WinBUGS Development Interface (WBDev) – Implementing your own functions

Dave Lunn
Imperial College School of Medicine, London, UK
d.lunn@imperial.ac.uk

September 1, 2004

1 Introduction

This document explains how you can implement arbitrarily complex logical functions in WinBUGS (1.4) by 'hard-wiring' them into the system via compiled Pascal code. There are three main advantages to doing this: first, 'hard-wired' functions can be evaluated much more quickly than their BUGS-language equivalents; second, the full flexibility of a general-purpose computer language is available for specifying the desired function, and so piecewise functions, for example, can be specified straightforwardly whereas their specification via the BUGS language (using the `step(.)` function) can be somewhat awkward; finally, the practice of hiding the details of complex logical functions within 'hard-wired' components can lead to vastly simplified WinBUGS code for the required statistical model, which reduces the likelihood of coding errors and is easier both to read and to modify. We demonstrate how to implement such 'hard-wired' components via a worked example in which the following function becomes a single element of the BUGS language.

$$C(t) = \begin{cases} 0 & t < 0 \\ \frac{D}{V}\frac{k_a}{k_a - k_e}\left[\exp(-k_e t) - \exp(-k_a t)\right] & t \geq 0 \end{cases} \tag{1}$$

This is known in the field of pharmacokinetics as a "one compartment open model with first-order absorption". Here $C(t)$ denotes the concentration, at time t, of drug in blood plasma following (oral) administration of a dose D; the system parameters V, k_e and k_a denote the drug's volume of distribution, elimination rate constant, and (first-order) absorption rate constant, respectively. For reasons of identifiability we parameterise the model in terms of $\theta' = \log(V, k_e, k_a - k_e)$ – thus every possible combination of real values (positive or negative) for the elements of θ gives rise to physically feasible values for V, k_e and k_a and generates a *distinct* concentration-time profile.

2 System set-up

Computer code for the scalar-valued function defined in Equation 1 above can be found in the file `WBDev/Mod/ScalarTemplate.odc`. This is a *Component Pascal module* that under normal circumstances, i.e. if you had written it yourself, would need to be compiled and then 'linked' into the core WinBUGS software before it could be used from within the software – I have already done this for this example, however, to aid with this exposition (see later). Before discussing the new module in detail we provide instructions on how to set up your system so that Component Pascal code can be compiled.

1. Download *BlackBox Component Builder* from the following web-page:
   `http://www.oberon.ch/blackbox.html`

2. Unzip the downloaded file, if necessary. Install 'BlackBox' by double-clicking on the `Setup.exe` icon and following the instructions. The software should be installed into the new directory `Program Files/BlackBox`.

3. Open `My Computer` (or its equivalent) and navigate to the `Program Files/WinBUGS14` directory; then press `Ctrl+A` (or select `Select All` from the `Edit` menu) to select all files and sub-directories within the `WinBUGS14` directory. Now press `Ctrl+C` (or select `Copy` from the `Edit` menu) to copy those files and sub-directories.

4. Continue using `My Computer` to navigate to the `Program Files/BlackBox` directory and then press `Ctrl+V` (or select `Paste` from the `Edit` menu) to paste the copied files and sub-directories to this location. Select "`Yes to All`" if prompted about replacing existing files.

5. Now your copy of BlackBox should include the *full* functionality of WinBUGS 1.4 (including this `WBDev` interface) within it, and so the `BlackBox.exe` icon on the desk-top or that in the `Program Files/BlackBox` directory can be used either to run WinBUGS in the normal way or to conduct WinBUGS development work (or even more general Component Pascal programming).

3 New module – "WBDevScalarTemplate"

Now start your copy of BlackBox and open the new module:

`Program Files/BlackBox/WBDev/Mod/ScalarTemplate.odc`

Note that to reduce the risk of errors creeping into the system we recommend that all other new components are also stored in the `WBDev/Mod` directory. As the name suggests, the `ScalarTemplate` module can be used as a template for such new components, so long as they represent scalar-valued functions (see later for details regarding vector-valued functions). Please note that only those parts of the code that are currently marked in blue should be modified. The following notes pertain to areas of code labelled with the corresponding numbers within comment markers, i.e. `(*` and `*)`, e.g. `(* this is a comment in Component Pascal *)`.

`(*1*)` The first line of a Component Pascal module should always read `MODULE`, followed by the module's name, in this case `WBDevScalarTemplate`, followed by a semi-colon. The last line of the module should read `END`, followed by the module's name, followed by a *full stop* (period). All new module names for new components of this type should begin with `WBDev`; the corresponding *file* names should be identical but with the `WBDev` prefix removed (they must also begin with at least one capital letter); all new files of this type should be saved in the `WBDev/Mod` directory.

`(*2*)` Various other modules can be 'imported' into each new module, which means that procedures and/or data structures defined in those modules can be used/exploited from within the new module. The `Math` module is an integral part of the BlackBox software since it defines many fundamental mathematical functions, which are called from within other modules via the syntax `Math.` followed by the relevant procedure name, e.g. `Math.Ln(.)` for natural logarithms, `Math.Exp(.)` for exponentials – see lines `(*13*)`–`(*15*)` and `(*21*)`. Documentation regarding the `Math` module can be accessed by highlighting the word `Math` in BlackBox and selecting `Documentation` from the *second* `Info` menu (the first `Info` menu belongs to WinBUGS).

`(*3*)`–`(*6*)` As the name suggests, we use the `DeclareArgTypes(.)` procedure to declare the types of arguments required to define the function of interest. In the case of our one compartment pharmacokinetic model defined in Eq. 1 above, the required arguments are: the parameter vector θ, the dose D, and the time t. Thus we have a vector followed by two scalars and so we set the `args` variable equal to `"vss"` (`v` for vector; `s` for scalar).

`(*7*)`–`(*26*)` The `Evaluate` procedure is used to define a variable called `value`, which stores the function's current value (given the current values of its arguments). Throughout the procedure we make use of a variable called `func`, which represents the function itself. In particular, we refer several

times to one of its 'internal' fields, `arguments`. This is an 'irregular' matrix where each row corresponds to one of the arguments declared in `DeclareArgTypes(.)` above (in the same order). If a particular argument is a vector (`v`) then the length of the corresponding row of `func.arguments` is equal to the length of that vector, whereas if an argument is a scalar (`s`) then the length of the corresponding row is 1. The procedure call `func.arguments[i][j].Value()` returns the value of the `j`th element of the `i`th argument. (Note that all array indices start at 0 in Component Pascal rather than 1.) Thus the value of $\log k_e$, for example, can be obtained via the call `func.arguments[0][1].Value()` – because $\log k_e$ is the second element (index = 1) of the θ vector, which is the function's first argument (index = 0). On lines `(*8*)` and `(*9*)` we define three constants that allow us to index the various rows of `func.arguments` via meaningful names rather than directly by the relevant numbers themselves, i.e. we can use the names `parameters`, `dose` and `time` in place of `0`, `1` and `2` to access the function's first, second and third arguments, respectively. This is by no means essential but *is* considered to be 'good practice' as it reduces the likelihood of coding errors arising.

`(*10*)`–`(*11*)` Note that any number of 'local' variables can be declared and used to aid in specifying the new function, so long as their names do not clash with other variable/procedure names – the compiler (`Ctrl+K`) will normally inform the programmer of any errors.

`(*18*)`–`(*22*)` This is a standard "IF/THEN/ELSE" statement in Component Pascal; note that the "ELSE" branch can be omitted where appropriate – see below.

`(*23*)`–`(*25*)` This is a standard "IF" statement in Component Pascal – here we simply set equal to zero any negligibly small values that have been calculated as negative due to (a lack of) machine precision.

Please note that (almost) every Component Pascal statement ends with a semi-colon. Hopefully this brief example demonstrates sufficient use of the Component Pascal syntax that the reader is able to begin writing their own modules from this template. Detailed documentation on both BlackBox and the Component Pascal language can be accessed via the file:

`Program Files/BlackBox/Docu/Help.odc`

Further insight may also be gained by examining our second template, which shows how to implement new components to represent vector-valued functions – see later. Please read the instructions below before attempting to write your own modules.

4 Using "WBDevScalarTemplate" as a template

The following instructions should be followed closely when defining a new BUGS function via the `WBDevScalarTemplate` template:

1. Choose a name for the new component, `NewFunction`, say (the new name *must* begin with a capital letter). Start your copy of BlackBox and open the `WBDev/Mod/ScalarTemplate.odc` template from within it; then save the template under the new name, e.g. `WBDev/Mod/NewFunction.odc` – **be careful not to overwrite an existing module!** Now modify the module name both at the top and at the bottom of the new file – change these from `WBDevScalarTemplate` to `WBDev` followed by the new component's name, e.g. `WBDevNewFunction`. Save and compile the new component by pressing `Ctrl+S` (save) followed by `Ctrl+K` (compile) – there should be no compilation errors at this stage since only the module name has been changed.

2. Now modify the code in the new module according to the desired function, i.e. declare the types of arguments required on the line labelled `(*5*)` and redefine the `Evaluate(.)` procedure. You can save the new module at any time by pressing `Ctrl+S` (or by selecting `Save` from the `File` menu). You can also attempt to compile the code at any time by pressing `Ctrl+K` (or by selecting `Compile` from the `Dev` menu). If there are any compilation errors when you attempt to compile your code, each one will be marked in the code by a grey box with a white cross running through

it. An error message pertaining to the first error will be displayed on the status bar (which lies across the bottom of the BlackBox 'program window') and the cursor should automatically position itself next to the corresponding grey box. We advise that you deal with any compilation errors in order, but if for some reason this makes things awkward (or is not possible) then error messages for specific compilation errors can be obtained by clicking on the appropriate grey boxes – a single click shows the error message on the status bar whereas double-clicking reveals it within the code, in place of the grey box (double-click again to revert back to the grey box).

3. Once the new module has been successfully compiled (and saved) then it can be 'linked' into the WinBUGS software by modifying the file `WBDev/Rsrc/Functions.odc`. The first line of this file contains the required entry for the one compartment pharmacokinetic model defined in the `WBDevScalarTemplate` module:

```
s <- "one.comp.pk.model"(v, s, s)                "WBDevScalarTemplate.Install"
```

The notation is described as follows: '`s`' means that the component represents a scalar as opposed to a vector (`v`), whereas '`<-`' indicates that the component represents a function as opposed to a distribution (∼); '`one.comp.pk.model`' is the name chosen for specifying the new function via the BUGS language; '`(v, s, s)`' tells the system that the function requires one vector and two scalar arguments (in that order) as declared in the `DeclareArgTypes(.)` procedure; and '`WBDevScalarTemplate.Install`' is the 'installation' procedure for the component – this is defined towards the bottom of the template module. Make a copy of this first line immediately beneath it, and replace `one.comp.pk.model` on the *new* line with the desired BUGS-language name for your new function, e.g. `new.function` (this is the name with which you wish to refer to the new function during future WinBUGS sessions). Now declare the new function's arguments and specify the name of the module where its 'installation' procedure can be found – replace `WBDevScalarTemplate` with your new module's name, e.g. `WBDevNewFunction`. You should end up with something like

```
s <- "one.comp.pk.model"(v, s, s)                "WBDevScalarTemplate.Install"
s <- "new.function"(s, v)                        "WBDevNewFunction.Install"
```

at the top of the `WBDev/Rsrc/Functions.odc` file. Now save the new `WBDev/Rsrc/Functions.odc` file – the new component will be available from the next time that BlackBox is started, so don't forget to shut down the software before trying to use your new function. The new function in the illustrative example above could be accessed from within WinBUGS via BUGS syntax similar to the following:

```
model {
...
value <- new.function(x, par[1:p])
...
}
```

Good luck!

5 Vector-valued functions – "WBDevVectorTemplate"

Vector-valued functions can be 'hard-wired' into the system by making use of a different template, which can be found in `WBDev/Mod/VectorTemplate.odc`. Here we define a version of our one compartment pharmacokinetic model that is now vector-valued by virtue of the fact that we now wish to evaluate it at a vector of times rather than a single time. Except for a few minor points discussed below, the details of implementation are exactly analogous to those for scalar-valued functions.

`(*5*)` The `args` variable is now set equal to `"vsv"` rather than `"vss"` since the function's third argument has changed from a single time to a vector of times.

`(*7*)`, `(*30*)` The `Evaluate(.)` procedure must now return *an array* of values, via the '`values`' variable, rather than a scalar (previously via '`value`').

`(*18*)` Component Pascal's `LEN(.)` function returns the length of the specified argument, so long as that argument is a one-dimensional array: thus `LEN(func.arguments[times])` is the number of times at which the function is to be evaluated. If the array has more than one dimension then `LEN(.)` returns the length of the *first* dimension – see the BlackBox documentation for more details.

`(*19*)`–`(*32*)` The lines labelled `(*19*)`, `(*20*)`, `(*31*)` and `(*32*)` form the basic structure of a Component Pascal `WHILE`-loop. (Note that `INC(i)` increments the value of `i` by one.) During each 'pass' through the loop, the `t` variable is set equal to one of the times at which the function is to be evaluated and the corresponding evaluation is performed by making use of the 'temporary' variable `val`. At the end of each pass, `values[i]` (`i` $= 0, ...,$ `numTimes` $-$ `1`) is set equal to `val`.