        }
}
model{
        tau.c ~ dgamma(0.001, 0.001)
        alpha.c ~ dnorm(0.0, 1.0E-6)
        alpha.tau ~ dgamma(0.001, 0.001)
        beta.c ~ dnorm(0.0, 1.0E-6)
        beta.tau ~ dgamma(0.001, 0.001)
        for( i in 1 : N ) {
            alpha[i] ~ dnorm(alpha.c, alpha.tau)
            beta[i] ~ dnorm(beta.c, beta.tau)
        }
        for( i in 1 : N ) {
            for( j in 1 : T ) {
                Y[i , j] ~ dnorm(mu[i , j], tau.c)
            }
        }
}
generated quantities{
        real alpha0;
        real sigma;

        sigma = 1 / sqrt(tau.c)
        alpha0 = alpha.c - xbar * beta.c
}
```

Note the block structure, the variable declarations and the reordering of statements in the model block. The STAN style PPL has a more imperative flavour but does not contain information that is not in the compiled BUGS model.

## Work flow

The statistical model and data are presented to the BUGS system in a series of stages. In the first stage the model text is parsed into a tree and the name table constructed. The data is then loaded and checked against the model. The data can be split over a number of source. Once all the data has been loaded the model is compiled. Compiling builds the graphical model and does a large number of checks on the consistancy of the model. Finally initial values can be given or generated for the model.

The parse tree would be more accurately described as a list of trees with one link in the list for each statement in the model. Each statement in the list is put in a standard form where any for loops that enclose a stochastic or logical assignment are placed before the assignment statement. For example in the Rats model the first block of statements

```
for( i in 1 : N ) {
        for( j in 1 : T ) {
            Y[i , j] ~ dnorm(mu[i , j],tau.c)
            mu[i , j] <- alpha[i] + beta[i] * (x[j] - xbar)
        }
        alpha[i] ~ dnorm(alpha.c,alpha.tau)
        beta[i] ~ dnorm(beta.c,beta.tau)
    }
```

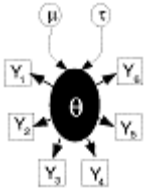
is treated as

```
for( i in 1 : N ) {
        for( j in 1 : T ) {
            Y[i , j] ~ dnorm(mu[i , j],tau.c)
        }
    }

for( i in 1 : N ) {
        for( j in 1 : T ) {
            mu[i , j] <- alpha[i] + beta[i] * (x[j] - xbar)
        }
}

for( i in 1 : N ) {
        alpha[i] ~ dnorm(alpha.c,alpha.tau)
    }

for( i in 1 : N ) {
        beta[i] ~ dnorm(beta.c,beta.tau)
    }
```

The compiler treats each of these standard form statements in the order in which they appear in the model. This does not affect the specification of a model but will effect the way any errors are reported. If for example there is an error in the last element of mu[i, j] and one in the first element of alpha[i] then the error for mu will be reported and not the one for alpha. If the error for mu is fixed then the error for alpha will be reported.

The compiler creates a node in the graphical model for each scalar name and each component of a tensor name in the BUGS language model. The compiler checks that only one node is created for each scalar name or component of a tensor name. Reading in a data source causes the compiler to create special nodes called constant nodes to hold the values of the data. The compiler processes logical relations before stochastic relations. Any logical relations that only have constant nodes on their right hand side become new constant nodes with the appropiate fixed value. Even if a logical relation can not be reduced to a constant some parts of the relation might be reduced to constants. For example in the Rats mode the relation for mu contains the factor (x[j] - xbar) if both x[j] and xbar are constatnts this factor is a constant and the expression can be simplified. If x[j] was equal to xbar the expression could be simplified to just alpha[i].

Any constant nodes that have an assocciated stochastic relation become data nodes in the graphical model. For example in the Rats model Y is given values in the data file. First each Y[i, j] is represented by a constant node then later by a normal node with the correct value. Any stochastic node that is not data needs its value inferring. The compiler will create the data structures needed by the inference algorithms. Initially a Gibbs sampling type inference algorithm is set up but this can later be replaced by Hamiltonian Monte Carlo or Variational Inference if the user desires




# Setting up the development tools

Andrew Thomas
MRC Biostatistics Unit
Cambridge

October 2021

## Contents



## BlackBox [top]

The OpenBUGS software is writen in the Component Pascal language originally developed by Oberonmicrosystem of Zurich. The language is a form of Pascal with single inheritance and modules. OpenBUGS uses the open source integrated development enviroment maintained by blackboxframework.org.

Download and install the integrated software development enviroment BlackBox component builder v1.7.x from https://blackboxframework.org/.

Unpack the source code for OpenBUGS into some directory different from where you have installed BlackBox. If you have installed BlackBox in a directory on the c drive it will be convenient to also install OpenBUGS in a directory on the same drive (say *c:/OpenBUGS*). Next edit the short cut to BlackBox so that its target contains */USE "c:/OpenBUGS"* (or where you have placed the source code) after the text BlackBox.exe. Note the use of forward slashes as separators in file paths. Next double click on the BlackBox icon on the desk top to start the development enviroment BlackBox.

Once you have got the BlackBox development enviroment installed you will be able to view the OpenBUGS source code files and the developer documentation. The high level developer documentation is in the directory Developer. The source code for each module is in a seperate file which are grouped into a number of subsystems and stored in the Mod subdirectory of each

subsystem. The programmer documentation for each source code file is stored in a seperate file in the corresponding Docu subdirectory. The OpenBUGS distribution also contains resource files that provide the user interface to the OpenBUGS software. These resource files are just documents and can be edited using tools in the BlackBox development enviroment.

If you want to work on MultiBUGS the multicore version of the OpenBUGS software install the Microsofts version of MPI. Then run the BlackBox software under MPI. To do this edit the short cut to BlackBox to look something like:
*"C:\Program Files\Microsoft MPI\Bin\mpiexec.exe" -n 1 "C:\BlackBox Component Builder 1.7.2\BlackBox.exe" /USE c:/OpenBUGS*
Double clicking on the BlackBox icon will then start the BlackBox as an MPI application.

The OpenBUGS distribution contains additional source code modules for MultiBUGS but these are not used in OpenBUGS only in MultiBUGS.

There are versions of BlackBox for many linux distributions but it is far above my pay grade to explain how to install them. If you are able to use Linux then you will be familiar with how to install Linux software. The source code for OpenBUGS is common to Windows and Linux with the exception of two modules Environment and BugsConfig. These two files contain both Windows and Linux source code.

The Component Pascal compiler produces the same compiled code irrespective of the operating system. The Windows and Linux versions of BlackBox use different linkers to produce executable code PE and ELF files repectively. BlackBox and OpenBUGS / MultiBUGS contain a small executable that loads compiled modules as needed at runtime. This small executable just contains a few basic services such as file handling and a linking loader. It rarely needs to be changed. Usually you can just make a copy the executable that comes with BlackBox, called BlackBox.exe on Windows, and call this copy OpenBUGS.exe. The MultiBUGS software uses a different small exectutable called MultiBUGS.exe this implements the same basic services as BlackBox.exe plus it starts up under MPI.

IF you want to link your own version of OpenBUGS.exe click into the round blob below (this executable only differs from the BlackBox exectutable in having a different icon --- Bugslogo.ico).

❗DevLinker.Link
OpenBUGS.exe := Kernel$+ Files HostFiles StdLoader
1 Bugslogo.ico 2 Doclogo.ico 3 SFLogo.ico 4 CFLogo.ico 5 DtyLogo.ico
1 Move.cur 2 Copy.cur 3 Link.cur 4 Pick.cur 5 Stop.cur 6 Hand.cur 7 Table.cur

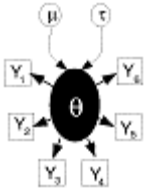
## Beautifier [top]

There is a tool called Beautifier in the web repository Component Pascal Collection (http://zinnamturm.eu/downloadsAC.htm#CpcBeautifier) which can be used to format Component Pascal Source code. We have used it to give the OpenBUGS source code a standard appearance.

## TAUCS library [top]

The OpenBUGS can make use of the TAUCS sparse matrix library (http://www.tau.ac.il/~stoledo/taucs). A Windows version of this library, libtaucs.dll, is distributed with OpenBUGS. This library is used in a sampling algorithm for Gaussian Markov Random Field (GMRF). Without TAUCS this sampling algorithm for GMRF will be much slower for high dimension problems. If you do not want OpenBUGS to use the TAUCS library just remove libtaucs.dll from the OpenBUGS directory.

We do not have Linux or a C compiler so we can not produce a Linux version of the TAUCS library. However the library is Open Source and can be downloaded from the web as source code. If you obtain a version of the TAUCS library in ELF format and you wish to use TAUCS to do sparse matrix algebra you might have to edit the module MathTaucsLib and replace the string "libtaucs" with the name of the corresponding Linux library..



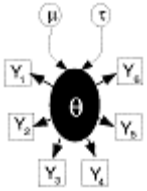

# Compiling the Source Code

To compile the complete Component Pascal source code open the make file and use the mouse to click into the round blob containing an ! mark. If the compiler detects an error in a module the compilation will stop at that module which will then be opened in a window with the error marked and described. If the directory into which the compiler wishes to place a code or symbol file does not exist the user will be prompted to create the directory.

The make file is set up to compile the Windows version of BUGS. To change the make file to compile the Linux version make the following three edits after Enviroment, BugsConfig and PlotsAxis change "OS":"Windows" to "OS":"Linux" so as to select the Linux source code for these two modules.

Note that if you change the interface to a module your new module may not be able to work with other developers versions of OpenBUGS. The Component Pascal module loader checks the "finger prints" (crytographic check sums) of the exported items of each module it loads against the "finger prints" the importing module expects and flags an error if these are inconsistant.

OpenBUGS can be run from inside the BlackBox developement enviroment. Indeed this has the advantage of making the full BlackBox debugger (DevDebug) available for diagnosing run time traps. The exact point in the source code where the problem occurs is usually detected!




# Compiling OpenBUGS

This document contains a list of source code modules which constitute the OpenBUGS software.

To produce an executable version of OpenBUGS the compiled form of some of these modules must be linked with certain compiled modules of Oberonmicrosystems BlackBox software. See the linking section of "Setting up the development tools"document for details.

The list of OpenBUGS modules follows after the string "DevCompiler.CompileThis". To compile all the modules click into the round blob.

To delete the interface definition files of all the modules in the OpenBUGS distribution repace the string "DevCompiler.CompileThis" with the string "BugsTools.DeleteSymbolFiles" before clicking into the round blob.

❗DevCompiler.CompileThis

Windows

HostMenus

Environment("OS":"Windows")
MpiMPI MpiMaster MpiWorker
MpiMslib MpiMsimp
MpiLibmpiso12 MpiImpmpiso12
MpiLibmpichso12 MpiImpmpichso12
MpiLibmpichso0 MpiImpmpichso0
MpiLibmpichso10 MpiImpmpichso10

MathSort MathMatrix MathSparsematrix MathTaucsLib MathTaucsImp MathDiagmatrix

MathODE MathFunctional MathRungeKutta45 MathAESolver MathIntegrate MathFunc

MathRandnum MathTT800

MathJacobi

MathBGR MathSmooth MathIS MathPolyaGamma

BugsMsg BugsRegistry BugsMappers BugsFiles

BugsInterpreter BugsScripts BugsScripting BugsTimer

BugsRandnum

GraphRules GraphGrammar GraphNodes GraphLoader GraphLogical GraphScalar GraphKernel GraphVector GraphStochastic GraphUnivariate GraphMultivariate GraphConjugateUV GraphConjugateMV GraphChain GraphMRF GraphGMRF GraphLinkfunc GraphFunctional GraphJacobi GraphDynamic GraphDAG GraphCheck

GraphVD GraphVDDescrete GraphVDContinuous

GraphMessages GraphResources

GraphConstant GraphDummy GraphDummyMV GraphSentinel

GraphMixture GraphMixtureVector GraphMarginal

GraphDeviance

MonitorSamples MonitorSamplesDisc MonitorSummary MonitorDeviance MonitorPlugin MonitorRanks MonitorModel

UpdaterUpdaters UpdaterLoader UpdaterMethods UpdaterUnivariate UpdaterMultivariate UpdaterContinuous UpdaterConjugateMV UpdaterMetropolisUV UpdaterMetropolisMV UpdaterRandWalkUV UpdaterAuxillary UpdaterEmpty UpdaterActions UpdaterJoint UpdaterMessages UpdaterExternal UpdaterResources

GraphBinary

GraphMAP GraphVI

BugsVersion BugsGrammar BugsNames BugsIndex BugsVariables BugsMonitors BugsAST BugsParser BugsEvaluate BugsArgs BugsOptimize BugsCPWrite BugsCPCompiler BugsNodes BugsCodegen BugsGraph BugsPrettyprinter BugsLatexprinter BugsTranslator

UpdaterHMC

BugsData BugsInterface BugsSerialize BugsRobjects BugsTraphandler1 BugsExternal BugsMAP BugsMessages BugsResources BugsInfo

BugsRectData BugsSplusData

BugsPartition BugsParallel

SamplesMonitors SamplesIndex SamplesStatistics SamplesInterface SamplesFormatted SamplesMessages SamplesResources

SummaryMonitors SummaryIndex SummaryInterface SummaryFormatted SummaryMessages SummaryResources

RanksMonitors RanksIndex RanksInterface RanksFormatted RanksMessages RanksResources

ModelsMonitors ModelsIndex ModelsInterface ModelsFormatted ModelsMessages ModelsResources

DevianceMonitors DevianceParents DeviancePlugin DevianceIndex DevianceInterface DevianceFormatted DevianceMessages DevianceResources

CorrelInterface CorrelFormatted CorrelMessages CorrelResources

BugsModules BugsLeafs BugsComponents BugsMaster BugsInfoDist

GraphPiecewise GraphODEmath GraphODElang GraphODEBlockL GraphODEBlockM

GraphAESolver GraphCloglog GraphCut GraphHalf GraphDensity GraphGammap GraphInprod GraphIntegrate GraphKepler GraphLog GraphLogdet GraphLogit GraphProbit GraphProduct GraphRanks GraphSumation GraphTable GraphPValue GraphReplicate GraphPi GraphSplinescalar

GraphEigenvals GraphInverse GraphODElangRK45 GraphODEBlockLRK45 GraphStick

GraphBern GraphBetaBinomial GraphBetaBinomialTilted GraphBinomial GraphCat GraphCat2 GraphFounder GraphGeometric GraphHypergeometric GraphMendelian GraphNegbin GraphPoisson GraphRecessive GraphZipf

GraphMultinom

GraphBeta GraphChisqr GraphDbexp GraphExp GraphF GraphFlat GraphGeneric GraphGEV GraphGPD GraphGamma GraphGengamma GraphHalfT GraphHazard GraphWeibullHazard GraphLogistic GraphLognorm GraphNormal GraphPriorNP GraphPareto GraphPolyaGamma GraphPolygene GraphSkewnorm GraphStable GraphStudent GraphT GraphTiltedBeta GraphTrapezium GraphTriangle GraphUniform GraphWeibull GraphWeibullShifted

GraphDirichlet GraphGPprior GraphMVNormal GraphMVT GraphRENormal GraphRandwalk GraphSample GraphStochtrend GraphWishart GraphFlexWishart

GraphCoSelection GraphSpline GraphSplinecon

GraphScalarT GraphVectorT GraphUnivariateT GraphScalartemp1 GraphVectortemp1 GraphUnivariatetemp1

UpdaterAM UpdaterKernel UpdaterDE UpdaterSlicebase

UpdaterForward

UpdaterCatagorical UpdaterDescreteSlice UpdaterMetbinomial UpdaterPoisson

UpdaterBeta UpdaterGamma UpdaterNormal UpdaterPareto

UpdaterGriddy UpdaterMetover UpdaterMetnormal UpdaterNaivemet

UpdaterRandEffect

UpdaterRejection

UpdaterSCAAR UpdaterSCDE UpdaterSDScale

UpdaterSlice UpdaterSlicegamma

UpdaterStage1 UpdaterStage1P UpdaterVD

UpdaterAMblock UpdaterKernelblock UpdaterDEblock UpdaterDirichlet UpdaterElliptical UpdaterEllipticalMVN UpdaterEllipticalD UpdaterGLM UpdaterMRFConstrain UpdaterGMRF UpdaterGMRFess UpdaterMultinomial UpdaterMVNormal UpdaterMVNLinear UpdaterStage1M UpdaterVDMVN UpdaterVDMVNContinuous UpdaterVDMVNDescrete UpdaterWishart

UpdaterUnivariateT UpdaterMultivariateT

SpatialExternal SpatialResources SpatialUVCAR
SpatialBound SpatialCARI1 SpatialCARNormal SpatialCARProper SpatialDiscKrig SpatialExpKrig SpatialMaternKrig SpatialPoissconv SpatialPoissconvaux

ReliabilityBS ReliabilityBurrXII ReliabilityBurrX ReliabilityExpPower ReliabilityExpoWeibull ReliabilityExtExp ReliabilityExtendedWeibull ReliabilityFlexibleWeibull ReliabilityGenExp ReliabilityGPWeibull ReliabilityGompertz ReliabilityGumbel ReliabilityInvGauss ReliabilityInvWeibull ReliabilityLinearFailure ReliabilityLogisticExp ReliabilityLogLogistic ReliabilityLogWeibull ReliabilityModifiedWeibull ReliabilitySystem ReliabilityWrapper ReliabilityExternal ReliabilityResources

PharmacoInputs PharmacoModel PharmacoExternal PharmacoResources

PharmacoPKIVbol1 PharmacoPKIVbol2 PharmacoPKIVbol3 PharmacoPKIVinf1 PharmacoPKIVinf2 PharmacoPKIVinf3 PharmacoPKFO1 PharmacoPKFO2 PharmacoPKFO3 PharmacoPKZO1 PharmacoPKZO2 PharmacoPKFOlag1 PharmacoPKFOlag2 PharmacoPKZOlag1 PharmacoPKZOlag2 PharmacoPKIVbol1ss PharmacoPKIVbol2ss PharmacoPKIVbol3ss PharmacoPKIVinf1ss PharmacoPKIVinf2ss PharmacoPKIVinf3ss PharmacoPKFO1ss PharmacoPKFO2ss PharmacoPKZO1ss

PharmacoPKZO2ss PharmacoPKFOlag1ss PharmacoPKFOlag2ss PharmacoPKZOlag1ss PharmacoPKZOlag2ss PharmacoSum

PKBugsScanners PKBugsNames PKBugsData PKBugsParse PKBugsCovts PKBugsPriors PKBugsNodes PKBugsTree PKBugsMessages PKBugsResources

DiffExternal DiffResources

DiffExponential DiffLotkaVolterra DiffFiveCompModel DiffChangePoints DiffHPS_V2_FB

MapsMessages MapsResources

BugsDialog

BugsStartup Init

UpdaterSettings

DevDebug

BugsVI BugsCmds BugsBatch BugsConfig("OS":"Windows")

DoodleNodes DoodlePlates DoodleModels DoodleMenus DoodleDialog DoodleViews DoodleParser DoodleCmds DoodleMessages DoodleResources

BugsInfodebug BugsDecode BugsDocu BugsSearch

PlotsAxis("OS":"Windows") PlotsViews PlotsDialog PlotsNomaxis PlotsStdaxis PlotsEmptyaxis

SamplesViews SamplesPlots SamplesCmds SamplesCorrelat SamplesDensity SamplesDiagnostics SamplesHistory SamplesQuantiles SamplesTrace SamplesJumpdist SamplesAccept

SummaryCmds

RanksDensity RanksCmds

ModelsCmds

DevianceCmds

CompareViews CompareBoxplot CompareCaterpillar CompareModelFit CompareScatter CompareDenstrip CompareCmds

CorrelBivariate CorrelMatrix CorrelPlots CorrelCmds

MapsMap MapsImporter MapsIndex MapsAdjacency MapsViews MapsViews1 MapsCmds

MapsArcinfo MapsEpimap MapsSplus

PKBugsCmds

StdCmds1

BugsC

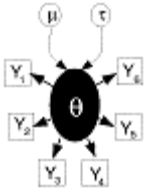
HtmlExporter

BugsPackage BugsTools BugsLeafs1 BugsComponents1 BugsTypes

MathSeeds

ParallelFiles ParallelRandnum ParallelActions ParallelHMC ParallelLoop ParallelWorker ParallelDebug ParallelWorker1

TestUtil TestRats TestDogs TestSeeds TestTdof TestStudent




# OpenBUGS Software Subsystems

Index of the various subsystems that constitute the OpenBUGS software.

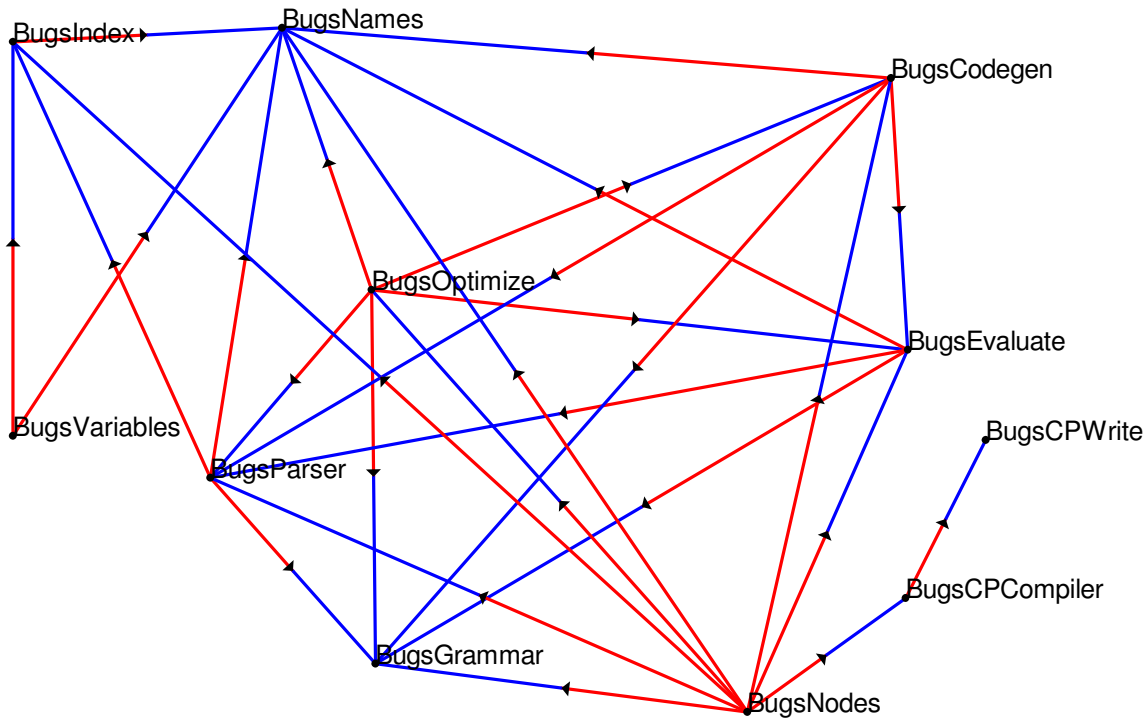
| | | |
|---|---|---|
| Bugs | Dev-Man | Sys-Map |
| Compare | Dev-Man | Sys-Map |
| Correl | Dev-Man | Sys-Map |
| Deviance | Dev-Man | Sys-Map |
| Diff | Dev-Man | Sys-Map |
| Doodle | Dev-Man | Sys-Map |
| Graph | Dev-Man | Sys-Map |
| Map | Dev-Man | Sys-Map |
| Math | Dev-Man | Sys-Map |
| Models | Dev-Man | Sys-Map |
| Monitors | Dev-Man | Sys-Map |
| Mpi | Dev-Man | Sys-Map |
| Parallel | Dev-Man | Sys-Map |
| Pharmaco | Dev-Man | Sys-Map |
| PKBugs | Dev-Man | Sys-Map |
| Plots | Dev-Man | Sys-Map |
| Ranks | Dev-Man | Sys-Map |
| Reliability | Dev-Man | Sys-Map |
| Samples | Dev-Man | Sys-Map |

| | | |
|---|---|---|
| Spatial | Dev-Man | Sys-Map |
| Summary | Dev-Man | Sys-Map |
| Updater | Dev-Man | Sys-Map |

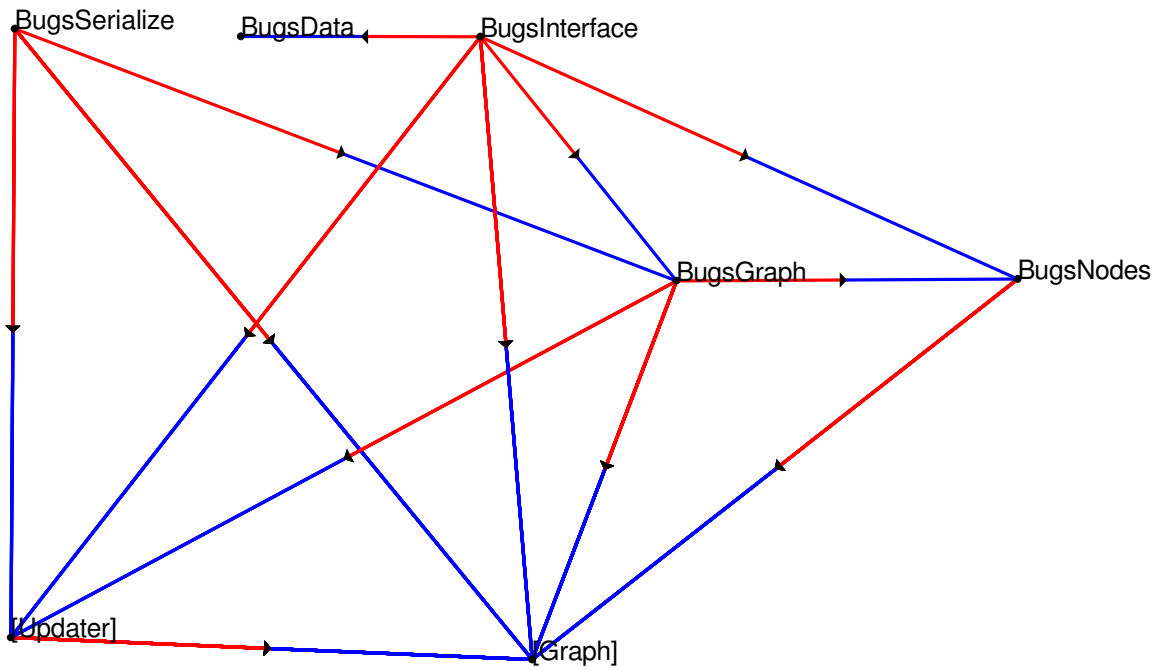
# Bugs Subsystem

# Developer Manual

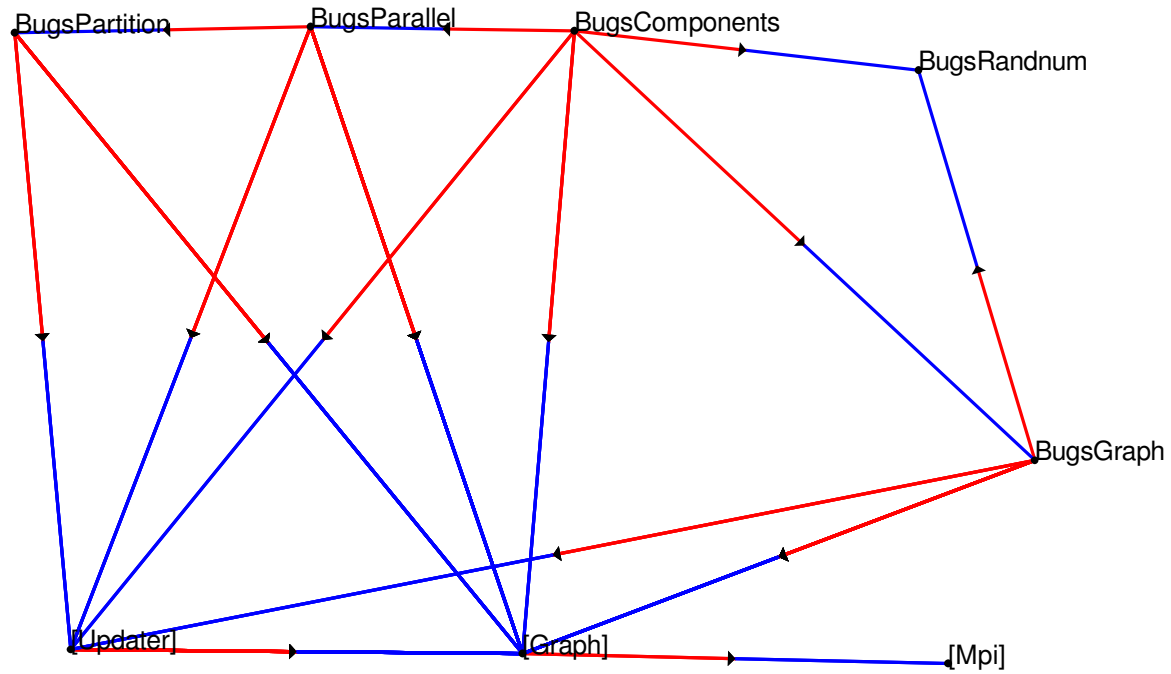
The Bugs subsystem handles communication between the statistical model specified in the BUGS language and the low level data structures that represent the graphical model. The most substantial part of the Bugs subsystem is a compiler for the BUGS language which creates the low level graphical model as its executable code. The module BugsNodes creates the nodes of the graphical model and specifies their parents. The nodes created by BugsNodes are stored in an index and can be retrieved by using their name as used in the BUGS language specification of the model.

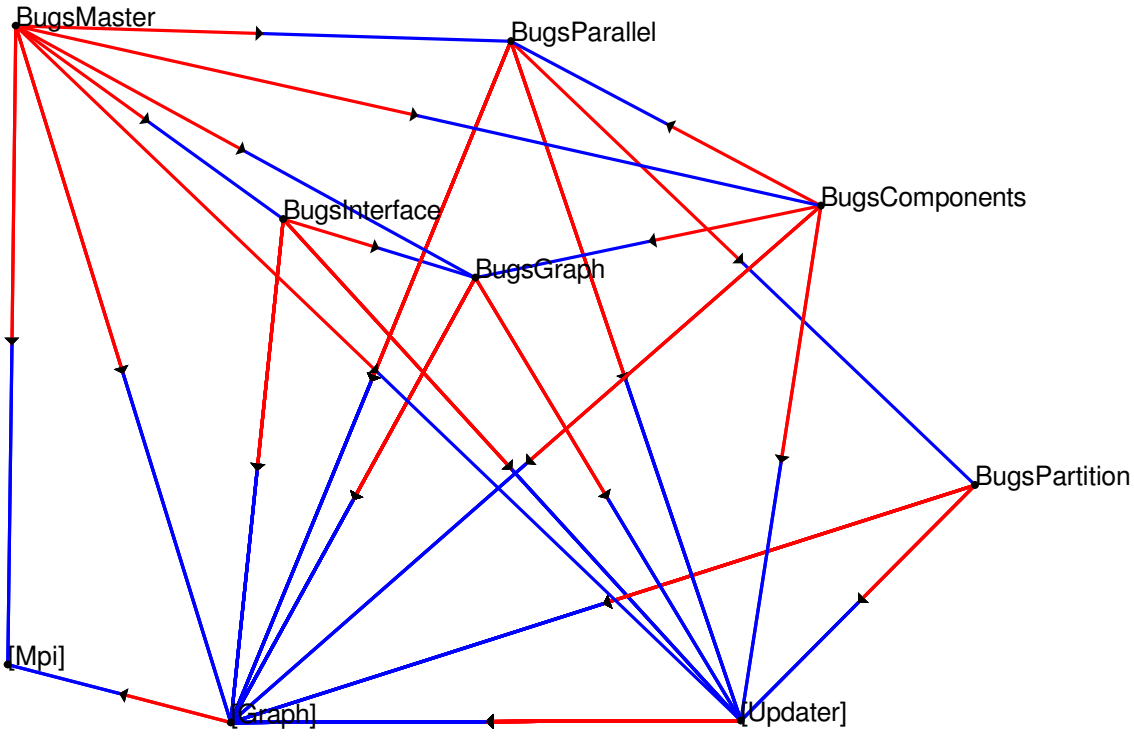



The module BugsGraph takes the nodes created by BugsNodes and where they are stochastic parameter nodes works out their dependent logical and stochastic children in the graphical model. MCMC updater objects are then created for each stochastic parameter node. BugsMonitors creates various types of monitors to watch how the MCMC updater objects are changing quantities in the BUGS language model. BugsInterface provides a programmable interface to the BUGS compiler.




There is a parallel implementation of the Bugs software called MultiBUGS. This is structured as a master worker system. The master software consists of four modules BugsPartition, BugsParallel, BugsComponents and BugsMaster. The first three modules are only imported by BugsMaster. BugsMaster instantiates a hook in BugsInterface and is loaded dynamically on demand. In this way the core of the BUGS software is unaware of the parallel implementation MultiBUGS and does not need to have an implementation of the MPI message passing library available.



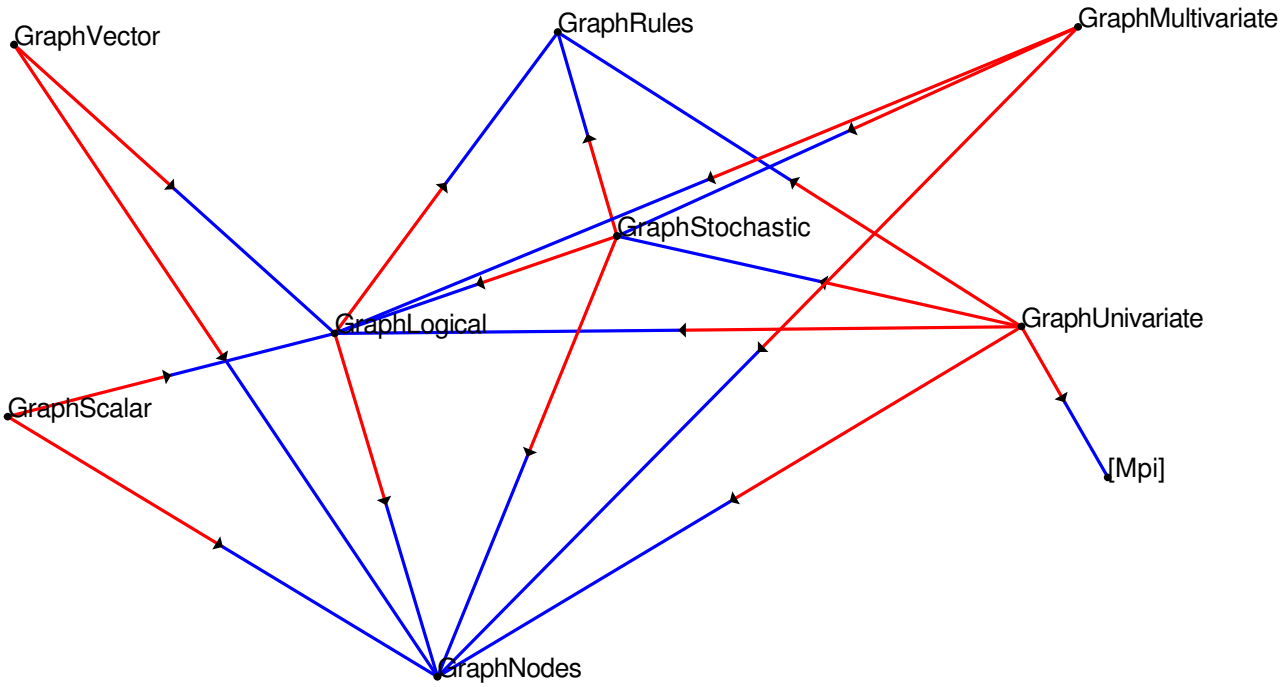




The BUGS software uses a graphical user interface with menus and dialogs. Each subsystem of the BUGS software has a module called Cmds, standing for commands, which has code to implement the graphical user interface for the functionality in that subsytem. For some purposes a scripting interface is more appropiate. The BUGS has a scripting language which is implemented in a small part of the Bugs subsystem. The module BugsScripting translates commands in the scripting language into small snippets of Component Pascal code which are

then executed using Component Pascal's metaprogramming features. The scripting language works in effect by filling in dialog boxes and pressing buttons in the appropiate Cmds modules.

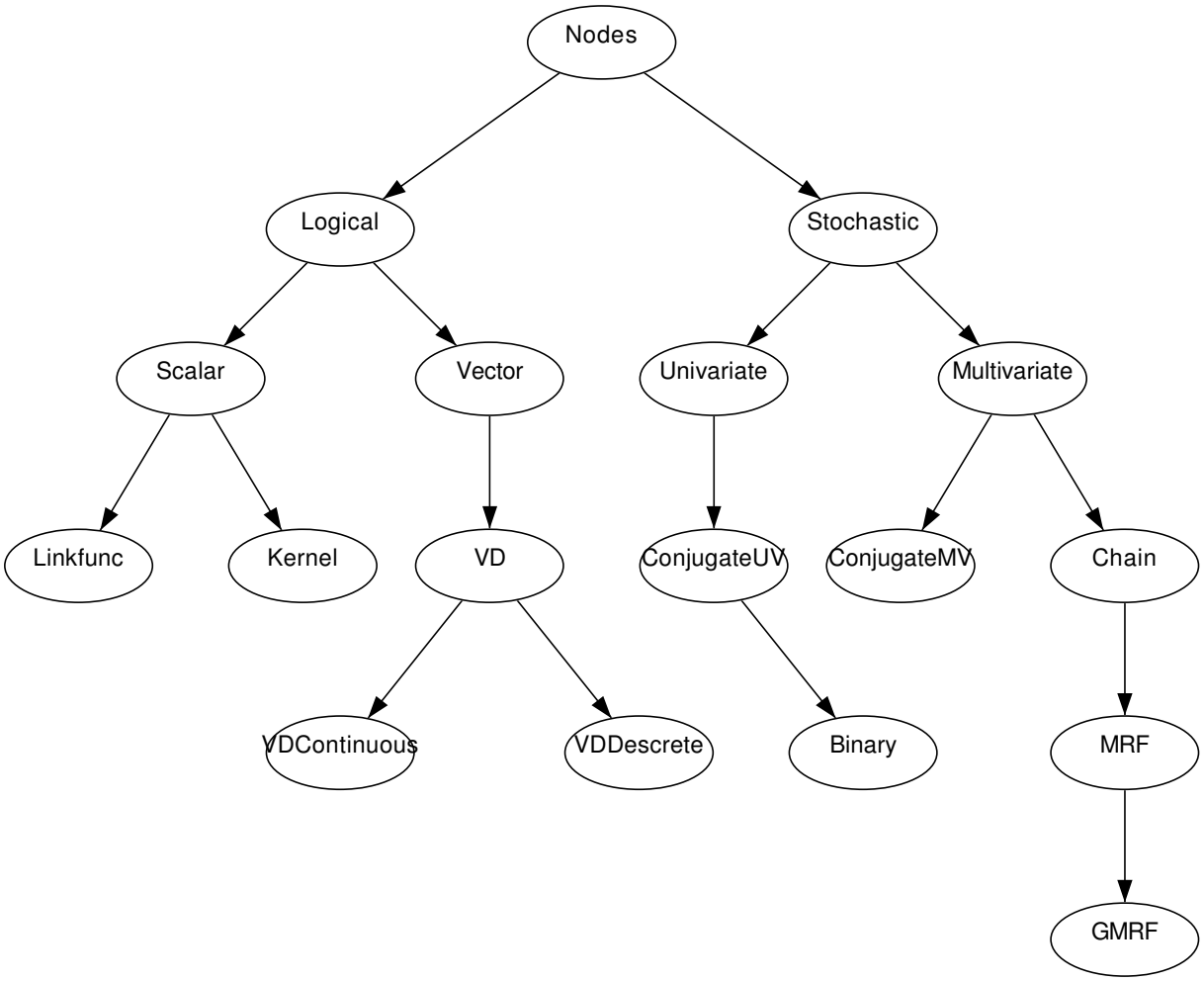
# Graph Subsystem

# Developer Manual




The Bayesian statistical model is represented in the OpenBUGS software as a graph of objects. Each node in the graph contains pointers to the node's direct parents. Some nodes in the graph also contain pointers to their children nodes. The different types of objects used to build the graph are all derived from a base type and are implemented in modules grouped in the Graph subsystem. The different types of node can be arranged in a type tree. The types near the root of the tree are abstract classes: objects of abstract class can not be created: they are design patterns.

In the diagram below all the node types are abstract. Generally the leafs of this tree are specialized to concrete types which can be used as modelling components. Moving down the class tree new features are added to the parent class along each branch. These features can be new parameter fields or new methods. Each type is implemented in a seperate module in the Graph subsystem. In the diagram of the type tree the vertices are labeled by the name of the module where the type is implemented. For example the Component Pascal code for the base type is implemented in module GraphNodes in source code file /Graph/Mod/Nodes.odc. The full name for the base type is GraphNodes.Node. Similarly the type for conjugate multivariate distributions is implemented in module GraphConjugateMV in source code file /Graph/Mod/ConjugateMV.odc and has the full name GraphConjugateMV.Node.




The interface of a type consists of all fields and methods of the type that are visible outside the module that implements the type. If a type has a field that is visible outside the module where it is defined but can only be modified in the module where it is defined then it is tagged by a minus mark. In Component Pascal there are two distinct concepts of type interface: client interface and extension interface. The client inteface of a type lists those methods of a type that can be used outside the module in which the class is defined. The extension interface lists the abstract methods of a type which must be implemented in all concrete realizations of the class. Note that there can be methods in the extension interface of a type that can not be used outside the module in which the class is defined. Any method in the extension interface that is not in the client interface is distinguished with a minus mark. These methods are used by the framework but can not be used by the client.

The abstract type GraphNodes.Node has a small interface, the client interface is

```
TYPE
   Node = POINTER TO ABSTRACT RECORD
      label-: INTEGER;
      props: SET;
      value: REAL;
      (node: Node) Check (): SET, NEW, ABSTRACT;
      (node: Node) Install (OUT install: ARRAY OF CHAR), NEW, ABSTRACT;
      (node: Node) Parents (all: BOOLEAN): List, NEW, ABSTRACT;
      (node: Node) Representative (): Node, NEW, ABSTRACT;
      (node: Node) Set (IN args: Args; OUT res: SET), NEW, ABSTRACT;
      (node: Node) Size (): INTEGER, NEW, ABSTRACT
   END;
```

and the extension interface is

```
TYPE
   Node = POINTER TO ABSTRACT RECORD
      label-: INTEGER;
      props: SET;
      value: REAL;
      (node: Node) Check (): SET, NEW, ABSTRACT;
      (node: Node) ExternalizeNode- (VAR wr: Stores64.Writer), NEW, ABSTRACT;
      (node: Node) InitNode-, NEW, ABSTRACT;
      (node: Node) Install (OUT install: ARRAY OF CHAR), NEW, ABSTRACT;
      (node: Node) InternalizeNode- (VAR rd: Stores64.Reader), NEW, ABSTRACT;
      (node: Node) Parents (all: BOOLEAN): List, NEW, ABSTRACT;
      (node: Node) Representative (): Node, NEW, ABSTRACT;
      (node: Node) Set (IN args: Args; OUT res: SET), NEW, ABSTRACT;
      (node: Node) Size (): INTEGER, NEW, ABSTRACT
   END;
```

The extension interface has three methods: ExternalizeNode, InitNode and InternalizeNode that have a minus mark and can not be used by the client. The nine methods of abstract type GraphNodes.Node can be grouped into a small number of functional categories: information, initialization, serialization and topology.

The abstract type GraphLogical.Node is used to represent logical relations in the BUGS language. It is derived from the base type GraphNodes.Node and introduces additional fields and methods. The new methods are associated with evaluating and differentiating logical expressions. The client interface interface is

```
TYPE
   Node = POINTER TO ABSTRACT RECORD (GraphNodes.Node)
      (* GraphNodes.Node *) label-: INTEGER;
      (* GraphNodes.Node *) props: SET;
      (* GraphNodes.Node *) value: REAL;
      work-: POINTER TO ARRAY OF REAL;
      parents-: GraphNodes.Vector;
      nesting-: INTEGER;
      (node: GraphNodes.Node) Check (): SET, NEW, ABSTRACT;
      (node: GraphNodes.Node) Install (OUT install: ARRAY OF CHAR), NEW, ABSTRACT;
      (node: GraphNodes.Node) Parents (all: BOOLEAN): GraphNodes.List, NEW, ABSTRACT;
```

```
    (node: GraphNodes.Node) Set (IN args: GraphNodes.Args; OUT res: SET), NEW, ABSTRACT;
    (node: GraphNodes.Node) Size (): INTEGER, NEW, ABSTRACT;
    (node: Node) Representative (): GraphLogical.Node, ABSTRACT;
  END;
```

and the extension interface is

```
TYPE
  Node = POINTER TO ABSTRACT RECORD (GraphNodes.Node)
    (* GraphNodes.Node *) label-: INTEGER;
    (* GraphNodes.Node *) props: SET;
    (* GraphNodes.Node *) value: REAL;
    work-: POINTER TO ARRAY OF REAL;
    parents-: GraphNodes.Vector;
    nesting-: INTEGER;
    (node: GraphNodes.Node) Check (): SET, NEW, ABSTRACT;
    (node: GraphNodes.Node) Install (OUT install: ARRAY OF CHAR), NEW, ABSTRACT;
    (node: GraphNodes.Node) Parents (all: BOOLEAN): GraphNodes.List, NEW, ABSTRACT;
    (node: GraphNodes.Node) Set (IN args: GraphNodes.Args; OUT res: SET), NEW, ABSTRACT;
    (node: GraphNodes.Node) Size (): INTEGER, NEW, ABSTRACT;
    (node: Node) AllocateDiffs- (numDiffs: INTEGER), NEW, EMPTY;
    (node: Node) Evaluate-, NEW, ABSTRACT;
    (node: Node) EvaluateClass-, NEW, ABSTRACT;
    (node: Node) EvaluateDiffs-, NEW, ABSTRACT;
    (node: Node) ExternalizeLogical- (VAR wr: Stores64.Writer), NEW, ABSTRACT;
    (node: Node) InitLogical-, NEW, ABSTRACT;
    (node: Node) InternalizeLogical- (VAR rd: Stores64.Reader), NEW, ABSTRACT;
    (node: Node) Link-, NEW, EMPTY;
    (node: Node) Representative (): GraphLogical.Node, ABSTRACT
  END;
```

The extension interface has eight methods that have the minus mark and thus can not be called by the client. The client interface has all the mehods that the client interface of GraphNodes.Node but note the change to the signiture of Representative. The base type of logical nodes also has three additional fields: work, parents and nesting.

Note that the methods ExternalizeNode, InitNode and InternalizeNode are no longer in the extension interface: they have been given concrete implementations. The BUGS software often uses the chain of command software design pattern. For example the ExternalizeNode method externalizes the work, parents and nesting fields and then call the ExternalizeLogical method which when implemented will externalize the fields of any specializations of GraphLogical.Node.

The abstract type GraphStochastic.Node is used to represent stochastic relations in the BUGS language. It is also derived from the base type GraphNodes.Node. It introduces several methods associated with the new functional category of probability density distributions. The client interface is

```
TYPE
  Node = POINTER TO ABSTRACT RECORD (GraphNodes.Node)
    (* GraphNodes.Node *) label-: INTEGER;
    (* GraphNodes.Node *) props: SET;
```

```
    (* GraphNodes.Node *) value: REAL;
    depth-, classConditional-: INTEGER;
    children-: GraphStochastic.Vector;
    dependents-: GraphLogical.Vector;
    (node: GraphNodes.Node) Check (): SET, NEW, ABSTRACT;
    (node: GraphNodes.Node) Install (OUT install: ARRAY OF CHAR), NEW, ABSTRACT;
    (node: GraphNodes.Node) Parents (all: BOOLEAN): GraphNodes.List, NEW, ABSTRACT;
    (node: GraphNodes.Node) Set (IN args: GraphNodes.Args; OUT res: SET), NEW, ABSTRACT;
    (node: GraphNodes.Node) Size (): INTEGER, NEW, ABSTRACT;
    (node: Node) Bounds (OUT lower, upper: REAL), NEW, ABSTRACT;
    (node: Node) CanSample (multiVar: BOOLEAN): BOOLEAN, NEW, ABSTRACT;
    (node: Node) ClassifyLikelihood (parent: GraphStochastic.Node): INTEGER, NEW, ABSTRACT;
    (node: Node) ClassifyPrior (): INTEGER, NEW, ABSTRACT;
    (node: Node) Deviance (): REAL, NEW, ABSTRACT;
    (node: Node) DiffLogConditional (): REAL, NEW, ABSTRACT;
    (node: Node) DiffLogLikelihood (x: GraphStochastic.Node): REAL, NEW, ABSTRACT;
    (node: Node) DiffLogPrior (): REAL, NEW, ABSTRACT;
    (node: Node) InvMap (y: REAL), NEW, ABSTRACT;
    (node: Node) Location (): REAL, NEW, ABSTRACT;
    (node: Node) LogDetJacobian (): REAL, NEW, ABSTRACT;
    (node: Node) LogLikelihood (): REAL, NEW, ABSTRACT;
    (node: Node) LogPrior (): REAL, NEW, ABSTRACT;
    (node: Node) Map (): REAL, NEW, ABSTRACT;
    (node: Node) Representative (): GraphStochastic.Node, ABSTRACT;
    (node: Node) Sample (OUT ok: BOOLEAN), NEW, ABSTRACT
  END;
```

and the extension interface

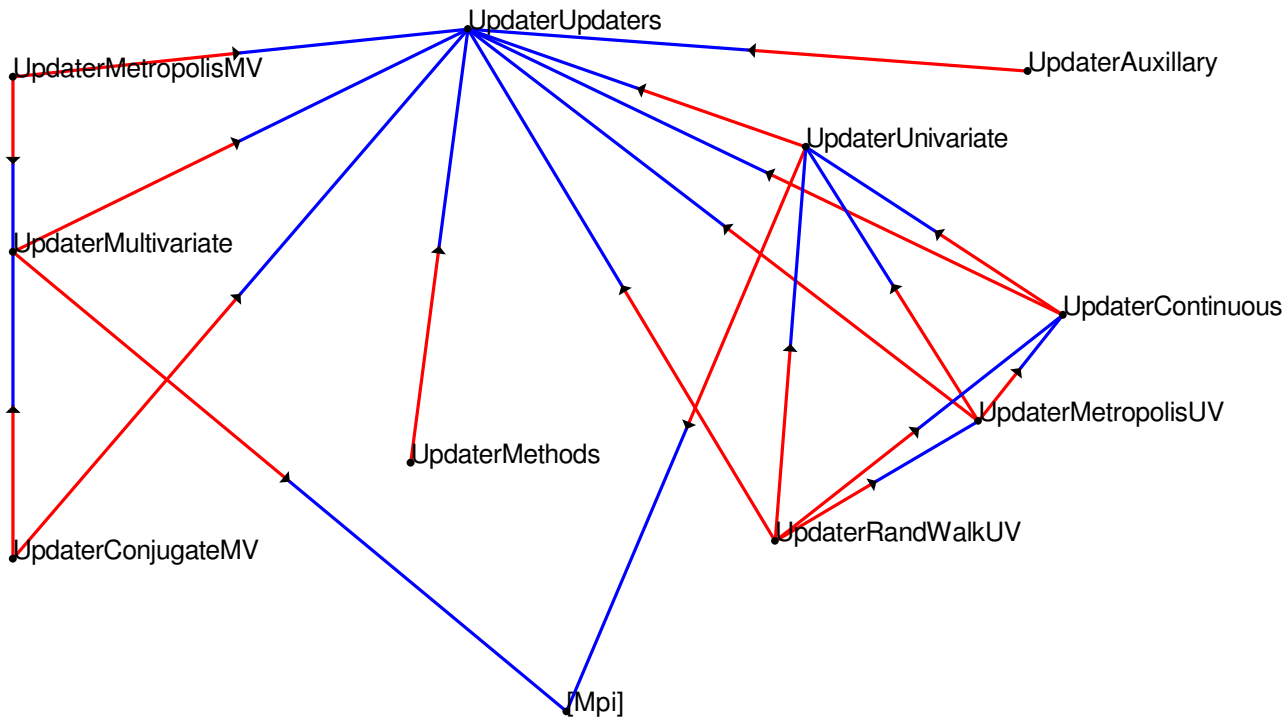
```
TYPE
  Node = POINTER TO ABSTRACT RECORD (GraphNodes.Node)
    (* GraphNodes.Node *) label-: INTEGER;
    (* GraphNodes.Node *) props: SET;
    (* GraphNodes.Node *) value: REAL;
    depth-, classConditional-: INTEGER;
    children-: GraphStochastic.Vector;
    dependents-: GraphLogical.Vector;
    (node: GraphNodes.Node) Check (): SET, NEW, ABSTRACT;
    (node: GraphNodes.Node) Install (OUT install: ARRAY OF CHAR), NEW, ABSTRACT;
    (node: GraphNodes.Node) Parents (all: BOOLEAN): GraphNodes.List, NEW, ABSTRACT;
    (node: GraphNodes.Node) Set (IN args: GraphNodes.Args; OUT res: SET), NEW, ABSTRACT;
    (node: GraphNodes.Node) Size (): INTEGER, NEW, ABSTRACT;
    (node: Node) Bounds (OUT lower, upper: REAL), NEW, ABSTRACT;
    (node: Node) CanSample (multiVar: BOOLEAN): BOOLEAN, NEW, ABSTRACT;
    (node: Node) ClassifyLikelihood (parent: GraphStochastic.Node): INTEGER, NEW, ABSTRACT;
    (node: Node) ClassifyPrior (): INTEGER, NEW, ABSTRACT;
    (node: Node) Deviance (): REAL, NEW, ABSTRACT;
    (node: Node) DiffLogConditional (): REAL, NEW, ABSTRACT;
    (node: Node) DiffLogLikelihood (x: GraphStochastic.Node): REAL, NEW, ABSTRACT;
    (node: Node) DiffLogPrior (): REAL, NEW, ABSTRACT;
    (node: Node) ExternalizeStochastic- (VAR wr: Stores64.Writer), NEW, ABSTRACT;
    (node: Node) InitStochastic-, NEW, ABSTRACT;
    (node: Node) InternalizeStochastic- (VAR rd: Stores64.Reader), NEW, ABSTRACT;
    (node: Node) InvMap (y: REAL), NEW, ABSTRACT;
    (node: Node) IsLikelihoodTerm- (): BOOLEAN, NEW, ABSTRACT;
```

```
    (node: Node) Location (): REAL, NEW, ABSTRACT;
    (node: Node) LogDetJacobian (): REAL, NEW, ABSTRACT;
    (node: Node) LogLikelihood (): REAL, NEW, ABSTRACT;
    (node: Node) LogPrior (): REAL, NEW, ABSTRACT;
    (node: Node) Map (): REAL, NEW, ABSTRACT;
    (node: Node) Representative (): GraphStochastic.Node, ABSTRACT;
    (node: Node) Sample (OUT ok: BOOLEAN), NEW, ABSTRACT
  END;
```

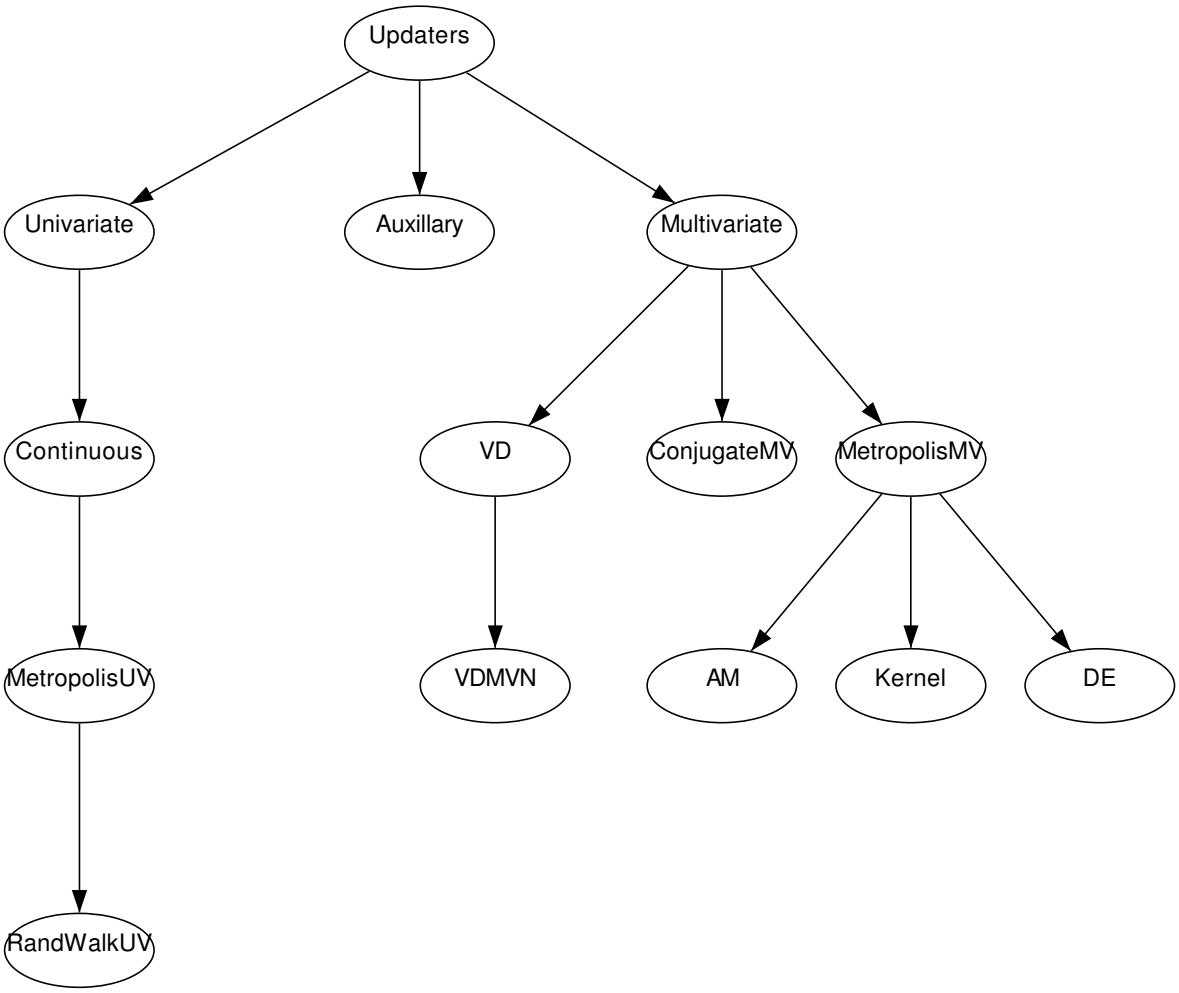
The client interface has all the methods of GraphNodes.Node with the return type of Representative again being specialized. Again ExternalizeNode, InitNode and InternalizeNode are no longer in the extension interface: they have been given concrete implementations. They call ExternalizeStochastic, InitStochastic and InternalizeStochastic respectively.

# Updater Subsystem

# Developer Manual




The OpenBUGS software uses Markov Chain Monte Carlo, MCMC, to make statistical inference for models specified in the BUGS language. The MCMC algorithms used are implemented in the Updater subsystem. This subsystem uses the graph build in the Graph subsystem to do basic probability calculations. Each MCMC algorithm is implemented as a type derived from the base type UpdaterUpdaters.Updater in source code file Updater/Mod/Updaters.odc. Objects of type UpdaterUpdaters.Updater change the value of one or more nodes in the graphical model. Objects of type UpdaterUnivariate.Updater change the value of a single node in the graphical model while objects of type UpdaterMultivariate.Updater change the value of a block of nodes.




The interface of a type consists of all fields and methods of the type that are visible outside the module that implements the type. In Component Pascal there are two distinct concepts of type interface: client interface and extension interface. The client inteface of a type lists those methods of a type that can be used outside the module in which the class is defined. The extension interface lists the abstract methods of a type which must be implemented in all concrete realizations of the class. Note that there can be methods in the extension interface of a type that can not be used outside the module in which the class is defined. Any method in the extension interface that is not in the client interface is distinguished with a minus mark. These methods are used by the framework but can not be used by the client.

The abstract type UpdaterUpdaters.Updater has a small interface, the client interface is

```
TYPE
   Updater = POINTER TO ABSTRACT RECORD
      (updater: Updater) Children (): GraphStochastic.Vector, NEW, ABSTRACT;
```

```
    (updater: Updater) Depth (): INTEGER, NEW, ABSTRACT;
    (updater: Updater) GenerateInit (fixFounder: BOOLEAN; OUT ok: BOOLEAN), NEW, ABSTRACT;
    (updater: Updater) Install (OUT install: ARRAY OF CHAR), NEW, ABSTRACT;
    (updater: Updater) IsAdapting (): BOOLEAN, NEW, ABSTRACT;
    (updater: Updater) LogConditional (): REAL, NEW, ABSTRACT;
    (updater: Updater) LogLikelihood (): REAL, NEW, ABSTRACT;
    (updater: Updater) LogPrior (): REAL, NEW, ABSTRACT;
    (updater: Updater) Prior (index: INTEGER): GraphStochastic.Node, NEW, ABSTRACT;
    (updater: Updater) Sample (overRelax: BOOLEAN; OUT ok: BOOLEAN), NEW, ABSTRACT;
    (updater: Updater) Size (): INTEGER, NEW, ABSTRACT
  END;
```

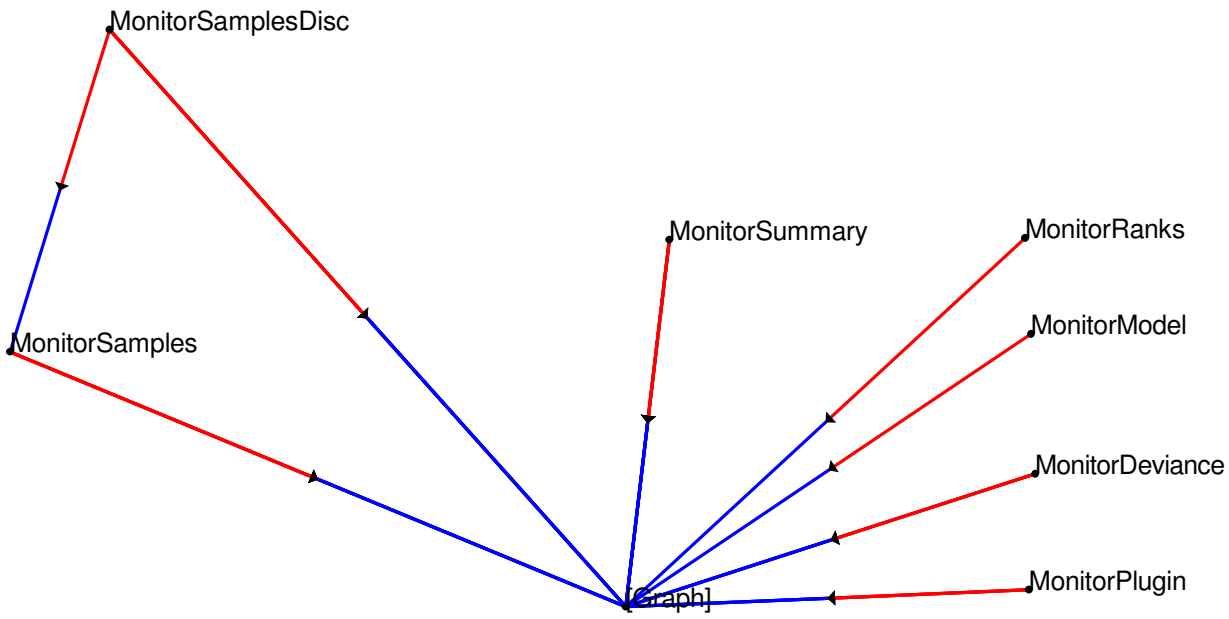
and the extension interface is

```
TYPE
  Updater = POINTER TO ABSTRACT RECORD
    (updater: Updater) Children (): GraphStochastic.Vector, NEW, ABSTRACT;
    (updater: Updater) Clone- (): UpdaterUpdaters.Updater, NEW, ABSTRACT;
    (updater: Updater) CopyFrom- (source: UpdaterUpdaters.Updater), NEW, ABSTRACT;
    (updater: Updater) Depth (): INTEGER, NEW, ABSTRACT;
    (updater: Updater) Externalize- (VAR wr: Stores64.Writer), NEW, ABSTRACT;
    (updater: Updater) ExternalizePrior- (VAR wr: Stores64.Writer), NEW, ABSTRACT;
    (updater: Updater) GenerateInit (fixFounder: BOOLEAN; OUT ok: BOOLEAN), NEW, ABSTRACT;
    (updater: Updater) Initialize-, NEW, ABSTRACT;
    (updater: Updater) Install (OUT install: ARRAY OF CHAR), NEW, ABSTRACT;
    (updater: Updater) Internalize- (VAR rd: Stores64.Reader), NEW, ABSTRACT;
    (updater: Updater) InternalizePrior- (VAR rd: Stores64.Reader), NEW, ABSTRACT;
    (updater: Updater) IsAdapting (): BOOLEAN, NEW, ABSTRACT;
    (updater: Updater) LogConditional (): REAL, NEW, ABSTRACT;
    (updater: Updater) LogLikelihood (): REAL, NEW, ABSTRACT;
    (updater: Updater) LogPrior (): REAL, NEW, ABSTRACT;
    (updater: Updater) Optimize-, NEW, EMPTY;
    (updater: Updater) Prior (index: INTEGER): GraphStochastic.Node, NEW, ABSTRACT;
    (updater: Updater) Sample (overRelax: BOOLEAN; OUT ok: BOOLEAN), NEW, ABSTRACT;
    (updater: Updater) SetPrior- (prior: GraphStochastic.Node), NEW, ABSTRACT;
    (updater: Updater) Size (): INTEGER, NEW, ABSTRACT
  END;
```

# Monitor Subsystem

# Developer Manual

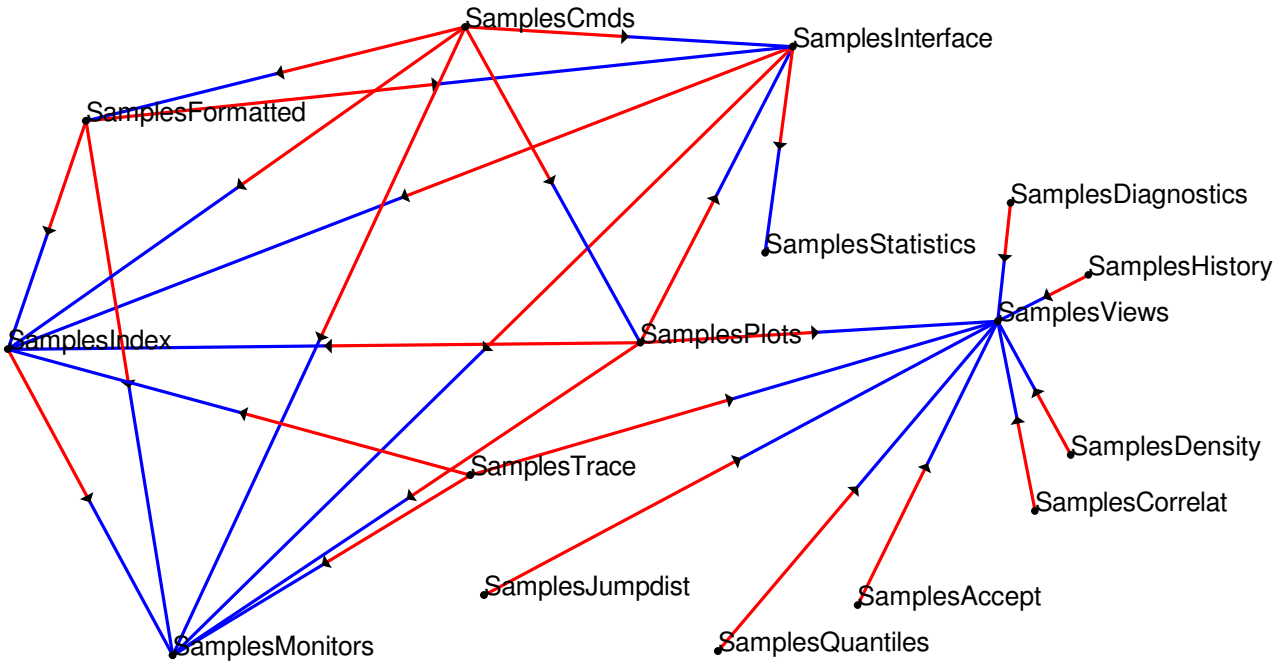
The Monitor subsystem whatches the graph of the statistical model and records changes in values of its nodes. The modules in this subsystem operate on the low level graphical model defined in terms of the Graph subsystem. They can be aggregated to operate on the level of the BUGS language model. Various types of monitor are implemented in this subsystem. Each of these types is elaboarated in a seperate subsystem to operate on the BUGS language model. The Samples subsystem uses either MonitorSamples or MonitorSamplesDisc, the Summary subsystem MonitorSummary, the Ranks subsystem MonitorRanks, the Model subsystem MontorModel and the Deviance subsystem MonitorDeviance and MonitorPlugin.



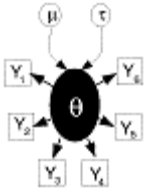

When the graph of the statistical model is distributed the Monitor subsystem works in a slightly different way. It records which nodes in the graph are monitored and sends this information to the worker threads. These then send the appropiate changed node values back to the graph on the master as the model is updated. Each time that the nodes that are monitored in the graphical model changes the information that the worker threads need to send to the master changes. In general the monitoring is done on the master. The exception to this is calculations involving the models deviance. As deviance is a global quantity its calculation is spread over the worker threads and therefore its monitoring also occurs on the worker threads

# Samples Subsystem

# Developer Manual




The Samples subsystem is built on top of the MonitorSamples module. Each component of a name in the BUGS language model can be assocciated with a MonitorSamples.Monitor object. If a component of a BUGS name is observed or is a constant then it is not possible to set a monitor for that component. All the objects of type SamplesMonitors.Monitor are stored in an index in module SamplesIndex and can be accessed by using BUGS langauge names with the appropiate indices. Various display models take the samples stored in an monitor object and produce plots.



# BUGS Logical relations in the BUGS Language

Andrew Thomas
MRC Biostatistics Unit
Cambridge

October 2021

The OpenBUGS software compiles a description of a statistical model in the BUGS language into a graph of objects. Each relation in the statistical model gives rise to a node in the graph of objects. Each distinct type of relation in the statistical model is represented by a node of a distinct class. For stochastic relations there is a fixed set of distributions that can be used in the modelling. For logical relations the situation is more complex. The software can use arbitrary logical expressions build out of a fixed set of basic operators and functions. For each distinct logical expression a new software source code module is writen to implement a class to represent that logical expression in the graph of objects. The software module is then compiled using the Components Pascal compiler and the executable code merged into the running OpenBUGS software using the run time loading linker.

The BUGS language description of a statistical model is parsed into a list of trees. The sub-trees that represent logical relations in the statistical model are first converted into a stack based representation and then into Component Pascal source code. The source code is generated in module BugsCPWrite and the source code is then compiled in module BugsCPCompiler. Usually the generated source code is not displayed. Checking the *Verbose* option in the Info menu will cause each each source code module generated by the OpenBUGS software to be displayed in a seperate window.

One advantage of a stack based representation of an expression is that it is straight forward to use it to derive source code that calculates the derivative of the expession with respect to its arguments. This part of the source code generation is carried out in module BugsCPWrite in procedure WriteEvaluateDiffMethod. Each operator in the stack representation of the logical expression causes a snippet of Component Pascal code to be written. These code snippets are generally very simple with those of binary operators slighly more complex than those of unitary operators. Each binary operators can emit three different code snippets: the general case and two special snippets depending on whether the left or right operands are numerical constants. The only complex code snippet is when an operand that is a logical relation in the statistical

model is pushed onto the stack -- the case of nested logical relations. In this case the nested logical relation will have its own code to calculate derivatives and these values can be passed up the nesting level.

The OpenBUGS software now uses a backward mode scheme to calculate the value of logical nodes in the statistical model. All the logical nodes in the statistical model are held in a global array and sorted according to their nesting level with unnested nodes at the start of the array. To evaluate all the logical nodes in the statistical model this array is then traversed and each logical node evaluated and the value stored in the node. The same scheme is used to calculate derivatives.

The graphs derived from the BUGS language representation of statistical models are generally sparse. The OpenBUGS software uses conditional independence arguments to exploit sparsity in the stochastic parts of the model. There is also a sparsity structure in logical relations.Each logical relation will often depend on just a few stochastic parents and derivatives with respect to other stochastic nodes in the model will be structurally zero. Each logical node has an associated array of stochastic parents for which the derivatives are non zero. Moving up the level of nesting the number of parents can grow. Dealing with this issue leads to the complexity in the code snippet for the operator that pushes a logical node onto the stack. These issues can be seen in the non-linear random effects model called Orange trees in volume II of the OpenBUGS examples. In this model eta[i,] is a function of phi[i,1], phi[i,2] and phi[i,3] where the phi are also logical functions of the stochastic theta[i,].

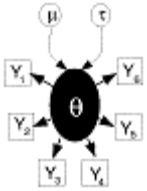
One refinement of the backward mode scheme used to calculate the value of logical nodes is to consider seperately any logical nodes in the statistical model which are only used for prediction and do not affect the calculation of the joint probability distribution. These nodes need only be evaluated once per iteration of the inference algorithm. Examples of such nodes are sigma[k] and sigma.C in the Orange trees example. There is no need to evaluate the derivatives of these prediction nodes.

The workings of the backward mode scheme are easy to visualize when the inference algorithm updates all the stochastic nodes in the statistical model in one block. Local versions of the backward mode scheme can be used when the inference algorithm works on single nodes or when a small blocks of nodes are updated. Each stochastic node is given its own vector of logical nodes that depend on it either directly or via other logical nodes and this vector is sorted by nesting level. Each updater that works on small blocks of nodes contains a vector of logical nodes which is the union of the vectors of dependent logical nodes for each of its components.

The idea of the backward mode scheme for evaluating logical nodes can be used with caching in Metropolis Hastings sampling. First the vector of logical nodes depending on the relevant stochastic node(s) is evaluated and their values cached. The log of the conditional distribution is then calculated. Next a new value of the stochastic node is proposed. The vector of logical nodes

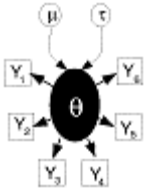
is re-evaluated and the log of the conditional distribution calculated. If the proposed value is rejected then the cache is used to set the vector of logical nodes back to its old values.

The OpenBUGS software also calculates what class of function each logical node is in terms of its stochastic parents. If the software can prove for example that a logical node is a linear function of its parents more efficient sampling algorithms can be used. If a linear relation can be proved then the calculation of derivatives can also be optimized in some cases because they will be constant and so only need to be calculated once. Generalized linear models are implemented in a way that allows fast calculation of derivatives. The structure of the algorithm to classify the functional form of logical nodes is very similar to that for derivatives and uses a backward mode scheme




# Initializing OpenBUGS

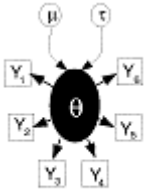
Before the OpenBUGS software runs it needs to be initialized. Each subsystem, Xxx, that needs initializing contains a XxxResources module which contains an exported initialization procedure called Load. Executing this procedure causes the needed resources to be loaded. The code to initialize the unlinked version of the OpenBUGS software is in module Config stored in file Bugs/Mod/Config.odc. This initialization code searches for the code file of the revelevent XxxResources modules and if found uses meta programming to load the module and then call their Load procedures.




# Distributing OpenBUGS

To distribute OpenBUGS open the "BugsPackage" file in the Bugs/Rsrc subdirectory. Type the name of the directory were you want to put the new OpenBUGSdistribution into the dialog box. Check all the check boxes and then click the ok button. All the compiled code, libraries, resource files and documentation needed for a functioning version of OpenBUGS will be copied (but not the BlackBox developement tools). The source code for OpenBUGS will also be copied plus the source code of some additional software development tools. If you do not want to distribute source code then do not check the copy source box. Warning before using this distribution tool close all open windows (except the log window).

The dialog box also contains another button which produces html versions of the source code, examples and documentation.




# Writing OpenBUGS Extensions

Andrew Thomas
MRC Biostatistics Unit
Cambridge

October 2021

## Contents



## Introduction [top]

The OpenBUGS software is a framework that handles the logic of interaction between several class hierarchies. One class hierarchy, the Graph hierarchy, is used to build descriptions of complex Bayesian statistical models. A second, the Updater hierarchy, implements sampling algorithms for variables in the statistical model. New types of logical functions and distributions can be added to the BUGS language by implementing new classes in the Graph hierarchy. New sampling algorithms can be added to Updater hierarchy.

Component Pascal uses the key word ABSTRACT to denote abstract classes, that is classes used as templates for other concrete classes. The same key word is also used to denote abstract methods. The OpenBUGS software has been designed so that only abstract classes are exported, that is made visible outside the module in which they are defined. Concrete classes are completely enclosed in modules and only an object, the factory object, is exported to allow the creation of instances of the concrete class. This means that concrete classes can only be derived from abstract classes. OpenBUGS also follows the convention that method are either abstract or final (can not be modified). The only exception to this is some methods are empty (denoted by key word EMPTY) that is their implementation in concrete classes is optional. Writing a new class in OpenBUGS involves extending an abstract class such that all the abstract methods are implemented and implementing any empty methods that are relevent. The Component Pascal compiler checks that a concrete class has no abstract methods and the

browser tool (Extension Interface option) can be used to check which methods of a class need implementing.

## The Graph hierarchy [top]

OpenBUGS describes statistical models in terms of a graph. Each node in the graph is represented by an object belonging to one of the classes in the Graph hierarchy. Each of these classes is descendant from the class GraphNodes.Node (that is the class Node in module GraphNodes). The class GraphNodes.Node is first specialized to GraphLogial.Node to represent logical function and GraphStochastic.Node to represent distributions in the BUGS language. The class GraphLogical.Node is futher specialized to GraphScalar.Node to represent scalar valued logical functions eg the logit link and to GraphVector.Node to represent vector (or matrix) valued logical funtions eg matrix inversion. The class GraphStohcastic.Node is specialized to GraphUnivariate.Node to represent univariate distributions eg the beta distribution and to GraphMultivariate.Node to represent multivariate distributions. The class GraphMultivariate.Node is furher specialized to GraphConjugateMV to represent multivariate distributions with conjugate properties eg the multivariate normal and GraphChain.Node to represent chain graph distributions eg the spatial CAR model. These node classes are abstract, that is they have unimplemented methods, objects of these type can not be created however they can be futher specialised to concrete classes, where all the methods are implemented, which can be used to create nodes in the graph describing the statistical model..

To ease the writing of new logical nodes some additional abstract classes have been developed. These class implement as many of the methods of the logical node classes as have reasonable default behavoir and leave the developer with the minimum number of methods to implement. These semi implemented classes are in modules GraphScalarT and GraphVectorT. Using these classes does not allow as full an exploitation of functionality of the OpenBUGS software as extending the classes in GraphScalar and GraphVector would but is much less work and adequate for some applications.Their use is illustrated in modules GraphScalartemp1 (implementing the calculation of the harmonic mean) and GraphVectortemp1 (implementing the calculation of an integer power of a square matrix). Modules GraphScalartemp1 and GraphVectortemp1 can be used as templates for developing other logical nodes. Simillarly to ease the writing of new univariate stochastic nodes some additional abstract classes have been developed. These classes are in module GraphUnivariateT and their use is illustrated in module GraphUnivariatetemp1 (implementing the normal distribution but with no knowedge of conjugacy).

Once a new node type has been implemented OpenBUGS must be told what name to use for this new node type in the BUGS language and the module where it is implemented. This information is stored in the configuration module BugsExternal. This module contains pairs of string: the first one being the name of the node type and the second the name of the module which implements the new node type followed by a period followed by the "install procedure" usually called Install.

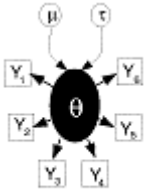
# The Updater hierarchy [top]

There is also a hierarchy of updater classes to implement the updating or sampling algorithms. The sampling algorithm for each node or block of nodes in the graphical model is contained in an object of the appropiate class. These updater classes use the graphical model to efficiently calculate the required conditional probabilities. All the updater objects for the graphical model are stored in an array sorted according to the topological order of the assocociated nodes in the graphical model (nodes in the graphical model purely used for prediction have by convention their depth in the graph multiplied by minus one). Using this ordering of updaters MCMC can be used for nodes with observed offspring and forward sampling for the nodes with no observed offspring. One complete update of the model can be carried out by traversing the updater array and calling the appropiate sampling method.

All updater classes are derived from class UpdaterUpdaters.Updater. This class is first specialised into univariate and multivariate (block) updaters. The univariate class is specialized to the continous updaters. The multivariate class is specialized to conjugate multivariate updaters. Writing a new updater class involves specializing one of the folllowing classes: UpdaterUnivariate.Updater, UpdaterContinuous.Updater, UpdaterMultivariate.Updater and UpdaterMultivariate.Conjugate. The methods that must be implemented when these classes are made concrete can be found by selecting the modules name followed by the class name (eg "UpdaterMultivariate.Conjugate") and then picking "Extension Interface" from the Info menu. Concrete updaters are made by factory objects. Factory objects have a New method which takes one argument the node in the graphical model for which an updater is required. This New method can return a new updater object if this is a valid and desirable thing to do otherwise it returns a nil pointer.

Two template modules UpdaterUnivariateT and UpdaterMultivariateT have been writen to ease the writing of new univariate and block updaters.

Once a new updater has been written OpenBUGS must be told in which module it is implemented. This information is stored in the configuration module UpdaterExternal, which contains a list of install procedures to install updater algorithms. A list of installed updater algorithms is displayed in the Updater options tool of the Model menu




# Sampling Algorithms

Andrew Thomas
MRC Biostatistics Unit
Cambridge

October 2021

## Inroduction

OpenBUGS uses Markov Chain Monte Carlo (MCMC) simulation [Gilks] to make inference for complex Bayesian statistical models. Many different sampling algorithms are used within the MCMC simulation depending on the structure of the statistical model. At the highest level of abstraction the sampling algorithms are either Metropolis-Hastings algorithms [Hastings] or Slice Sampling algorithms [Neal]. Metropolis-Hastings algorithms can be further classified by their choice of proposal distribution. Gibbs Sampling is just a special case of Metropolis-Hastings where the proposal distribution is the same as the conditional distribution and hence the acceptace ratio is always unity. The so-called Hybrid algorithms [Duane et al] (better called Hamiltonian algorithms) are an attempt to build arbitary good poposal distributions for general conditional distributions. They depend on theory from classical (Hamiltonian) dynamics and require knowledge of the derivative of the conditional distribution. A common choice of proposal distribution is a random step round the current point. In this case the Metropolis-Hastings acceptance ratio reduces to the value conditional distribution at the new point over the value at the old point.

Most proposal distributions used in Metropolis-Hastings algorithms depend on one or more parameters. Choosing reasonable values for these parameters has a crucial effect on the performance of the algorithm. Often these parameter values must be learned which breaks the Markov nature of the Metropolis-Hastings algorithm and brings into question the convergence of the simulation. If the parameters change less and less as the learing process progresses (diminishing adaptation) then the simulation can still be convergent [Roberts & Rosenthal]. Otherwise the simulation must be broken into two phases: the first where the parameters are learnt and the second where the simulation is allowed to converge and then inference made.

An interesting extension to Metropolis-Hasting is called delayed rejection [Tierney & Mira]. If the candidate point from the proposal distribution is rejected a new candidate point is sampled (maybe from a different propsal distrubtion) and accepted with a modified acceptance ratio.

Typically delayed rejection first tries a proposal distribution which makes big steps and if this fails next tries a smaller step.

## Choice of Sampling Algorithm

When OpenBUGS starts up a module called UpdaterExternal in Updater/Mod which contains information about MCMC sampling algorithms is loaded and processed. The modules implementing the sampling methods listed in this Extermal module are then loaded dynamically. Factory objects to create updater objects which use these methods are stored in an array (in the same order as they occur in the UpdaterExternal module). Users of the Windows interface of OpenBUGS can view this array of sampling algorithms by opening the Updater options dialog under the Model menu. The first factory object in this array is used to create updaters for all the stochastic nodes in the model that need updating and for which the associated sampling method is valid. If there are any nodes which have not had updater objects created by this first factory object then the second factory is tried and so on. In this way only a single updater object is created for each node that needs sampling. If none of the factory objects are able to create an updater object for a particular node in the model that needs sampling an error is reported. It is the task of the factory object to decide if its associated sampling method is valid for a particular node in the model. However the factory object can choose not to create an update for a particular node even if such an updater would be valid.

The factory objects used to create updater objects are extensions of the class UpdaterUpdaters.Factory. Updater objects are created by calling the factory's New method. The New method takes one input parameter of type GraphSochastic.Node. Typicaly this input parameter is the node in the model for which the factory object should try and create an updater object. If a valid updater object can not be created or if the factory does not wish to create an updater for the particular stochastic node given as an input parameter then it should return NIL. For block updater algorithms the factory's New method calls a procedure which calculates a block of stochastic nodes assocciated with the stochastic node parameter. The block updater will then genertate samples for this block of nodes. Usualy the block updater will include its stochastic node in parameter in the block of nodes it updates.

There are three common situation where it can be useful to block update a group of nodes. Firstly when a group of nodes have a common likelihood and secondly when a group of nodes share a common prior. Fixed effects in a generalized linear model are typical of the first situation, while random effects are typical of the second. Even if a group of nodes has the same likelhood it might not be a good idea to block update them if the functional form of the likelihood is very different for each element of the block. We have this situation if the likelihood distribution has two parameters each depending on seperate blocks of nodes, it is best not to merge the two blocks into one larger block updater. A final case when a block of nodes might best be updated jointly is when they have a multivariate pior, for example a spatialy prior or a dirichlet prior. Some common algorithms for finding blocks of nodes that can be sampled jointly are in module UpdaterMultivariate.

UpdaterMultivariate.FixedEffects calculates the list of co-parents of a given node and then selects certain members of this list to block together with the node. The selection can depend on the classification of the form of conditional distribution, the topological depth in the graph of the co-parent or whether the co-parent has bounds. This procedure can be used to form different blocks of nodes suited to particular block updating algorithms.

There is a slight problem with using the classification of univariate conditional distributions to choose block updaters. Consider these two snippets of BUGS language code:

```
y ~ dnorm(mu, tau)
mu <- beta[1] + beta[2] * x
beta[1:2] ~ dmnorm(mean[], T[,])
```

and

```
y ~ dnorm(mu, tau)
mu <- beta[1] * beta[2]
beta[1:2] ~ dmnorm(mean[], T[,])
```

In each case beta[1] and beta[2] have univariate normal conditionals but only in the first case does beta[1:2] have a bivariate normal conditional. Similar situations can occur when choosing block updaters for GLM. The example is rather trival but more realistic setups include covariate measurement error and factor models.

Factory objects associated with sampling algorithms listed at the start of the External module can impose strict conditions on the node for which they are designed to create an updater object. In general, algorithms at the start of the methods file do block updating of nodes. Users can develop special-purpose sampling algorithms and place them at the start of the External module. Algorithms towards the end of the External module file tend to be general - purpose (and somewhat less efficient) than earlier algorithms.

Different factory objects can create the same type of updater object. This feature allows a fine tunning of the choice of update algorithm. For example there might be particular curcumstances where a general purpose sampling algorithm is known to be very efficient. This sampling algorithm could be given two associated factory objects the first checking for the special situation of efficient sampling. An example of this possibility is the Slice Sampler. If it can be proved that the distribution to be sampled is unimodal the algorithm can be made more efficient.

An algorithm might be commented out from the External file because it is less efficient than a competing algorithm, because it performs badly in some test situations, because it causes less efficient algorithms to be chosen for other nodes or because it has not been tested enough. Users are encouraged to try some of the commented out algorithms and to report their good and bad experiences.

The module UpdaterNaivemet is given as an example of a general purpose updater that can be used for sampling real valued parameter. The algorithm used is a random walk metropolis algorithm with a fixed standard normal proposal distributionl. It is interesting to observe how badly the algorithm performs when the conditional distribution to be sampled has a scale very different from the standard normal proposal.

## Description of Sampling Algorithms

Here is an alphabetical list of install procedures for updaters that can be used followed by a brief note on how the algorithm works and what type of situation it is appropiate to.

*UpdaterAMblock.InstallGLM*

Current point Metropolis type updater with delayed rejection and continously adapted multivariate normal proposal distribution. For a fixed effect block of nodes all of which have a logistic or log-linear conditional distribution. Non-Markov algorithm. Not currently used.

*UpdaterAMblock.InstallNL*

Current point Metropolis type updater with delayed rejection and continously adapted multivariate normal proposal distribution. For a fixed effect block of nodes the first element of which has a generic conditional distribution. Non-Markov algorithm.

*UpdaterBeta.Install*

Gibbs type of updater for beta conditional distribution.

*UpdaterCatagorical.Install*

Gibbs type of updater for univariate node which takes discrete values with a upper bound. Works by enumeration. Slow if many categories.

*UpdaterDescreteSlice.Install*

Slice type sampler for discrete variable. Competes with UpdaterCatagorical if discrete variable has upper bound and with UpdaterMetbinomial if no upper bound.

*UpdaterDirichlet.Install*

Gibbs type of updater for a conditional distribution that is dirichlet.

*UpdaterForward.InstallUV*

Gibbs type of updater for univariate node which does not have any likelihood.

*UpdaterGamma.Install*

Gibbs type of updater for gamma conditional distribution.

*UpdaterGLM.InstallLogit UpdaterGLM.InstallLog*

Metropolis Hastings type updater for a block of nodes which have either a logistic or log-linear conditional distribution with unbounded support.

*UpdaterMultinomial.Install*

Metropolis type updater with multinomial prior.

*UpdaterGLM.InstallNorma*l

Gibbs type of updater for multivariate conditional distribution where the prior is a set of univariate normal nodes. Not currently used.

*UpdaterGMRF.InstallGeneral*

Current point block Metropolis algorithm for nodes with a Gaussian Markov Random Field prior. Can update large blocks. Also able to implement contsraints. Uses sparse matrix algebra [Rue].

*UpdaterGriddy.Install*

Independent Metropolis Hastings type updater for generic univariate distribution. Distribution must have bounded support. Builds a trapezoidal approximation to the conditional to use as the proposal. Slow algorithm. Not currently used, slice sampling is preferred but see note below about multimodality.

*UpdaterMetbinomial.Install*

Current point Metropolis type updater with binomial proposal for univariate node which takes discrete values with no upper bound.

*UpdaterMetnormal.InstallDelayed*

Current point Metropolis type updater for generic univariate distribution with unbounded support. Uses a normal proposal distribution that is adapted during a tuning phase. Uses delayed rejection if first proposal is rejected.

*UpdaterMetnormal.InstallMH*

Current point Metropolis type updater for generic univariate distribution with unbounded support. Uses a normal proposal distribution that is adapted during a tuning phase.

*UpdaterMultinomial.Install*

Independence Metropolis type of updater for a conditional distribution that has a multinomial prior.

*UpdaterMVNLinear.Install*

As for UpdaterMVNormal.Install but the likelihood can have terms that are multivariate normal of a different dimension to the prior and the link function does not have to be the identity but can have a linear form.

*UpdaterMVNormal.Install*

Gibbs type of updater for multivariate normal conditional distribution where each term in the likelihood is either normal, log normal or multivariate normal of same dimension as the prior with a unit link function.

*UpdaterNormal.Install*

Gibbs type of updater for univariate normal conditional distribution.

*UpdaterPareto.Install*

Gibbs type of updater for pareto conditional distribution.

*UpdaterPoisson.Install*

Gibbs type of updater for poisson prior with single binomial likelihood term having order equal to the prior.

*UpdaterRejection.InstallLog UpdaterRejection.InstallLogit*

Gibbs type of updater for univariate node with log-linear or logistic distribution and unbounded support.

*UpdaterSCAAR.InstallMH*

Single component adaptive Metropolis algorithm that tunes the acceptance rate [Roberts & Rosenthal] .

*UpdaterSDScale.Install*

Single component adaptive Metropolis algorithm that tunes the acceptance rate in a way that depends on the scale of the variable [Roberts].

*UpdaterSlice.Install*

Slice type of updater for generic univariate distribution. Uses stepping out search procedure to find the slice. Step length of search procedure adapted during tuning phase. Can miss modes for multimodal conditional distribution if the support is not bounded.

*UpdaterWishart.Install*

Gibbs type of updater for Wishart conditional distributions.

## References

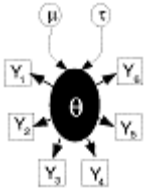
[Gilks] THE MCMC BOOK

[Hastings] Hastings, W.K. (1970). "Monte Carlo Sampling Methods Using Markov Chains and Their Applications". Biometrika 57 (1) pp 97–109

[Tierney & Mira] Tierney L. and Mira A. (1999) "Some adaptive Monte Carlo methods for Bayesian Inference" Statistics in Medicine vol 18 pp 2507 - 2515

[Neal] Neal R. "Slice Sampling" (2003) Annals of Statistics vol 31 pp 705 - 767

[Roberts & Rosenthal] Roberts G. O. and Rosenthal J. S. "Examples of Adaptive MCMC" (2009) Journal of Computational and Graphical Statistic




# Blue Diamonds

Andrew Thomas
MRC Biostatistics Unit
Cambridge

October 2021

I like blue diamonds, maybe you also like blue diamonds? Your version of BUGS can now have blue diamonds. A modified version of the module DevDebug has been written. The Node inspector tool will now gives you blue diamonds so that you can explore Graph and Updater objects. The blue diamonds allow complex graphical model build by BUGS to be explored in detail. Where possible hexidecimal numbers that represent nodes in dynamic data structures are replaced by their name in the BUGS language model. All addresses that represent nodes in the graphical model are displayed between angled brakets (<>). Adding the DevDebug module to BUGS will also give more informative trap messages when things go wrong. However if module BugsTraphandler is present traps will be replaced by simple error messages.

As a simple example of blue diamonds in action compile and initialize the Beetles model. Then select the "Node inspector" tool from the Info menu and type r in the dialog box and click on the type button. You should see a window containing

| | Type | |
|---|---|---|
| r[1] | dbin | ◆ |
| r[2] | dbin | ◆ |
| r[3] | dbin | ◆ |
| r[4] | dbin | ◆ |
| r[5] | dbin | ◆ |
| r[6] | dbin | ◆ |
| r[7] | dbin | ◆ |
| r[8] | dbin | ◆ |

Next click on the blue diamond opposite r[1] to get information about node r[1] in the Beetles model, you will get another window containing

dbin^

| | | |
|---|---|---|
| <r[1]>GraphBinomial.Node^ | ➡ | |
| .value | REAL | 6.0 |
| .props | SET | {1..3, 18, 24} |
| .label | INTEGER | 0 |

| | | |
|---|---|---|
| .graph | GraphStochastic.Graph | NIL |
| .censor | GraphUnivariate.Limits | [02B4C710H] ◆ |
| .truncator | GraphUnivariate.Limits | [02B4C710H] ◆ |
| .p | GraphNodes.Node | <p[1]> ◆ |
| .omega | GraphStochastic.Node | NIL |
| .n | GraphNodes.Node | <n[1]> ◆ ⬅ |

The r[1] node has several field amongst them p[1] and n[1]. More information about p[1] can be gained by clicking on its blue diamond. You will get another window containing

p^

<p[1]>GraphLogit.Node^ ➡

| | | | |
|---|---|---|---|
| .value | REAL | 0.5 | |
| .props | SET | {3, 4, 15} | |
| .label | INTEGER | 0 | |
| .work | POINTER | [029E01D0H] ◆ | |
| .parents | GraphNodes.Vector | | [029E01B0H] ◆ |
| .nesting | INTEGER | 2 | |
| .predictor | GraphNodes.Node | | [02A15240H] ◆ ⬅ |

So the p[1] is of type GraphLogit.Node, the logistic node. It has several fields including the predictor field which can be explored by clicking on its blue diamond. Doing this opens a window containg

p^.predictor^ ◆

[02A15240H]DynamicNode0.Node^ ➡

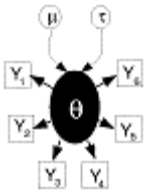
| | | | |
|---|---|---|---|
| .value | REAL | 0.0 | |
| .props | SET | {10, 15} | |
| .label | INTEGER | 0 | |
| .work | POINTER | [029DA780H] ◆ | |
| .parents | GraphNodes.Vector | | [029DA340H] ◆ |
| .nesting | INTEGER | 1 | |
| .c0 | REAL | -0.1027250000000002 | |
| .s0 | GraphStochastic.Node | | <alpha.star> ◆ |
| .s1 | GraphStochastic.Node | | <beta> ◆ ⬅ |

By clicking on the first blue diamond in the window you can go back one level, do this then click on the blue diamond next to the logical field

We have explored in some detail the part of the graphical model that corresponds to the two lines on BUGS language code

```
r[i] ~ dbin(p[i],n[i])
logit(p[i]) <- alpha.star + beta * (x[i] - mean(x[]))
```

Happy clicking




# MultiBUGS software

Andrew Thomas
MRC Biostatistics Unit
Cambridge

October 2021

## Contents



## Introduction [top]

The parallel implementation of the BUGS software the MultiBUGS software consistes of two coupled programs: the master program and the worker program. One copy of the master program is run which interacts with multiple copies of the worker program. The master program handles interaction with the user, sends requests to the worker programs and receives results from the worker programs. The message passing library MPI is used for communications between the master program and the worker programs and for communications between the worker programs. MPI is a large library but MultiBUGS only uses a small fraction of the procedures in the library. A Component Pascal interface module MpiMPI has been writen so that the MPI library can be used in MultiBUGS. This interface module is low level and not type safe. Two safe modules have been based on the MpiMPI interface: MpiMaster and MpiWorker. MpiMaster handles communication between the master program and the worker programs. MpiWorker handles both communication between the worker programs and the master program and communications between worker programs. MpiMaster is used by the master program while MpiWorker is used by the worker program.

The MultiBUGS software has been designed as an extension to the OpenBUGS software and

makes use of many of the modules of the OpenBUGS software plus a few additional modules. These additional modules fall into two groups: those that constitute the master program (some modules in subsystem Bugs) and those that are used in the worker programs (modules in subsystem Parallel). The OpenBUGS software uses dynamic loading and linking of modules to construct, at runtime, software for a particular statistical model. This dynamic approach can be extending to loading additional modules to add the functionality of the master program to the running OpenBUGS software. A different approach is taken to the worker programs. The particular statistical model is analysed and the modules needed for its MCMC simulation are listed. This list of modules is combined with a few additional models (from subsystem Parallel) and statically linked to form the worker program executable.

## Master Progam [top]

The master program consistes of a small number of additional modules in the Bugs subsystem: BugsPartition, BugsParallel, BugsLeafs, BugsComponents, BugsMaster and BugsInfoDist. Executing the Install procedure in module BugsMaster causes the loading of these additional modules and sets a global hook in module BugsInterface. The OpenBUGS software will then behave as the MultiBUGS software. Setting the global hook back to NIL will return the behavoir of the software to OpenBUGS mode. If module BugsMaster is never loaded then the OpenBUGS software never needs access to the MPI library.

If the BUGS software is run under MPI then it is possible to use the features of MultiBUGS. The software looks for an enviroment variable to indicate if it is running under MPI. If the enviroment variable is set then dialog boxes which offer the possibility of using parallel MCMC algorithms are selected. If the software is not run under MPI it is not possible to use parallel processing, indeed the modules that use MPI are never loaded. To run the BUGS software under MPI the short cut to the OpenBUGS.exe file must be changed so that the target looks something like *"C:\Program Files\Microsoft MPI\Bin\mpiexec.exe" -n 1 "C:\OpenBUGS\OpenBUGS.exe"* where mpiexec.exe is the MPI executable and -n 1 tells to run one copy of OpenBUGS is parallel.

Module BugsPartition analyses analyses the current statistical model and decides which parameters can be sampled in parallel and which log likelihood type computations can be split between multiple worker programs. Module BugsParallel has several functions. Firstly it serializes the data structures that describe the shards of the graphical model. The worker programs read in these serialized data structures to set up their MCMC simulation algorithms. Secondly it tracks which parameters and logical nodes (derived parameters) are updated on which worker program. Module BugsComponents watches the serialization of the statistical model and records which modules are required to rebuild this model on the worker programs. BugsLeafs is a helper module which enables the worker program to be statically linked. It generates two source code modules GraphLoader0 and UpdaterLoader0 which are compiled on the fly and then linked into the worker exectutable.

BugsMaster uses the Component Pascal linker to link the modules of the worker program to create a small executable called BugsWorker.exe. The MPI library is then used to spawn a number of copies of BugsWorker and to establish communications between the master program

and these workers.

## Worker Progam [top]

The worker program does not interact directly with the user. It is able to read and write files and use MPI. On start up the worker program reads a file which contains the serialized data structures needed for MCMC simulation. It then builds the graph of objects used to do MCMC simulation. Once the worker has rebuilt the graph and created the required sampling objects it waits for instructions from the master. For details of how the model is serialized see the next chapter -- ***MultiBUGS binary file format***.. The worker program consists of a loop that waits for instuctions from the master. The instructions are exectuted and results and status information returned to the master.

The worker partitions the global MPI communicator COMM_WORLD to which all the worker programs belong into seperate communicators for each chain of the MCMC simulation. These smaller communicators are used by the parallel MCMC algorithms for tasks such as summing contributions to log likelihoods.

Some modules in the Graph and Updater subsystems can be run both as part of OpenBUGS and as part of the worker programs in MultiBUGS. When they are used in the worker programs they might need to use MPI for tasks such as summing contributions to log likelihoods. These modules therefore import MpiWorker but will only cause the MPI library to be loaded when they are used as part of the worker programs.

## Master worker communication [top]

The statistical model is split between a number of worker programs. This means that in general not all nodes of the statistical model will be on all worker programs. We require the master component of the MultiBUGS software to track which nodes in the statistical model are updated on which worker. Module BugsParallel maintains a two dimensional table of stochastic nodes where the nodes in each column are updated on the worker with MPI rank corresponding to the column label. This table can be used to ask the appropiate worker to send updated values of stochastic nodes back to the master (or a subset of such nodes that are of interest to the user). For some models each worker also needs to know the worker on which each stochastic node is updated. In this case each worker also contains this two dimensional table of stochastic nodes.

The values of logical nodes are also the interest to the user. The user may wish for example to monitor the value of a logical node in the model in order to make inference. Often the value of a logical node gets evaluated at the same time as the value of the stochastic node it depends on is sampled. This means that the values of logical nodes can be fetched from the appropriate worker. The module BugsParallel contains a two dimensional table of logical nodes for this purpose. A slight complication is caused by logical nodes that are in the statistical model soley for prediction purposes and do not affect the estimation of the model's stochastic parameters. These prediction nodes are only evaluated on the master program in MultiBUGS. However their

stochastic or logical parents are updated on the worker programs. So when a prediction type node is monitored the worker programs have to send values of the required stochastic and logical nodes back to the master.

Communication between the master and worker programs occurs in just two modules: BugsMaster and DevianceInterface. These are the only two modules that import the MpiMaster module and hence use the MPI library. The Deviance subsystem works with information criterion such as DIC and WAIC. As deviance is a global model property its calculation is distributed over the worker programs and the calculation of DIC and WAIC involves some complex communication between the master and worker programs.

## Tools for visualization [top]

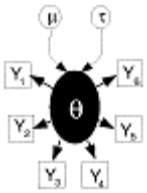
The MultiBUGS software contains a number of tools to visualize how the graph of the statistical model is dissected into shards in order to be distribute to a number of worker programs. There are four different tools. These tools are found at the end of the Info menu in a pull down menu.

The first tool ***Show Distribution*** shows how sampling objects are distributed over cores. Each column of the display shows which parameters are sampled on which worker. If the sampling of a parameter is distributed over the workers then the name of the parameter will occur in a row across all columns. The last column of the display is either an integer or a bar. The integer is used when there is a group of sampling algorithms that can be executed in parallel and gives an offset into the block of updaters. A star after the integer denotes that that updater is the last one in the block and some housekeepoing might be required.

The second tool ***Show Deviance Dist*** shows how the deviance is partitioned between workers. Besides data this display can also include censored observations. Censored observations are postfixed with the string ***.cens***. Censored observations have a hybrid character: they are both observations and require sampling. The MultiBUGS distribution algorithms ensure that that the sampling object for censored observations occurs on the same worker as the contribution to deviance.

The third tool ***Distributed graph...*** opens a dialog box asking the user which worker they want to visualize. The display is a more detailed version of that produced by ***Show Distribution***. The main addition is blue diamonds associated with each entry. Clicking on these blue diamonds allows the user to explore in detail the data structure representing the shard that is on that worker. This tool uses the same algorithms that are evoked when the user distributes a model by clicking the ***distribute*** option in the ***Specification tool***. It is therefore useful for debuging the disection of a graphical model into shards.

The fourth and final tool ***Distribution info*** displays information about how the model has been distributed once the ***distribute*** option in the ***Specification tool*** has been clicked. As well as memory usage statistics it list the software modules that have been linked to create the worker program




# MultiBUGS binary file format

Andrew Thomas
MRC Biostatistics Unit
Cambridge

August 2021

## Contents



## Introduction [top]

The OpenBUGS software can write a detailed representation of the graphical model to a binary formatted file. In MultiBUGS this information is written out in such a way that that parts of the file can be read in by each worker thread to implement a parallel version of the MCMC simulation. This binary file consists of a number of sections starting with some header information and then blocks of information for each shard that MultiBUGS has dissected the graphical model into plus the MCMC updater objects that act on that shard. Finally the current values of stochastic nodes are appended to the end of the file. This binary file is read by each worker thread which extracts the information it requires to set up the data structures representing its graph shard and do MCMC simulation.

The {} notation is used for information that occurs one or more times.

## Header [top]

The header consists of the following information:

*numberOfChains*: INTEGER

The number of chains in the MCMC simulation.

*workersPerChain*: INTEGER
The number of workers per MCMC chain. This is the number of shards the graphical model is dissected into.

*state of random number generators*
See BugsRandnum.ExternalizeRNGenerators for details.

*debug*: BOOLEAN
If true writes debug information into the binary file. Usually set to false.

*devianceExists*: BOOLEAN
True if the model has a deviance.

*seperable*: BOOLEAN
True if the model shards seperable.

{*pos*: LONGINT}
Start position in the file of each graph shard's information. There are *workersPerChain* pos.

*posCurrentValues*: LONGINT
Start position in the file of current values for all the graph shards and MCMC chains.

## Graph shards [top]

Blocks of graph shards information

*numStochasticPointers*: INTEGER
The number of stochastic nodes sampled on the shard.

*numLogicalPointers*: INTEGER
The number of logical nodes in the shard.

*numDataPointers*: INTEGER
The number of data nodes in the shard.

*numDeviancePointers*: INTEGER
The number of nodes in the shard that contribute to the deviance.

*numUpdaters*: Integer
The number of MCMC updater objects for the shard.

*sizeDevianceWorkSpace*: INTEGER

The size of work space needed for deviance calculations for the shard.

{*globalId*: INTEGER}
Integer array of length *numUpdaters* giving information about when the shards need to communicate with each other.

*numNodes*: INTEGER
The number of nodes in the graph shard.

*nodeTypeInfoPos*: LONGINT
The position in the file to jump to to read node type information. This is just after all the information about the nodes in the shard has been writen.

*nodeLabel*: INTEGER; *typeLabel*: INTEGER
A pointer to a univariate stochastic dummy node and its type.

*nodeLabel*: INTEGER; *typeLabel*: INTEGER
A pointer to a multivariate stochastic dummy node and its type.

{*nodeLabel*: INTEGER *; typeLabel*: INTEGER}
A rectangular array of pairs of stochastic pointers and their type in column order. A row of this table can consist of the same node repeated if this node represents a fixed effect in the model. The rectangular array has dimensions *numStochasticPointers* by *workersPerChain*.

{*nodeLabel*: INTEGER; *typeLabel*: INTEGER}
One dimensional array of pairs of pointers to logical nodes and their types

{*nodeLabel*: INTEGER; *typeLabel*: INTEGER}
One dimensional array of pairs of pointers to data nodes and their types.

{*nodeLabel*: INTEGER}
One dimensional array of pointers to deviance nodes. Note these pointers always occur in either the data array or stochastic array, depending on whether the entry is observed or censored, and so there is no need for type information.

{*addresses*: INTEGER}
One dimensional array of debug information. Only writen if *debug* is true.

Internal fields (including pointers) of all the nodes in the graph shard. This information is writen in the same order as the node pointer information has been writen. Note some node pointers can be writen out more than once but the internal fields are only writen once per node.

*numNodeTypes*: INTEGER
The number of distinct types of node in the graph shard. This number is writen at position *nodeTypeInfoPos* in the file.

{*installNodeProc*: ARRAY OF CHAR}
Strings that when called as procedures install the appropiate factory objects to create the required node objects. There are *numNodeTypes* of *installNodeProc* strings.

*updaterTypeInfoPos*: LONGINT
The position in the file to jump to in order to read updater type information. This is just after all the information about the updaters in the shard has been writen.

{*typeLabel*: INTEGER {*priorLabel*: INTEGER}}
Information used to create updater objects. The *typeLabel* tells which type of updater to create and the single or array of *priorLabel* tells which nodes the updater object acts on.

{f*ilePositions*: LONGINT}
Position of the volatile internal information of the shards updaters by MCMC chain.

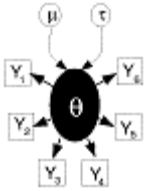
Internal fields of all the updaters in graph shard for each MCMC chain. Because BUGS uses some non markov MCMC sampler the updater objects can contain mutable state which needs to be writen out.

*numUpdaterTypes*: INTEGER
The number of distinct types of updater in the graph shard.

{*installUpdaterProc*: ARRAY OF CHAR}
Strings that when called as procedures install the appropiate factory objects to create the required updater objects. There are *numUpdaterTypes* of *installUpdaterProc* strings.

## Current values [top]

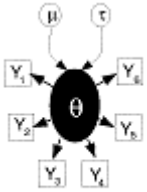
Blocks of current values per graph shard and per chain. A real value is written for each entry in the two dimensional array of stochastic nodes. The values are repeated over chains. In total *numStochasticPointers* by *workersPerChain* by *numChains* real values are writen.




# Distributing OpenBUGS

To distribute OpenBUGS open the "BugsPackage" file in the Bugs/Rsrc subdirectory. Type the name of the directory were you want to put the new OpenBUGSdistribution into the dialog box. Check all the check boxes and then click the ok button. All the compiled code, libraries, resource files and documentation needed for a functioning version of OpenBUGS will be copied (but not the BlackBox developement tools). The source code for OpenBUGS will also be copied plus the source code of some additional software development tools. If you do not want to distribute source code then do not check the copy source box. Warning before using this distribution tool close all open windows (except the log window).

The dialog box also contains another button which produces html versions of the source code, examples and documentation.




# The grammar of the BUGS PPL

Andrew Thomas
MRC Biostatistics Unit
Cambridge

November 2021

I use braces {} to denote one or more occurences of a construct and square brackets [] to denote that the construct is optional. Terminals are in block capitals. The rules from <program> to <range> concern the syntax of the BUGS language. The rules from <name> to <real> concern the lexical structure of the BUGS language. The grammar of the BUGS language in BNF is:

<program> ::= MODEL [ <name> ] BRACEL { <statement> } BRACER

<statement> ::= <compound_statement> | <simple_statement>

<compound_statement> ::= <for_statement> | <if_statement>

<for_statement> ::= FOR BRACKETL <name> IN <index> COLON <index> BRACKETR
BRACEL { <statement> } BRACER

<if_statement> ::= IF BRACKETL <scalar> BRACKETR BRACEL { <statement> } BRACER

<simple_statement> ::= <stochastic_statement> | <logical_statement>

<stochastic_statement> ::= <uni_statement> | <multi_statement>

<logical_statement> ::= <scalar_statement> | <tensor_statement>

<uni_statement> ::= <scalar> DISTRIBUTED <distribution> [ <censored> ] [ <truncated> ]

<multi_statement> ::= <tensor> DISTRIBUTED <distribution>

<scalar_statement> ::= <lhs> BECOMES <expression>

<tensor_statement> ::= <tensor> BECOMES <tensor_function>

<distribution> ::= <name> BRACKETL <arguement_list> BRACKETR

<arguement_list> ::= [ <arguement> { COMMA <arguement> } ]

<arguement> ::= <scalar> | <tensor> | <number>

<censored> ::= CENSOR BRACKETL [ <bound> ] COMMA [ <bound> ] BRACKETR

<truncated> ::= TRUNCATE BRACKETL [ <bound> ] COMMA [ <bound> ] BRACKETR

<bound> ::= <scalar> | <number>

<lhs> ::= <scalar> | <link_function> BRACKETL <scalar> BRACKETR

<expression> ::= <term> | <expression> ( PLUS | MINUS ) <term>

<term> ::= <factor> | <term> ( MULT | DIV ) <factor>

<factor> ::= [ MINUS ] (BRACKETL <expression> BRACKETR | <number> | <scalar> |
<unary_internal_function> | <binary_internal_function> | <external_function> )

<link_function> ::= CLOGLOG | LOG | LOGIT | PROBIT

<external_function> ::= <name> BRACKETL <arguement_list> BRACKETR

<tensor_function> ::= <name> BRACKETL <argument_list> BRACKETR

<unary_internal_function> ::= <unary_function_name> BRACKETL <expression> BRACKETR

<binary_internal_function> ::= <binary_function_name>
BRACKETL <expression> COMMA <expression> BRACKETR

<unary_function_name> ::= ABS | ARCCOS| ARCCOSH | ARCSIN | ARCSINH | ARCTAN |
ARCTANH | CLOGLOG | COS | COSH| EXP | ICLOGLOG | ILOGIT | LOG | LOGFACT |
LOGGAM | LOGIT | PHI | ROUND | SIN | SINH | SOFTPLUS | SQRT | STEP | TAN | TANH |
TRUNC

<binary_function_name> ::= EQUALS | MAX | MIN | POWER

<scalar> ::= <name> [ SQUAREL <index> { COMMA <index> } SQUARER ]

<tensor> ::= <name> SQUAREL <range> [ { COMMA <range> } ] SQUARER

<index> ::= <integer> | <scalar>

<range> ::= ( <index> [ COLON <index> ] ) | SPACE

<name> ::= <letter> [ { <letter> | <digit> | UNDERSCORE | PERIOD } ]

<number> ::= [ <sign > ] ( <integer> | < real> )

<letter> ::= "a" .. "z" | "A".."Z " | "α".."ω.." | "A".."Ω"

<digit> ::= "0".."9"

<sign> ::= PLUS | MINUS

<integer> ::= <digit> [ {digit} ]

<real> ::= [ <integer> ] PERIOD <integer> [ EXPONENT <sign> <integer> ]

The terminal symbols are:

BECOMES ::= "<-" BRACEL ::= "{" BRACER ::= "}" BRACKETL ::= "("

BRACKETR ::= ")" CENSOR ::= "C" COLON ::= ":" COMMA ::= ","

DIV ::= "/" DISTRIBUTED ::= "~" EXPONENT ::= "E" MINUS ::= "-"

MULT ::= "*" PERIOD := "." PLUS ::= "+" SPACE ::= " " SQUAREL ::= "["

SQUARER ::= "]" TRUNCATE ::= "T" UNDERSCORE ::= "_"

ABS ::= "abs" ARCCOS ::= "arccos" ARCCOSH ::= "arccosh" ARCSIN ::= "arcsin"

ARCSINH ::= "arcsinh" ARCTAN ::= "arctan" ARCTANH ::= "arctanh"

CLOGLOG ::= "cloglog" COS ::= "cos" COSH ::= "cosh" EXP ::= "exp"

ICLOGLOG ::= "icloglog" ILOGIT ::= "ilogit" LOG ::= "log" LOGFACT ::= "logfact"

LOGGAM ::= "loggam" LOGIT ::= "logit" PHI ::= "phi" PROBIT ::= "probit"

ROUND ::= "round" SIN ::= "sin" SINH ::= "sinh" SOFTPLUS ::= "softplus"

SQRT ::= "sqrt" STEP ::= "step" TAN ::= "tan" TANH ::= "tanh"

TRUNC ::= "trunc" EQUALS ::= "equals" MAX ::= "max" MIN ::= "min"

POWER ::= "power" FOR ::= "for" IF ::= "if" IN ::= "in" MODEL ::= "model"

All distribution and function names in the BUGS language are followed immediately by a left bracket (that is there is no space between the last letter of the name and the left bracket). The same applies to the key words "for" and "if".

There are no built in distributions in the BUGS language. The compiled module for each distribution is loaded at run time if that distribution is used in the BUGS language model. The loaded module has to inform the compiler of the signiture of the distribution that it implements. and whether the distribution is a univariate distribution or a multivariate distribution. The BUGS software contains resource files containg pairs of distribution name module name. When the parser detect a distribution name in a BUGS language model the resource files are searched for that name and if found the relevent module loaded and the signiture recovered. If no match is found an error is reported. The signiture of distribution is in the form of a string:

signiture ::= [ { "s" | "v" } ] ["C"] ["T"]

where "s" denotes a scalar argument and "v" a tensor argument. The "C" and "T" denote that the distribution can be censored or truncated (both options can be possible at the same time). Note that distributions can not have expressions as arguments.

There are no tensor valued functions built into the BUGS language. Some scalar valued functions are also implemented externally.They are implemented in the same way as distributions are. Internal scalar functions and external scalar functions differ in the types of arguments that they can have. External scalar functions can have tensor valued arguments but not expression as arguments while internal scalar functions can have expressions as arguments but not tensor valued arguments.

Some examples of simple statements

| | |
|---|---|
| <uni_statement> | x ~ dnorm(mu, tau) |
| <uni_statement> | tau ~ dgamma(0.001, 0.001) |
| <uni_statement> | n ~ dcat(p[1:4]) |
| <uni_statement> | t[i, j] ~ dweib(r, mu[i])C(t.cen[i, j],) |
| <multi_statement> | x[1:2] ~ dmnorm(mu[], tau[,]) |
| <multi_statement> | tau[1:2, 1:2] ~ dwish(R[,], 2) |
| <scalar_statement> | mu[i] <- alpha - beta * pow(gamma,x[i]) |
| <scalar_statement> | logit(p[i]) <- alpha.star + beta * (x[i] - mean(x[])) |
| <tensor_statement> | Sigma[1 : M , 1 : M] <- inverse(Omega[ , ]) |

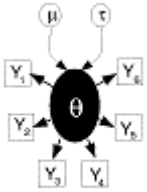
Some examples of compound statements

<for_statement>

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

<for_statement>

```
for( i in 1 : N ) {
   for( j in 1 : T ) {
      Y[i , j] ~ dnorm(mu[i , j],tau.c)
      mu[i , j] <- alpha[i] + beta[i] * (x[j] - xbar)
   }
   alpha[i] ~ dnorm(alpha.c,alpha.tau)
   beta[i] ~ dnorm(beta.c,beta.tau)
}
```




# Domains in the BUGS PPL

Andrew Thomas
MRC Biostatistics Unit
Cambridge

February 2022

Consider the BUGS language statement:

```
x ~ dnorm(mu, tau)
```

this has the effect of adding a term

$$1/2 (\log(\tau / 2\pi) - \tau (x - \mu)^2)$$

to the logarithm of the model's joint probability density. Because of the form of the normal probabity density tau must be greater than zero, therefore the BUGS language statement implies that tau is greater than zero. Consider the two BUGS language statements:

```
x ~ dnorm(mu, tau)
tau ~ dgamma(r, lambda)
```

the gamma distribution has support on the posative reals therefore tau is greater than zero and the first of the two statements is valid. We might not want to put a prior directly on the tau parameter. Instead we could write

```
x ~ dnorm(mu, tau)
log(tau) <- logTau
logTau ~ dnorm(0, 0.0001)
```

inverting the relation for tau gives tau <- exp(logTau) which is always posative so the first statement is valid. Finally consider the following two statements:

```
x ~ dnorm(mu, tau)
tau ~ dnorm(10, 1)
```

strictly speaking tau is not garanteed to be posative so the first statement is not strictly valid.

However in practice tau would never take a negative value.

The BUGS compiler is able to work out if a statement has valid parameters on its left hand side and if not to insert run time checks to test if the parameters have valid values. We call this topic the domain structure of the BUGS language.

Some distributions require parameters that as well as being posative have to be less than or equal to one: that is they must be proportions. Consider the BUGS language statement:

```
r ~ dbin(p, n)
```

the first parameter of the binomial distribution must be a proportion. The following two statements are valid

```
r ~ dbin(p, n)
r ~ dbeta(2, 2)
```

because the beta distribution has support on the interval [0,1]. Equally the two statements

```
r ~ dbin(p, n)
logit(p) <- alpha0 + alpha1 * x
```

are valid because inverting the second statement involving the logit function implies that p lies between 0 and 1 for all alpha0, alpha1 and x.

We could consider the slightly strange model:

```
r ~ dbin(p, n)
log(p) <- alpha0 + alpha1 * x
```

here it is easy to prove that p is greater than or equal to zero but not that it is less than one so a run time domain check is needed.

Two distributions, the catagorical and multinomial, have a vector valued parameter. Each component of this vector parameter must be a proportion and the sum of each of the components must always be one: that is the parameter must be a unit simpllex. For example

```
r[1:4] ~ dmulti(p[1:4], 10)
```

implies that 0 <= p[1], p[2], p[3], p[4] <= 1 and that p[1] + p[2] + p[3] + p[4] = 1. One way of ensuring this is to put a dirichlet prior on p[1:4]:

```
r[1:4] ~ dmulti(p[1:4], 10)
```

```
p[1:4] ~ ddirich(alpha[1:4])
```

A more complex model is multinomial logistic regression:

```
r[1:4] ~ dmulti(p[1:4], 10)

for(i in 1: 4) {
   log(mu[i]) <- alpha0 + alpha1 * x
   p[i] <- mu[i] / sum(mu[1:4])
}
```

Here it is easy to prove that each mu[i] is greater than or equal to zero, that each p[i] is a proportion and that the p[i] always sum to one.

One strategy for proving that a vector forms a unit simplex is to prove that at an initial point the vector lies on the unit simplex and that the derivatives of the sum ofthe vectors components with respect to its stochastic parents are all zero plus that each component is posative. For our multinomial logistic regression model it is simple to use automatic differentiation to prove that the derivatives of p[1] + p[2] + p[3] + p[4] wrt alpha0 and alpha1 are zero and that the initial point alpha0 = 0 and alpha1 = 0 that each mu[i] is one and each p[i] is 1/4 so that p[1] + p[2] + p[3] + p[4] = 1.

An more complex class of models are those used for ordered catagorical data such as the Bones or Inhaler models in Volume I of the BUGS examples. Here the p[i] vector is modeled as the difference of components of a Q[i] vector where each Q[i] is a proportion usually modeled using a logistic transformation and the Q[i] are ordered so that 1 > Q[1] > Q[2] > Q[n] > 0. Again automatic differentiation can be used to prove that the p[i] always sum to one but it is harder to prove that the p[i] are always posative and a run time domain check might be needed. For some models automatic differentiation can be used to prove the ordering of the the logit of the Q[i]. This is particularly simple for the Bones example.

The multivariate normal distribution has a matrix valued parameter: the precision matrix. This matrix must be symmetric and posative definite. We can use the derivative technique to prove that the matrix is always symmetric if it is symmetric for an initial value. It should also be possible to check that its diagonal components are always posative. However an runtime check will often be needed to prove that the matrix is posative definite.

One situation where the matrix can be proved to be posative definite is if it is specified as having a Wishart prior, consider the two statement:

```
x[1 : 2] ~ dmnor(mu[], tau[ , ])
tau[1 : 2, 1 : 2] ~ dwhish(Omega[ , ], 2)
```

the multivariate normal requires that tau[ , ] is symmetric posative definite and the Wishart makes sure that it is.

An alternative to a Whishart prior is to model the convariance matrix of the multivariate normal, for example

```
x[1 : 2] ~ dmnor(mu[], tau[ , ])
tau[1 : 2, 1 : 2] <- inverse(sigma[ , ])
sigma[1, 1] ~ dgamma(0.001, 0.001)
sigma[2, 2] ~ dgamma(0.001, 0.001)
sigma[1, 2] <- sqrt(sigma[1, 1] * sigma[2, 2]) * rho
sigma[2, 1] <- sigma[1, 2]
rho <- dunif(-1, 1)
```

The matrix inverse function expects that its parameter is a posative definite matrix and if so returns its inverse which is also a posative definite matrix. So the matrix inverse function can perform the test for posative definiteness.

Once a model has been given initial values each node, data, parameter and logical, is checked to make sure that its parameters have current values that respect the domains. The node's parameters nodes are then then checked to see whether they will always obey the required domain condition and if this can not be proved a run time check is inserted.

Each node type in the graphical model contains a field called props that is a bit set. Various bits of this set are used to denote the domain of the node and whether run time domain checks should be carried out. The following constants are used:

```
CONST
   (*    node properties    *)
   data* = 1;    (*    the node is data (has a fixed value)    *)
   integer* = 2;    (*    node has an integer value    *)
   posative* = 3;    (*    node has a posative value    *)
   proportion* = 4;    (*    node has a value between 0 and 1    *)
   simplex* = 5;    (*    node is component of unit symplex    *)
   cholesky* = 6;    (*    node is component of symetric posative definite matrix    *)
   checkDomain* = 7;    (*    check that node value respects domain    *)
```

If the props field of a node contains the bit data then the node has a fixed value. The node could be an observation with an assocciated distribution or it could be just a number such as a covariate.

If a variable node always has a posative value then its props is set to {3}. If a data node has a posative value its props is set to {1, 3}. If it is required that a variable node has a posative value but it can not be proved at compile time that the node is always posative then its props is set to {3, 7}.

Similarly if a variable node is always a proportion then its props is set to {3,4}. If a data node is a proportion its props is set to {1, 3, 4}. If it is required that a variable node is always a proportion but it can not be proved at compile time then its props is set to {3, 4, 7}

Finally if a variable node is always a component of a unit simplex then its props is set to {3, 4, 5}. If a data node is a component of a unit simplex then its props is set to {1, 3, 4, 5}. If it is required that a node is a component of a unit simplex but it can only be proved that the components of the simplex sum to one then its props are set to {3, 4, 5, 7} and the run time check will just need to verify that the node is posative.

Each class that implements a logical node in the BUGS language has a method called MarkDomain. This method tries to decide if the node will have an integer value, whether it will have a posative value and whether it will be a proportion by looking at the form of the function and the domain of its parents.

Consider this snippet of BUGS code:

```
for( i in 1 : I ) {
    for( j in 1 : J ) {
        X[i , j , 1:K] ~ dmulti(p[i , j , 1:K], n[i , j])
        n[i , j] <- sum(X[i , j , ])
        for( k in 1 : K ) {
            p[i , j , k] <- phi[i , j , k] / sum(phi[i , j , ])
            log(phi[i , j , k]) <- alpha[k] + beta[i , k] + gamma[j , k]
        }
    }
```

The node types for variables phi, sum(phi[i, j, ]) and p are GraphLog.Node, GraphSumation.SumNode and DynamicNode1.Node and the values of particular components of these variables are:

<phi[1,1,2]>GraphLog.Node^
.value REAL 1.0
.props SET {3, 15}
.label INTEGER 0
.work POINTER [02A8F6D0H]
.parents GraphNodes.Vector [02A8F6B0H]
.nesting INTEGER 2
.predictor GraphNodes.Node [02920F60H]

[028C3CF0H]GraphSumation.SumNode^
.value REAL 5.0
.props SET {3, 15}
.label INTEGER 0
.work POINTER [02A9F7C0H]
.parents GraphNodes.Vector [02A9F790H]
.nesting INTEGER 3
.vector GraphNodes.Vector [028C3D20H]

<p[1,1,2]>DynamicNode1.Node^
.value REAL 0.2
.props SET {3..5, 15}

```
.label    INTEGER    0
.work    POINTER    [02AB5660H] ◆
.parents    GraphNodes.Vector    [02AB5630H] ◆
.nesting    INTEGER    4
.l0    GraphLogical.Node    <phi[1,1,2]> ◆
.l1    GraphLogical.Node    [028C3CF0H] ◆
```

so the BUGS compiler has deduced that the phi are posative, that the sum of phi[i, j,] is posative, that the p are both posative and finally that p[i, j, ] sum to one so the vector p[i, j, ] must be a unit simplex.

The BUGS compiler generates a new class to represent the type DynamicNode1.Node and produces this for the class's MarkDomain method:

```
PROCEDURE (node: Node) MarkDomain;
   VAR
      integer0, posative0, proportion0, integer1, posative1, proportion1: BOOLEAN;
      val0, val1: REAL;
BEGIN
   (* logical *)
   integer0 := GraphNodes.integer IN node.l0.props;
   posative0 := GraphNodes.posative IN node.l0.props;
   proportion0 := GraphNodes.proportion IN node.l0.props;
   (* logical *)
   integer1 := GraphNodes.integer IN node.l1.props;
   posative1 := GraphNodes.posative IN node.l1.props;
   proportion1 := GraphNodes.proportion IN node.l1.props;
   (* div *)
   integer0 := FALSE;
   posative0 := posative0 & posative1;
   proportion0 := FALSE;
   IF integer0 THEN INCL(node.props, GraphNodes.integer) END;
   IF posative0 THEN INCL(node.props, GraphNodes.posative) END;
   IF proportion0 THEN INCL(node.props, GraphNodes.proportion) END;
END MarkDomain;
```

Note that this method does not prove that p is a proportion just that it is posative because its numerator and denomitor are posative.

When the BUGS compiler creates a stochastic node to represent the left hand side of a ~ relation it will set certain elements of the node's props set depending on the distribution on the right hand side. For example if the distribution is dbin then the integer and posative elements will be set, if dgamma then the posative element. When a constant node is created its props set is set according to the value of the constant: the following elements can be set integer, posative and proportion. When a stochastic node is data its props are the union of initial constant node and the props of the distribution.

Consider this example

```
model{
   x ~ dpois(5)
}

data list(x = 5.5)
```

When the node for x is first created it is a constant node and has posative set in its props set. When the model is compiled a poisson type node is created for x and has the integer and posative set in its props set. When the node is checked a conflict is detected between the requirement that x has an integer value and the fact that its value is 5.5.

After the logical nodes in the model have been created andtheir internal fields writen their MarkDomain method is called to work out their domain. This is done in procedure GraphLogical.MarkDomains. Note that if the vector of logical nodes input into this procedure are sorted according to nesting level the newly discovered domain information will be propagated upwards.

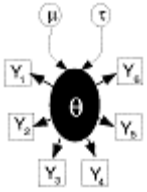
Some examples of tests of run time domain checks:

```
model{ x ~ dnorm(0, tau) tau ~ dnorm(4, 1) }

model{ x ~ dnorm(0, tau1) tau1 <- 1 + tau tau ~ dnorm(3, 1) }

model{
   x[1:2] ~ dmnorm(mu[], tau[,])
   mu[1] <- 0 mu[2] <- 0
   sigma[1,1] <- 1 sigma[1, 2] <- rho
   sigma[2, 1] <- rho sigma[2, 2] <- 0.9999
   tau[1:2, 1:2] <- inverse(sigma[ , ])
   rho ~ dunif(-1, 1)
}
```

In these simple examples the forward sampling algorithms is used. In the first example a value of tau will eventually be sampled that is less than zero. In the second example a value of tau less than minus one will be sampled and this will make tau1 negative. In the final example a value of rho will eventually be sampled such that the determinant of sigma (0.9999 - rho*rho) is negative and a domain error will occur. The error messages produced when running these models are:

```
update error for node tau in algorithm univariate forward failed domain check for  tau

update error for node tau in algorithm univariate forward failed domain check for  tau1

update error for node rho in algorithm univariate forward failed domain check for  tau[1,1]
```

There are a number of questions about run time domain checks. For example do we need them? We could just let the simulation crash. In a way they are like warnings. There might be a problem. So should the BUGS compiler report that domain checks have been inserted. If so in how much detail? For every node? By name of variable in the BUGS language model? When a domain check fails it is possible to give an error. But can the simulation be continued afterwards?




# Helping the BUGS Project

## Introduction

The BUGS projects needs your help. There are many ways that you, the BUGS community, can help our BUGS project. Many different skills are valuable to our BUGS project. The BUGS project is small so you can easily make a real difference. People might get the idea that BUGS is a one man and his dog set up. This is not quite right. It is more a small fraction of a person and their cat (but the cat has moved on to playing with mice in the sky). So any contribution you make will be large and important.

The BUGS software is built from scratch so there are no external libraries or packages that you need to become experts on. The source code and documentation is all there is to the project and you do not need to understand all of it to be able to make contributions. The source code is fully documented: each procedure, data type, variable and constant is described and is only a right click away. There are no header files just interface definition files. There are no includes just imports. The software development tools are extremely simple to use and powerful. The BUGS software is written in the simple object orientated language Component Pascal. This language is so simple and small that it can be learnt in a few hours.

Below I list some areas where contributions would be really helpful.

## Hamiltonian Monte Carlo

BUGS can now use HMC to make inference. There are a large number of topics to work on here. A obvious question is what to do if HMC can not be used on all the parameters in the model. For simple mixture models the allocation variables can be marginalized out and this often works well. Models with a more complex structure of descrete parameters might be amendable to a marginalization approach using say junction trees. Models with change points seem more difficult. Models with censored observations could be handled by a marginalization approach if the log of the cdf / complemtenary cdf can be differentiated. How can HMC be used when there are constraints between parameters such as the CAR prior? The NUTS sampler does not yet work in BUGS. It would be great to get NUTS and some of its variants working.

## Polya-gamma sampling

BUGS can now use polya-gamma auxillary variables to rewrite logistic regression models as normal regression models. This also works for multileval logistic models. However the method seems to work best when both the fixed and random effects are updated as one block. If this block is large sparse matrix algebra should be used for speed. At present if the number of random effects is large univariate normal sampling is used which can be less efficient. There is sparse matrix code in the BUGS software but it needs connecting up with the polya-gamma stuff. More efficient generators for the polya-gammadeviates would also speed things up.

## Parallel versions of BUGS

BUGS can automatically spread the sampling of a MCMC chain over several computational cores by splitting the graphical model into shards. Promising results have been obtained for Gibbs sampling and small block metropolis algorithms. In theory the parallelization of the autodiff code needed for HMC should be simple. One important property that BUGS can detect is seperability of graph shards that run on seperate cores. For seperable models no information about sampled values needs exchanging between cores. A value for each fixed effect is sampled on all cores, contributions to the log likelihood or diff log likelihood for fixed effects are calculated on each core and combined using mpi reduce. If the shards are not seperable then when each parameter changes on a core this change must be propagated to the other cores using mpi allgather. Does a backward autodiff algorithm work for calculating the derivatives of the log joint probability distribution when the shards are not seperable?

## Other computer architectures

The software development tools used in the BUGS project target Intel type chips. They run on both Windows and Linux. The multicore version of BUGS makes a small executable version of the MCMC sampling algorithm tailored to a specific model. This executable consists of a small number of modules statically linked together. There are tools to translate the BUGS source code into C source code that can then be compiled to object code and linked into an executable. If this could be automated then BUGS could produce worker programs to run on any CPU type. If the BUGS software could be translated into C and compiled to .o files these would just need linking at run time. Experience of C compilers and linkers would be very helpfull. An alternative to the C source code route would be to use LLVM. The LLVM back end of the Component Pascal compiler is a little raw but bypassing C and all its horrors would be very worthwhile.

When BUGS is given a new model it generates source code for each distinct type of logical relation in the new model. For a typical model there would be a few to half a dozen source code modules generated. These modules are then compiled into object code using the Component Pascal compiler and loaded into the running version of the BUGS software using its linking loader. These source code modules could be translated into C, the C compiler called to produce .o files which could be linked with the already existing .o files to make an executable. For the Rats example model a multicore MCMC sampler is built out of the following modules:

Kernel+ Files HostFiles Files64

| | | | |
|---|---|---|---|
| HostFiles64 | Math | Strings | Stores |
| Stores64 | MathFunctional | MathAESolver | MathODE |
| MathRungeKutta45 | MathFunc | MathSort | MathMatrix |
| MathRandnum | BugsRandnum | GraphNodes | GraphLogical |
| GraphScalar | GraphKernel | GraphRules | GraphStochastic |
| MathTT800 | MpiMPI | MpiMsimp | MpiWorker |
| MonitorDeviance | MonitorPlugin | ParallelRandnum | BugsRegistry |
| UpdaterUpdaters | ParallelActions | ParallelFiles | GraphMultivariate |
| UpdaterAuxillary | UpdaterJoint | ParallelHMC | GraphConstant |
| DynamicNode0 | GraphUnivariate | GraphConjugateUV | GraphGamma |
| GraphNormal | GraphDummyMV | GraphDummy | UpdaterUnivariate |
| UpdaterContinuous | UpdaterGamma | GraphConjugateMV | UpdaterNormal |
| GraphLoader0 | UpdaterLoader0 | ParallelWorker | |

Where the modules in red are generated by the BUGS software. All the other modules could be translated to C and compiled prior to first running the BUGS. The modules in blue are part of the BUGS software while those in black are part of the development enviroment.

## New probability distributions

In the BUGS software probability distributions are represented as classes. A new class needs writing when a new distribution is added to the software. This is documented. Many users have contributed new distributions to the BUGS sofware. The task is fairly self contained and does not require knowledge of large parts of the BUGS system. Two distribution the Zipf and the skew normal have been partially implemented. It would be good to see if they work, if there are examples to test them etc.

## Variational Bayes

BUGS is now able to make Variational Bayesian Inference using the ADVI algorithm. Does ADVI work well enough to spend more effort on? Can it be parallelized? It is quite like HMC. Is it worth integrating ADVI into BUGS in the same way as HMC? Importance sampling might improve ADVI, Pareto smoothing needs implementing.

## Integration with R

Does R have metaprogramming? Could BUGS manipulate R objects? Say use R objects as data and write the values produced by MCMC into R objects.

## Pharmokinetic modelling

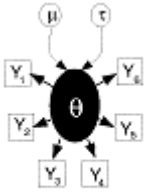
The BUGS software has two subsystems for use in pharmokinetic modelling. PKBugs is a tool

for specifying PK models using concepts from the Nomem package rather than using the BUGS language. Pharmaco is a number of building blocks for PK models. Recentley the Pharmaco subsystem has been modified so that HMC can be used for inference. We are not experts on these models and need help to make progress.

## Reversible jump models

The BUGS software has some support for reversible jump models. This is no longer working but it would be good to get it working again. Two examples of reversible jump modelling are the Hald and Curves examples. I have used VD standing for variable dimension in the name of all the modules connected to reversible jump.

The Graph subsystem contains three helper modules (GraphVD, GraphVDDescrete and GraphVDContinuous) to support the construction of reversible jump modelling components. The Graph subsystem also contains three modules that implement concrete reversible jump model components: GraphCoSelection, GraphSpline and GraphSplinecon. Specialized sampling algorithms are needed to sample reversible jump models. These are in the Updater subsystem in modules UpdaterVD, UpdaterVDMVN, UpdaterVDMVNContinuous and UpdaterVDMVNDescrete.




# Thanks and acknowledgements

Andrew Thomas
MRC Biostatistics Unit
Cambridge

November 2021

The BUGS team would like to thank the many people who have made the BUGS software possible.

The warmest thanks go to Prof Nicklaus Wirth for designing a sequence of elegant, balanced, powerful programming languages each better than it predecessor. At various times BUGS has been implemented in Modula-2, Oberon-2 and Component Pascal.

Thanks to Oberon Microsystems for the Component Pascal Language and the BlackBox development enviroment. Being able to move away from the command prompt to document centred user interfaces has been liberating. A big thanks for making the tools and libraries open source.

Thanks to the BlackBox Centre who now maintain and improve the development tools and have ported the tools to many Linuxess. Particular thanks to Ivan Denisov who has answered our many stupid questions about Linux issues. The Centre's work on a LLVM back end to the Component Pascal Compiler looks very interesting.

Thanks to the many people who have contributed new modelling components to the BUGS Probabilistic Programming Language and in particular to Vijay Kumar for the Relia subsystem for reliability analysis.

Thanks to all the people who have been part of the BUGS team over the years and who have put up with many changes of internal data structures used by the software.

Thanks for the support group that has helped new users with questions about the many

stangenesses of the BUGS langauge and obscure error messages.

A final thanks for everyone who has used the BUGS software.