SYMFOR Code Documentation

Paul Phillips

22nd September 2000

Abstract

SYMFOR is a software framework that contains models of ecological forest processes and forest management strategies. It is designed for use by researchers based in the developed and the developing world, from universities, research institutes, government advisory bodies, private industry and NGOs. It is designed to be straightforward to use in a simple case, but to have advanced levels of control for expert users. It has graphical data representations for run-time interpretation, and flexible data output mechanisms for larger analyses. It runs on IBM compatible personal computers using modern Windows operating systems.

For a combination of historical and practical reasons, the front end, or user interface, is written in Visual Basic. It is referred to as the "model manager". The data storage, the models and the data processing are written in C++ as a dynamic link library, referred to as the "DLL". The model manager liases between the user (the screen) and the DLL, with information necessarily passing both ways.

This document attempts to describe the structure of the software, from both a conceptual and a file-based perspective. Prospective maintainers of the code can use this document as a reference guide or general introduction. The intention is that they will then be able to know where to start for a particular problem, and will have an overall picture of how the software functions.

DFID Department for International Development

This document is an output from a project funded through the Forestry Research Programme of the UK Department For International development (DFID) for the benefit of developing countries. The views expressed are not necessarily those of DFID.
Project R6915 Forestry Research Programme.

Contents

1 Introduction

SYMFOR is a software framework that contains models of ecological forest processes and forest management strategies. It is designed for use by researchers based in the developed and the developing world, from universities, research institutes, government advisory bodies, private industry and NGOs. It is designed to be straightforward to use in a simple case, but to have advanced levels of control for expert users. It has graphical data representations for run-time interpretation, and flexible data output mechanisms for larger analyses. It runs on IBM compatible personal computers using modern Windows operating systems.

For a combination of historical and practical reasons, the front end, or user interface, is written in Visual Basic. It is referred to as the "model manager". The data storage, the models and the data processing are written in C++ as a dynamic link library, referred to as the "DLL". The model manager liases between the user (the screen) and the DLL, with information necessarily passing both ways.

There are on-line help pages, written in HTML for portability, that cover the usage of the software itself. A series of technical notes apply to SYMFOR, and should be referred to for training material, explanatory documents of models, case-studies and reports of the application of SYMFOR to real forest management issues problems. The structure and maintenance of the help pages is described here.

SYMFOR is a product, and has to be shipped as a conventional package, with a set-up program giving the usual options. The packages used to create these, and their scripts, are described here.

1.1 What is assumed of the reader

Some familiarity with SYMFOR is essential before wading into the code. If you wish to become familiar with the code, first get used to what SYMFOR does, by getting a copy and trying the options – doing a single run, using displays, changing the management options model to implement logging, outputting data to a file and using the multiple-run facility. If you are not familiar with these things, understanding the code will be much harder.

Prior knowledge of the BASIC and C++ languages will greatly aid understanding of what the code is trying to do. If those languages are not known, the reader is expected at least to know an object oriented language and be familiar with OO concepts, pointers and common data structures such as arrays and linked lists.

1.2 History of SYMFOR development

SYMFOR was conceived and initially developed by Allen Young and Robert Muetzelfeldt from 1991 until 1997, under the ODA's (now called DFID) Indonesian Tropical Forest Management Programme (ITFMP). Allen's PhD was part-time, and with the rapid advances in computing hardware and software during that period, he spent much time learning and implementing new programming methods. They also spent time developing and training a user-base in Indonesia. At the end of Allen's funding, the software used 16-bit compilers and would not run on 32-bit machines. The models within the framework lacked a thorough statistical basis, and implementing alternative models required programming in C by the user.

The DFID Forest Research Programme project R6915 was created in 1997, and took SYMFOR as a starting point. The user-base was considered valuable, and to sustain it the name and user-interface were retained. A metamorphic process changed the underlying structure of the framework to its

current state. It is now 32-bit, object-oriented and self-supporting under the windows operating environment. Through an iterative process, it has been developed to be a stable software that is well documented and now almost unchanging with time.

1.3 Languages, versions, operating systems

SYMFOR runs under any 32-bit Microsoft Windows operating system (95, 98, Me, NT or 2000). The user-interface utilises aspects of the Windows system directly for graphics, and so is not portable out of that environment. The Model Manager is written in Microsoft Visual Basic v5.0. The DLL is written in Microsoft Visual C++ v6.0, although it also compiles flawlessly under v5.0. No "visual" aspects of the C++ package are used in the DLL, but some Microsoft-specific variable definitions are used.

The SYMFOR version in 1997 was known as SYMFOR 2.2. The first release with the new framework structure was called SYMFOR 97, followed by releases each year until the current version release of SYMFOR 2000. Incremental changes have been made since the initial SYMFOR 2000 release, and the SYMFOR web site (www.symfor.org) should be consulted for the latest information.

1.4 Coding principles, conventions, practices

The software has been written to be maintained. Where possible, variable names are self-explanatory, at least in context, as are subroutine names and file names. The code is commented thoroughly, with later modifications being labelled with the date and author. Consistent indents are used to aid loop-tracking. Long subroutines are avoided as much as possible, and any multiple-coding has been avoided by creating subroutines that are called from two or more places. Object orientation has been used where an appropriate conceptual hierarchical structure exists that would benefit in clarity, speed and maintenance from representation in an OO manner. User-defined OO structures have not been used in the general case. An effort has been made to create the framework such that it can cope, without alteration to the structure, with any individual-based forest growth model, and any forest management model that impacts upon sample plot data.

No versioning software has been used in the development of this product. It was thought that since there was no parallel development (by two or more developers), and there was no desire to recreate old versions, there was little to be gained for the inefficiency and cost of a versioning tool.

1.5 On-line help pages

The on-line help pages are over 300 HTML files, linked to produce a complete documentation system. They are updated with each release of the software, and as they themselves are developed and improved. The help files are provided with the SYMFOR framework on installation CDs or download, and are also available on the SYMFOR web site without a complete download. They have been developed in parallel with the software itself. They document the issues regarding usage, and only include development issues that are likely to be common (such as adding a new module for part of a model).

1.6 Structure of this document and how to use it.

This document is designed for use by people wanting to understand how the software works, alter it, debug it, maintain it or produce release versions. Accordingly, it can be used as a reference manual (using the contents list or index), or read as an introduction to the software before or during examining the files on a computer. The document is divided into sections about the MM, the DLL, and then the other associated components of the SYMFOR system.

2 The Model Manager

This section explains what the basic functions of the MM are, and how they are achieved using VB code.

2.1 VB usage

2.1.1 Files

"Form"s are the code files that create windows (suffix ".frm"). Some forms have images associated with them that are stored in files with the same name but a different suffix (".frx"). Bitmap images loaded into a form and stored as a .frx file are not recoverable as bitmap files – be careful not to delete the original bitmap if you might want to edit it. "Modules" are the VB terms for code files that contain globally available subroutines and functions (suffix ".bas"). The MDI form is the first and controlling window in a VB application. When it is closed, the application ends. The main SYMFOR window, containing the menu, is the SYMFOR MDI form (called modman2.frm).

2.1.2 Variables and Scope

Variables in VB can be assigned a data type in one of two ways. I preferred the method of putting a type declaration character at the variable name end every time the variable is used. It helps reduce incorrect usage. The common ones are: $ for string, & for "long" (32 bit integer), ! for real. E.g., "idatapoint&". I try to use variables beginning with the letters i, j, k, l, m, n as integer (long) variables, and other letters indicate other types. If a variable is not assigned a type in VB, it is assigned a type "variant", whose meaning and usage is ambiguous to me. I have avoided that.

Variables declared in subroutines or functions do not have any defined meaning outside those functions. Variables declared at the start of a module or form (in "general declarations", outside all subroutines or functions) are valid for use by any code in the whole of that file. Variables declared as "Global" in modules are available for use anywhere in the whole VB program. I have defined all the global variables in modman.bas.

2.1.3 Debugging and creating executables

BASIC is an interpreted language, but Microsoft have realised that people may want to produce a standalone executable program using VB. From the "file" menu, there's an option to "Create executable…". The executable cannot be debugged, but runs about 20% faster than the interactive version, and does not require the VB software to provide the interactive environment. To debug in VB, press F5 or the play (right-pointing triangle) button. Breakpoints can be placed with F9 after putting the cursor on that line, then step through with F8 (see the debug menu when debugging).

2.1.4 Adding controls

When the SYMFOR VB project is copied to a new computer and used for the first time, it may not work correctly. As VB is loading the project, it may discover that it can't find particular controls, or doesn't have the licenses to use them at design time. This should be rectified. First, load and register the controls from the VB CD: go to the directory "tools\controls", add grid32.ocx, threed32.ocx, comctl32.ocx, graph32.ocx, gsw32.ocx, gswdll32.dll and vbctrls.reg to "c:\windows\system" (or equivalent), and run "regedit vbctrls.reg". It may also be necessary to run "regsvr32.exe grid32.ocx". Then add them to the project (click "ok" to all the loading errors, then use the "project" menu, "components" option. The selected controls should be: Microsoft chart control, Microsoft common dialog control 6.0, Microsoft grid control, Microsoft sysinfo control 6.0, Microsoft Windows common controls 5.0 and Sheridan 3-d controls).

2.2 Start-up

Double-clicking the "symfor2000.exe" icon (or shortcut) starts SYMFOR. The first code to be executed is in modman2.frm, MDIForm_load(). It checks the directory that should contain the auxiliary files ("modeldirectory$"), starts the log file, shows the title screen for a second or two, initialises the DLL (memory, model data structures, etc), sets some defaults for data input variables and waits for the user to do something.

2.3 Auxiliary files and structure

The Model Manager program is called "symfor2000.exe", and it links to the DLL, "sf2000.dll". These are the core SYMFOR files. Other files in the same directory allow SYMFOR to function smoothly and give more flexibility to the user.

Several files are used by the Model Manager to control SYMFOR and to store settings from previous SYMFOR sessions. They should all be kept in the same directory that the SYMFOR executable is in. They should not be edited manually except by an expert user, and then only with care – make a copy of the old file first in case it doesn't work.

The most basic auxiliary file is "mm.ini" (Model Manager initialisation). It contains default settings for data input, the displays, the default run length and the model directory location.

The model definitions are stored in "mods.txt" and "pars.txt". The definitions are simply a set of choices of module (or algorithm), and the values of the parameters included in that module. Many different definitions may be created in SYMFOR by the user and saved (in mods.txt and pars.txt). The default model is defined by the module set called "default" and the parameter set called "default". These may not be deleted by the user. These files are read into the MM at start-up, and saved whenever the user saves a module or parameter set. They are not saved on exit.

The sets of instructions about multiple runs are saved in "runsets.ini". These are used by the multiple run editor.

"autorun.exe" is a freeware program that will open a file as if it has been double-clicked in Windows. It is used to open help files with the default web browser, whatever that is. "searchengine.exe", "index.txt" and "search.html" are all used by the help pages search facility. "postp1.exe" is a program that can be used to analyse results that may be output by SYMFOR. It has its own help pages.

Three log files are produced by SYMFOR to help diagnose problems if SYMFOR crashes. "symfor.log" contains information about the SYMFOR session and each run that is done. "symforpars.log" and "symformods.log" contain copies of the last parameter set and module set, respectively, that were used.

2.4 Menus

SYMFOR is primarily controlled by the user with the menu in modman2. The menu can be edited in VB by viewing the modman2 form (not the code) and right-clicking the mouse. This gives the option to edit the menu. Each menu item represents a subroutine in the modman2 code. Selecting the menu item from the modman2 form will bring up the code editor for the appropriate subroutine.

2.5 Runs

There are two types of run. The single run may be started either by clicking the running person icon, or by choosing the “Start run” and “New single run” options from the menu. The multiple run editor may be started by using the menu and “Multiple run”. For any single run (or a run within a multiple run), SYMFOR requires: a model definition (a module set and a parameter set), input tree and stand data sets, the run length and optionally data output tables.

In both cases, the routine startexe() (MM) is used. For a single run, this routine uses windows to ask the user for input data and to check the run settings, loads the data and model definition, then brings up a run control window (“runcont.frm”). The multiple run provides that information itself, and startexe() simply loads the data and model definition. A global variable, modelstate&, is changed to keep track of the state of the program. Global constants represent the different states, and depending on those states the available buttons and menu items change. For example, it is only possible to see a map of the trees when tree data is loaded. Single runs are controlled by runcont.frm, and multiple runs are controlled by multirunf.frm.

Multiple runs are a collection of one or more instructions. Each instruction will be to repeat a particular single run one or more times. The multiple run instruction editor (mruninstr.frm) is used to change the details of a particular instruction. The instruction is passed back to the multiple run editor using storestring (see “Auxiliary routines”, “Passing data between forms”).

The run is controlled by the MM, so that the user can interrupt, and displays can be updated. Each year, the DLL routine “docycle()” is called, which simulates the current year of the forest. That is the main model call.

2.6 Model selection

The model is described by the set of modules, and the values of parameters used in the modules. The user can change the module choices and change the parameter values.

There is a concept introduced here called “swappable function”: each swappable function is simply a name, representing a process in the model that requires a module to fill it. Each module is associated to one, and only one, swappable function, but each swappable function may have one or more modules associated with it (though only one in any given model). There are two “types” of swappable function: “ecological” and “silvicultural”. The distinction is made to split the idea of the two models for the user, since the approach to changing them is completely different. The implementation differences are limited to a label, however, so programming alterations are simplified.

There are a few basic swappable functions, and a module for one of these may call (depend on) other (non-basic) functions. In turn, the modules for the depended functions may depend on other functions. In this way a tree structure is built up (this can be viewed with the “modelview.frm”). Getdeps() (MM) finds these dependencies by asking the DLL for them. The dependencies of a particular module are set when the module is written.

The module choices and parameter values are stored in the DLL (see section 3.4), and are loaded into the MM using the functions: newmodelsettings() (MM), getmodelswfuncs() (MM), getdeps() (MM), GetSWFuncInfo() (DLL), GetChosenModDeps() (DLL), GetModInfo() (DLL) and GetParInfo() (DLL). This happens when the user selects the module choice editor (modchoices.frm) or parameter editor (pared3.frm), or double-clicks on a parameter in the modelview (modelview.frm) window.

When the user makes a change to a module choice, the model tree might change, so newmodelsettings() is called to update this. The newly chosen module choice is set in the DLL using SetModChoice() (DLL), and the user does not have to click “OK”, or “set” for this to happen.

When the user makes a change to a parameter value, nothing is set in the DLL until the OK button is pressed on the parameter editor window. Then the current set of parameter values stored in the MM is uploaded to the DLL using DoCommitParameters() (DLL).

Module sets and parameter sets may be saved to file for later use. The user has to specify a unique label to identify the set, then a new set is created in the MM and all sets are written to file (mods.txt for modules, pars.txt for parameters). The MM stores the group of module (parameter) set names in ModSetName$ (ParSetName$), and the module choices (parameter values) in Modstring$ (Parstring$). The current set, as would be used by a simulation, is stored in the DLL.

2.7 Data input

There are two stages to data input: for trees and for stands. Both stages work in the same way, however, with similar code.

The original design for data i/o was to use ODBC (Open DataBase Connectivity) – a collection of software that presents a common interface to data in databases independent (almost) of the database type. The problems with this approach were that the Microsoft Data Access Components (MDAC) required for this were frequently updated, not all database drivers were included, MDAC versions were not backwards-compatible so some re-coding was sometimes required, testing was difficult because errors would occur for some database types and not others, and last but not least, the data interface was extremely slow. Sometimes it took 15-30 seconds to make a table and enter a few (10) lines of data. This was unacceptable, so a text-format version (either tab-separated variables, “.txt”, or comma separated variables, “.csv”) was written in standard C code. This is extremely fast. The ODBC options are still present through the “advanced” button, but are not recommended for reasons of speed and reliability. Also, it is possible to import the .txt and .csv formats into almost any database with minimal effort, and to view them using a simple text editor.

The MM is used to set the variables required to open a dataset: the database type (DRflag&=4 for text), the database location (DBbaseFile$ is a directory path for text), the table name (DBtable$ is a filename for text). Defaults for these are stored in “mm.ini” and are used as a starting point. These are passed to the DLL and the data is read in using readindata() (MM) and DoReadDBdata() (DLL). A similar DLL function and similar MM variables exist for the stand data.

Any usage of SYMFOR requires data about each and every tree in the plot (currently above 10cm diameter). This can only be input through the MM by using tables. The stand data is not always recorded in table format however, so a “stand wizard” (standwizard*.frm, where * is 0 to 4) is available to enter the same information from prompts. This is available as a button from the stand data entry form, selstanddata.frm.

2.8 Random number issues

SYMFOR has components that use pseudo random numbers – mainly to convert a probability into a definite case. The pseudo random number generator used in the DLL (“rand()”) requires a “seed” number to start it off. After that start, the last random number generated is used as the seed for the next number. Thus, given the same seed, the sequence of random numbers will always be the same. This enables subsequent runs to be different, because the seed is generated by the MM based on the

date and time. It is presented to the user with the tree data input ("seldatasource.frm"), and can be edited at this point. This is extremely useful for debugging, when a problem may occur in one run, but not in the next. If you have the random number seed, it is possible to precisely reproduce the erroneous run. The random number seed for each run is printed to the SYMFOR log file.

2.9 Data output

Data output is to database tables, using similar code to the data input. The user must select what tables to output to, and what data to output to those tables, before the run begins. This is done from the "outputs" menu, or from buttons in the multiple-run instruction editor, or the "run settings options" (startrun.frm) for single runs. An editor window (outputed.frm) allows the editing of existing options, or creation of a new output table.

In creating a new table, the first window to be encountered (DBouted.frm) specifies the type, location and name of the table, and when (during the run) to output the data. The second window (dataouted.frm) allows the user to select which sort of objects they would like data about (livetrees, fallentrees, stand, etc). These forms also use code in "outcode.bas".

These options are stored as variables in the MM module "outcode.bas". There is one set for the original values (O*), and one set for the newly changed values (T*). When editing is finished, the original values are changed to the new values with setdlltolocal() (MM), uploaded to the DLL using commitoutputs() (MM) and DoCommitOuts() (DLL). At the start of each run, the MM asks the DLL to reset any existing output tables using DoResetOuts() (DLL), and create the output table including the options of when to output, using DoSetupDBOuts() (DLL).

2.10 Displays

The code controlling the displays contains some of the most confusing sections within SYMFOR. This is because of the interface to complicated controls, and to the data needed to fill the displays. Getting this code to work is a long process of trial and error, because the documentation of the controls is often poor, and there are many exceptional circumstances that have to be caught. When a display works, it is generally best to leave it, and the development of a new display should not be undertaken lightly.

The 6 displays are: stand table (or summary table, "sumtab.frm"), frequency distribution (or object histogram, "frequency.frm"), profile view (or lollipop view, "prof.frm"), plan view (or map, "newmap.frm"), individual information table ("indivtab.frm") and time series plotter of run results ("plotter.frm"). These are implemented very differently to each other and will be described individually.

The displays are updated when they are opened, and every integer 10 years of the run thereafter. The function updatedisps() (MM) is used for this. The displays are updated every year of a multiple run. There is no particular reason for this, but it is not intended that displays would be used with a multiple run.

2.10.1 Stand Table

The stand table shows summary information about the current state of the forest. Each element in the table is called a cell. Double clicking on the cell allows the user to edit the contents, with "cellcont.frm". These forms are supported by code in "sumtabcode.frm". The contents may be of different types: empty, text or a numeric value. The numeric values are statistics (sum, mean or n) about an object's variable. The object, variable, subset and statistic are selected by the user. The selections are converted to a string with a format, passed back to sumtab.frm using

storestring()/retrievestring() (see “Auxiliary routines”, “Passing data between forms”) and interpreted using atomise() (see “Auxiliary routines”). The routine, changevaluecontents() (MM), interprets the instruction. SumtabEvalCell() (MM) uses GetDataDLL() (DLL) to retrieve data from the DLL and calculate the relevant statistic to fill the cell of the table. SumtabEvalCell() is used for each cell in turn each time the stand table is opened, when the data is updated (sumtabnewstate() (MM) is the function that is used to update the table), or when the definition of the table contents is changed.

Cellcont.frm uses a very similar construction to that found in other data subset selection forms. A combo-box is used to select the forest object, and another selects the variable of the object that is of interest. A statistic is selected, and a further window, “binsel.frm”, is used to select up to 4 sub-setting criteria based on values of any variable of the object in question. The criteria are put into string form and passed back to cellcont.frm using storestring()/retrievestring() (see “Auxiliary routines”, “Passing data between forms”).

2.10.2 Frequency distribution

This shows information about the number of individual objects as a function of some variable. It is possible to change the bin-width, the object, the variable and sub-criteria (e.g. trees with DBH>50cm). The frequency.frm is supported by freqcode.bas, and uses an “MSChart” control to do the actual drawing. Double clicking the form brings up the data selection editor, freqsettings.frm. This is similar to cellcont.frm (see Stand Table section). The code for drawing the chart and manipulating the data is in freqcode.bas. The default settings are read from “mm.ini” by freqsetinit() (MM). Freqnewsetting() (MM) interprets the setting instruction and stores the information as local variables. Freqredraw() (MM) is used to get the required data from the DLL (with GetDataDLL() (DLL)), calculate the relevant quantities for the chart, set up the chart as required (with SetChartOptions() (MM)) and fill the chart with data. It is called each time the frequency distribution is opened, when the data is updated, or when the definition of the table contents is changed.

There is an unusual feature of the MSChart control to do with mouse clicks. Once the chart has been selected, with a single click, often two clicks are required to move the focus to another window. I have not found documentation on this or a way around it. It is not a big problem, but makes operation slightly less smooth.

2.10.3 Profile view

This shows a graphic representation of the stem, crown and position of each tree in the plot. Different species are shown in different colour, and the size of each tree is related to the size in the data. It was originally intended to act as a view of the forest to demonstrate what the model was doing in terms that foresters would understand. Unfortunately it was promoted to the detriment of other aspects of the model, and while it is now maintained, it is sidelined. It is not possible to change this display by double-clicking. It may be resized, but that is all.

Prof.frm is supported by profcode.bas. The default settings are read from “mm.ini”, interpreted and the data is retrieved from the DLL using GetDataDLL() (DLL) by profreadstate() (MM). The view is updated using profnewstate() (MM) each time the profile view is opened, when the data is updated. Profredraw() (MM) redraws the picture using the data currently loaded.

2.10.4 Plan view

The plan view shows a schematic map of the modelled forest. A set of checkboxes allow the user to switch on and off the map features. It is possible to view: live trees (as filled green circles), felledtrees or fallentrees (as lines from the base of the tree in the "falldirection"), areas of damage (as hollow red regions) and skidtrails (as filled yellow rectangles). The border of the plot (as a black line) and the effects of "wrapping" to reduce edge effects can be shown. The map can be re-sized, but it is not possible to zoom in or out. Double-clicking has no effect – the checkboxes control all the display alterations.

All the code relevant to the map is in newmap.frm. The data used for the map are refreshed by calling GetDataForMap() (MM), which uses multiple calls to GetDataDLL() (DLL) for the variables for each object. All data are stored in local variables. The map display itself is updated with ReDrawMap() (MM). This tests the checkbox values, and draws the objects accordingly. The drawing tools are lines and circles – no controls are used. Separate routines exist to handle the co-ordinates if wrapping is required (lineinplot(), pointinplot(), plotwrap() and plotwrap2() (MM)).

2.10.5 Individual information table

This table shows the values of all variables for all instances of any type of forest object. Indivtab.frm contains all the relevant code. Each time the object combobox is clicked, the newobj() (MM) routine loads the value of each variable for each instance of that object using GetDataDLL() (DLL). The form_load() routine does this by setting the combo1.listindex to 0. The table is drawn using a "grid" control.

The individual information table is the only table not updated every 10 years during a single run, due to the length of time required for this.

2.10.6 Time-series plotter and Run Results

This display uses an MSChart control (see the "Frequency distribution" section) to plot a statistic describing the population of an object (such as mean DBH of livetrees) as a function of time.

In order that many objects/statistics can be observed during the same run, the statistics are calculated outside the plotter itself. They are known as "Run Results" and their details can be edited using the "Run Results" menu from the main SYMFOR window. They are stored in an independent file, by default called "results.csv". The user can change the file or delete the current file using options on the menu. The other option, "Change run settings", allows the user to alter the variables that are calculated each year. The form used for this is "resed.frm", supported by "resultscode.bas". The previous variables are stored in "mm.ini", which is updated if the selection is changed. The user creates an instruction by selecting an object, variable, statistic and optional sub-selection, and types a label for that variable that will appear on the graph. The "add" button moves the instruction to the listbox, meaning that it will be calculated each year of any subsequent run. Instructions may be deleted. Too many (>10?) instructions will cause the program to run significantly slower due to the increased number of calculations. In each annual run loop, runvarcalcvalues() (MM) is called to calculate the run results which are stored in the array runvarvalues!(instr&, iyear&). At the end of each successful run, the run results are written to file with runvarCommitValues() (MM).

"Plotter.frm" shows the currently selected run results variable as a function of time from the start of the run. If there is no current run, it will prompt the user for a filename for old run results, and read the sets of run results from that file, presenting the user with a choice of runs. For each run, the

same as for double-clicking on the chart area, the choice of run results are available to plot from a combo box.

2.11 Global variables for control

Global variables are, in general, to be avoided. However, when there are some variables used throughout a program it does not make sense to pass them from one block of code to another.

The global variables used in SYMFOR are identified at the start of modman.bas, and are commented in several sections. Many are constants that describe common array dimensions or are labels. Others are simply variables that are used in many parts of the code.

2.12 Auxiliary routines

Routines that are not specific to a particular piece of code are separated into a *.bas file. Described here are three functions that were written for SYMFOR, and two that are standard Microsoft functions.

2.12.1 GetPrivateProfileString()/WritePrivateProfileString()

To write and read meta-data, instructions, initialisation information, etc, reliably to and from files, it is useful to have a pair of standard functions for the job. The form of these two functions can be seen in the code, and the resulting file structure is used in mm.ini, and copied in pars.txt, mods.txt. One problem that has occurred with these functions is a difference of implementation between Windows 98 and Windows NT. The Windows 98 version will not read or write after 64KB, although the Windows NT version will. To avoid this problem for large files, the same format is created manually (see registerparameters() (MM)) and interpreted manually for files where this may be a problem (i.e., not mm.ini).

2.12.2 Passing data between forms

When a form is closed, control returns to the form that opened it (except if the MDIForm is closed, when the whole program ends). Often, a temporary form is used to allow the user to set some information or value. This information is passed back to the previous form by using two routines (in ”stringfns.bas”). The temporary form uses “storestring” to store a string globally with an identifying label. The previous form then uses “retrievestring” with the identifying label to retrieve the string for use.

2.12.3 Atomise()

Atomise is used to break up strings that are made up of several sections, separated by commas, colons, square brackets and so on. This is often necessary after passing data between forms. Alternatives are atomise2() and atomise3(), which are slight variations that treat some characters differently, such as spaces.

2.13 Communication with the DLL.

The DLL is compiled separately from the MM, into a separate file (sf2000.dll). The MM must pass information to and from it. The DLL is a Dynamic Linked Library, which means it has functions that are available to be used by outside programs. The way this is done in the DLL is described later, but the MM simply has to be told where the library is, what the functions are called and what the rules are for calling them. This is done at the top of modman.bas. The functions can then be used just as within VB.

Some functions pass a very large number of variables – primarily when data is being transferred. This is done “by reference” (VB-speak), or “using pointers” (C++-speak): instead of the function

returning the value of a variable, it returns the address in memory of where that variable is stored. If the data are stored sequentially in memory, then all that is required is the address of the first one. Using the first address, then looking at the next memory location, and so on, all the data from the DLL can be read by the MM implicitly. As long as both the DLL and the MM know that they should pass/receive addresses, rather than values, the system works well. VB knows how data should be interpreted because of the function declaration in modman.bas – by default the data are passed "by ref", but usually in SYMFOR it is explicitly stated to pass "by val". Usually it is also necessary to say how many values are stored after that first address, otherwise the MM wouldn't know when to stop looking at the next address for its next value.

"Integer"s in C++ (Microsoft VC++ v.5 and 6) are 32-bit, which means they occupy 4 bytes (=32 bits) of memory. "Integer"s in VB (Microsoft VB v.5) are 16 bit, but there is a different data type, called "long", which is identical to the VC++ "integer", and is used instead. "Integer"s in VB are almost never used.

2.14 Summary of MM files

Most of these files are described in the preceding sections, however it may be useful to describe the typical usage of the MM functions during SYMFOR operation.

General/start-up:

1. modman2: "modman2.frm", the SYMFOR MDI form.
2. title: "title.frm", the title screen when SYMFOR loads.
3. modman: "modman.bas", general code, global variables, DLL function declaration.

Auxiliary/multi-use:

4. binsel: "binsel.frm", used to create a sub-set of the data for displays and run results.
5. stringfns: "stringfns.bas", functions for manipulating string variables.
6. logcode: "logcode.bas", the code for creating a log-file to help debugging.
7. dview: "dview.frm", a viewer to display a text file such as "about.txt".

Model editing:

8. modchoices: "modchoices.frm", used to select appropriate modules for a model.
9. savemodset: "savemodset.frm", used to save a set of modules.
10. pared3: "pared3.frm", used to edit the values of parameters in a model.
11. saveparset: "saveparset.frm", used to save a set of parameter values.
12. disaggopts: "disaggopts.frm", displays parameters with more than one value.
13. modelview: "modelview.frm", displays a model schematically.
14. loadstandards: "loadstandards.frm", used to load a standard module and parameter set from a fixed location.
15. loadstandardswait: "loadstandardswait.frm", displayed while loading standard module and parameter sets.

Data input:

16. seldatasource: "seldatasource.frm", prompts for a tree data input table and random number seed.
17. selstanddata: "selstanddata.frm", prompts for a stand data input table.
18. standwizard0,1,2,3,4: "standwizard0,1,2,3,4.frm", the stand data input wizard sequence of forms.

Data output:

19. outputed: "outputed.frm", used to edit the selection of output data tables.

20. DBouted: “DBouted.frm”, used to edit the type and location of a particular output table and when the data should be output.
21. dataouted: “dataouted.frm”, used to edit the variables output to a particular table.
22. outcode: “outcode.bas”, general code relating to data output

Single run:
23. runcont: “runcont.frm”, the single run control window.
24. startrun: “startrun.frm”, prompts for final changes to run settings before the single run begins.

Multiple run:
25. multirunf: “multirunf.frm”, the multiple run control window.
26. mruninstr: “mruninstr.frm”, the multiple run instruction editor.
27. saverunset: “saverunset.frm”, used to save a set of multiple run instructions.

Displays:
28. sumtab: “sumtab.frm”, the summary, or stand, table display.
29. cellcont: “cellcont.frm”, the stand table contents editor.
30. sumtabcode: “sumtabcode.bas”, code supporting the stand table display.
31. frequency: “frequency.frm”, the frequency, or histogram, display.
32. freqsettings: “freqsettings.frm”, used to edit the variable plotted by the frequency display.
33. freqcode: “freqcode.bas”, code supporting the frequency display.
34. prof: “prof.frm”, the profile view display.
35. profset: “profset.frm”, the vestigial profile view display editor.
36. profcode: “profcode.bas”, code supporting the profile display.
37. map: “map.frm”, the schematic plan view of the forest objects.
38. indivtab: “indivtab.frm”, the table of data for each forest object type.
39. plotter: “plotter.frm”, the time-series plotter of run results.
40. plotterset: “plotterset.frm”, used to edit the time series plotter display.
41. plotcode: “plotcode.bas”, code supporting the time-series plotter.

Run Results:
42. resed: “resed.frm”, used to edit the variables calculated as run results.
43. resfileed: “resfileed.frm”, used to change the file where run results are stored.
44. resultscode: “resultscode.bas”, code supporting the run results display.

3 The DLL

This section explains what the basic functions of the DLL are, and how they are achieved using VC++ code.

There are three fundamental requirements of the DLL. It must:
- house and operate the models
- house the data
- communicate with the MM.

Communication with the MM is described with the functions that the DLL provides, in sections 3.3 and later. The first two sections introduce DLLs within C++ and give some tips on how to manage the files.

3.1 VC++ usage

C++ is a comprehensive, low-level, object-oriented language. As such it produces fast-running software, but there is often a compromise between the speed of execution and the flexibility and clarity of the code. Speed was not viewed as a priority for SYMFOR, but flexibility was. The code has been written in a formal, general way such that it should be relatively simple to edit.

C++ is a compiled language. The code cannot be run line-by-line as for interpreted languages. For the code to be intelligible to the computer, it must be compiled and then linked (in Microsoft VC++ this whole process is known as “building”). Compiling works file-by-file, each file being converted to assembly language that is directly intelligible to the computer. The compiled files are then linked together, and references to external functions are verified, producing a single output file (in this case “sf2000.dll”). To debug C++ code, additional information linking the assembly language statements to the C++ statements is required. This is produced using particular options during compilation. To make the process easier, there are two “build configurations” in Microsoft VC++: “debug” and “release”. These are only labels that represent a set of compilation and linker options that may be set using “Project settings”. “Release” is used when debugging information is not required, and instead producing fast execution is the priority.

C++ (or C) code is structured such that function and object attribute declarations are made in a header file (“*.h”). C++ (“*.cpp”) files contain the function definitions, which often include calls to other functions. For the compiler to know whether such a function call is made correctly, and for the linker to know where that function definition is, there is a list of the header files containing all the declarations of the functions used at the start of each C++ file.

“Object orientation” means that classes of objects are defined to represent the data structures or functional operation of a system. In this case, object orientation has been used to represent forest structures. A basic concept of a “tree” is represented by a class. It has attributes that all trees have: DBH (a variable), volume (a function), etc. There are two classes, “livetree”s and “deadtree”s which are derived from the “tree” class. They inherit all the attributes of “tree”s, but have some additional ones specific to themselves: “livetree”s have a growth function; “deadtrees” have an “ageatdeath” variable. There are different classes of “deadtree” reflecting categories of the way or reason a tree died: naturally, as a result of damage, as a result of logging or from silvicultural thinning processes. When a tree dies in the model, the livetree object is used to create a suitable deadtree object, and the livetree object is then deleted. In OO terminology, this whole structure is called a “class hierarchy”, and uses “inheritance”. Each tree recorded in the data is stored as an object – an instance of the class. There may be many objects of class “tree”, but all probably have different attributes. Similarly there are classes representing the stand (one instance only), skidtrails and gridsquares. More abstract classes are used to manipulate shapes – usually areas of damage to the forest. A skidtrail object has a “shape” attribute, e.g. Microsoft’s VC++ has a “class-view” that shows each class (or “globals”, if the function or variable is not in a class”), and can optionally show the contents of a class – the attributes. This can be very useful for exploring the project to determine structure.

C++ uses dynamic memory allocation, meaning that it is not necessary to reserve in advance a block of memory for the program to use while it is running. All the variables declared outside class definitions are allocated memory for the whole run-time, but everything else only uses memory when it is required. A class is simply a definition, and does have data or require memory. When an object of a particular class is created (using “new”, for a tree called “joe”, eg, “joe = new livetree;”), the memory needed for one instance of that class is used and reserved. The object may be referred to by name or by location in memory (a “pointer” is an address in memory). As more objects are

added, more memory is used. When an object is removed from the model, the “delete” statement is used to free the memory that had been used by the object. In this way, memory allocation is dynamic. (“New” and “delete” are the C++ equivalents of the “malloc” suite of commands used in C, but are much easier to use.)

3.1.1 Concepts in C++

There are two concepts in C++ that are not necessarily obvious or common to high level languages. These are pointers, and objects.

Pointers are records whose values are interpreted as addresses in memory at which data (of some sort) is held. Two pieces of information are necessary: the 32-bit memory address of the byte at the start of the data record, and the length of the data record (so the program knows where to stop reading the data). The following table may help understand some usage of pointers.

int myint, newint;	declaration of two integer variables in memory, with name “myint” and “newint”
int* pmyint;	declaration of a pointer-to-integer variable in memory, with name “pmyint”
pmyint = (int*) &myint;	assignment of the address (&) of the integer “myint” to the pointer “pmyint”. The type-conversion (int*) forces the address of “myint” to be interpreted as an integer pointer. C++ will do this anyway, but including it reduces compiler warnings and leads to clearer code.
newint = *pmyint;	Assigns “the value found at the address pointed to by pmyint” to newint. Placing the * after the variable is a multiplying operator, but placing it before indicates that the following variable is a pointer, and to use the value it points to.

Objects are treated the same way that other variables are. A truly object oriented language has even its fundamental variable types, such as integers and floats, internally described as objects. C++ does not (it is derived from C, and the fundamental variable types are preserved). Object attributes are accessed by one of two methods: if “mylivetree” is a livetree object, the value of its “x” attribute may be obtained using: “mylivetree.x”. If “pmylivetree” is a pointer to a livetree object, the value of x for the livetree object may be obtained by: “pmylivetree->x”. These indirections may be nested, such that if one object contains another object (usually of a different class), the internal object’s attributes may be accessed using the notation: “parentobj.childobj.childattribute”, or “parentobj->pchildobj.childattribute”, or “parentobj.childobj->pchildattribute” (where the “p” is used to indicate that the variable is a pointer).

3.2 What is a DLL?

Dynamic Linked Libraries, or DLLs, are Microsoft’s library files. As I understand it, the difference between DLLs and static linked libraries is that DLLs may be linked to by more than one program at once. There is little advantage to having the SYMFOR library dynamic, however they are better documented and work just as well.

C++ programs must have a function called “main()”, otherwise the compiler/linker doesn’t know at what point in the code the program should start. DLLs, however, don’t “start” as such – they just sit there until one of the library functions is called by an external program. So there is no “main” function. Instead, some of the functions in the DLL are labelled as being available to external programs. This is done by prefixing the function definition with

__declspec(dllexport)
and
__stdcall
and including the function in a list in the "*.def" file. In addition, the function declaration must be prefixed with
extern "C".
There are a lot of functions in the DLL, but only 24 are provided as entry-points to the DLL.

The code is compiled to form a DLL by setting the correct compiler and linker options. The compiler options used are:
/nologo /MD /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /D "_WINDLL" /D "_AFXDLL" /D "_USRDLL" /Fp"Release/sf99.pch" /YX /Fo"Release/" /Fd"Release/" /FD /c
and the linker options are:
/nologo /subsystem:windows /dll /incremental:no /pdb:"Release/sf2000.pdb" /machine:I386 /def:".\sf99.def" /out:"c:\program files\symfor\sf2000.dll" /implib:"Release/sf2000.lib"
Not all of these are specific to DLLs, but some are. It works, and I haven't bothered to find out which options do what.

All .cpp files have a corresponding .h file with the same name. The main code (and all the entry points) are in "sf99.cpp". Other global functions (global means that they are not included in a class definition) are in "treatment.cpp". The class definitions each have a file to themselves, of the same name, and comprise the rest of the DLL.

3.3 Introduction to the SYMFOR DLL

The SYMFOR DLL is most easily approached using the "class view" in Microsoft's VC++ interface. There are a large section of functions and variables classed as "globals". These are functions and variables that do not belong to particular object types, and are stored in sf99.h (.cpp) and treatment.h (.cpp). Most of the global variables are parameters, used in the model. The parameters themselves are hardcoded, but have their values set at run-time (see section "Model storage, structures, editing"). All the DLL entry points (functions) are in "sf99.cpp", but are also necessarily listed in "sf99.def".

Where practical, dynamically created and deleted objects have been used for storage instead of statically allocated arrays. However, some arrays are needed, and the size of these are stored as global constants. Other limits are also stored as constants, and these are all in "sfconsts.h". Some of these constants must be the same as the constants defined in the MM (modman.bas) for the SYMFOR system to function correctly, and these are labelled in "sfconsts.h".

3.4 Auxiliary classes and structures

Described here are the classes/structs: objectcore, climate, dbhlist, shape, symedge, symvertex. These are essential to DLL operation, but do not come under any particular category.

3.4.1 Objectcore class

Objectcore is the class of objects (objectcore.h) that represent groups of forest objects. It is necessary to group the livetrees, for example, together so that it is possible to operate on them all, or to search among them for a particular one. A linked list is an unnecessarily complicated structure in this context, and would often cause excessive computation time while searching for a particular individual. Instead, objectcore objects contain an array of pointers, and a record of the number of individuals that actually exist in the array. In this way, each pointer can be accessed in turn, or a particular one accessed by its array index. An addition to the group may be made as long as the

array bounds are not exceeded, and an individual may be removed by shifting all the following pointers up one in the array. An objectcore instance is created for each group of forest objects in sf99.cpp. The forest objects are accessed by reference, using a line of code such as:
plive = (livetree*) livetreegroup.ptrlist[itree];
where the livetree* indicates it is a pointer to a livetree object, the livetreegroup is an objectcore object representing the group of livetrees, and ptrlist[] is the array of pointers to livetree objects. From this point on, plive can be used to access the livetree object with code such as:
x = plive->x;

3.4.2 Climate class

The climate class of objects (climate.h) is not used in any implemented SYMFOR models at present. However, it could be used with future models with modification. The basic class is included to allow ease of integration of future models.

3.4.3 Dbhlist data structure

The dbhlist data structure (treatment.h) is a linked list structure. Each dbhlist variable holds a pointer to a livetree, an integer indicating its position in the (ordered) list, and a pointer to the previous and next dbhlist variables in the list. The starting variable in the list has a .previous attribute of “NULL”, as does the .next attribute of the last variable in the list. It is used for sorting the trees flagged for logging in order of decreasing size, but could be used for any ordered list of livetrees.

3.4.4 Shape class

The shape class of objects is used with the data structures “symedge” and “symvertex” (shape.h) to represent circles, rectangles and triangles, and their position with respect to particular points in 2-dimensional space. They are used primarily for simulating the areas of damage caused by falling trees and by logging operations, but also for competition indices where trees within a given radius are considered. The functions that apply to shape objects have been adapted to wrap around the plot edges – if a shape overlaps the plot edge, it will effectively re-appear on the other side of the plot. This theoretically gives the plot the shape of a torus (ring-doughnut). Most operations are to determine whether a point is inside or outside of a shape object.

3.5 Data storage, structures

Described here are the classes: Cforest, tree, livetree, deadtree, falltree, fallentree, felledtree, smashedtree, killedtree, stand, skidtrail and gridsquare. The Cforest class is provided because the MM variable, “runnumber”, is required for data output for any forest object. Consequently it is at the top of the hierarchical structure, and all other forest objects are derived from it. It provides only one variable, runnumber, and nothing else.

3.5.1 Tree class hierarchy

The tree class hierarchy is shown in figure 1.

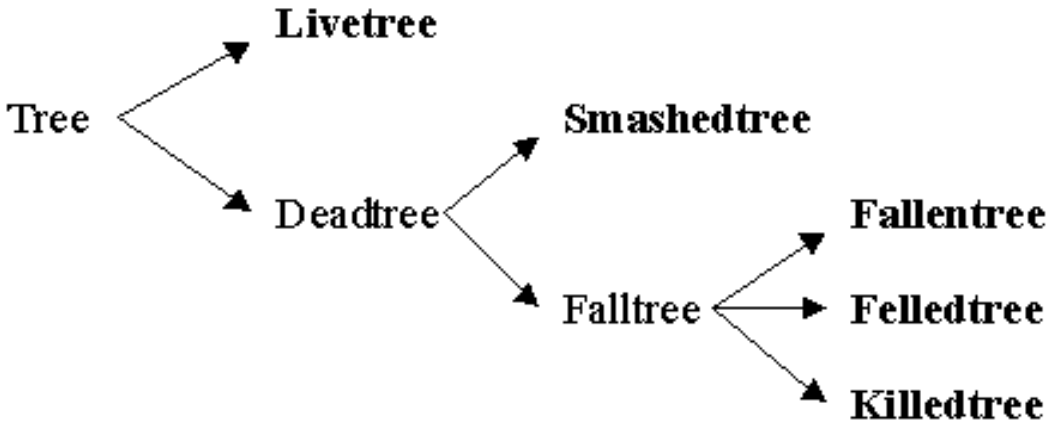

Figure 1: The class hierarchy of tree objects. The classes that actually have instances are shown in bold.

The details of each individual class can be examined in the code (object".h"). Described here are some aspects of the classes that may not be immediately obvious.

Tree objects have the size and descriptive attributes common to all trees, live or dead. Where quantities are not fundamental (such as height, volume), they are calculated from fundamental quantities (such as DBH, species) using a function. There are two model-specific fundamental attributes: planted, which has a value 1 if the tree was planted, and 0 if it is natural, and quality. Quality represents the suitability of the stem for logging from the perspective of straightness and non-hollowness. It takes a value between 0 and 1, assigned randomly when the tree is created, and may be used in the logging modules to decide whether or not to fell.

Livetree objects have inherited the tree attributes, and have some others specific to livetrees. Growthbias is a random number generated when the tree is created, and is used in some growth functions. The growth function (swappable function) is dbhincr(), and this is used to select between growth modules. It uses the standard module choices method controlled by the user through the MM, described later. Most growth modules are called dbhincr*(), but a process-based one is called GEFF() and uses CalcProcessData() to initialise the livetree attributes specific to the process-based model: heartC, heartN, sapC, sapN, leafC, leafN, storeC, storeN, rootC, rootN, hbc, harea and LAD. The other growth modules do not use these attributes. The functions stom() and phot() are also specific to the process-based module. There are two more swappable functions that are part of the livetree class. Compindex() chooses between the modules shadeindex(), nplot(), obaplot(), nrange() and obarange() in evaluating a numerical competition index for a particular tree. Often these functions use a subset of the livetrees available, and these are defined with neighbour(). Only one module exists for defining neighbours at present: neighbour1(). The other livetree function, "livetree()", is a constructor function. It is called automatically when a new livetree is made, and is used to set initial values to the attributes. When a livetree is created, "checktreeposn()" checks that it is not positioned such that it overlaps another tree, or takes appropriate action.

Deadtrees do not exist as data in the model, but their attributes are inherited by other classes. They inherit all the attributes of trees, plus the variables ageatdeath and yeardied. Yeardied represents the simulation year (starting at 0) that the tree died.

Smashedtree objects are trees that died as a result of damage, either by another tree falling, or by felling operations – skidtrail clearance, etc. They do not fall over in the simulation. They have two constructor functions that are distinguished by the number of arguments they take. There is a basic one that simply returns “true”. The main constructor copies the attributes from the livetree that is dying, and adds one to the group of smashedtrees.

Falltree objects do not exist as data in the model, but their attributes are inherited by other classes. They inherit the attributes of deadtrees. They have a falldirection attribute, which is a bearing, in radians, measured from North to the East. Some trees do not fall (even if they’re called falltrees) and they have a falldirection of –1.0. When a tree falls, it creates an area of damage mapped on to the ground, the “fallshape”. This area is made using the kiteshape() swappable function, with modules kiteshape1() or kiteshape2(). The fallingtree modules are used to simulate the tree falling and damage that ensues. The status attribute is used to indicate if the falltree is active or inactive, meaning whether it should appear on the map view of the MM. To save confusing the view, felltrees are not shown if they are older than a certain age, indicated by parameter “logpersist”.

Fallentree objects represent trees that died of natural causes, other than damage from other tree-fall, irrespective of whether they actually fall over. They have two constructor functions that are distinguished by the number of arguments they take. There is a basic one that simply returns “true”. The main constructor copies the attributes from the livetree that is dying, and adds one to the group of fallentrees.

Felledtree objects represents trees that died because they were felled for timber. The process of creating them involves analogies of the real process of felling a tree. There is damage due to the tree falling (falltree::kiteshape()), due to the tractor manoevering to attach the log (skidprepdamage()), due to the log being pulled into line with the skidtrail (dragdamage()) and finally due to the production of the skidtrail itself (see the skidtrail section). Felledtrees have shape attributes to represent the areas of dragdamage (triangular) and skidprepdamage (circular). They may be displayed on the plan view for a plot that has undergone simulated logging. Felledtrees have two constructor functions that are distinguished by the number of arguments they take. There is a basic one that simply returns “true”. The main constructor copies the attributes from the livetree that is felled, and adds one to the group of felledtrees. The sum of the volume of all felledtrees is the timber harvest.

Killedtree objects are those that were killed by silvicultural thinning. They have three constructor functions that are distinguished by the number of arguments they take. There are two basic ones that simply return “true”. The main constructor copies the attributes from the livetree that is dying, and adds one to the group of killedtrees. The attribute “thinflag” is 0 by default, 1 if the tree was thinned by poisoning and 2 if it was thinned by felling. Poisoned killedtrees do not fall over, but felled ones do.

3.5.2 The stand class

The stand class exists purely to facilitate the output of stand data using the output routines, and to follow convention for forest objects. There can only be one instance of a stand at one time in the model (or, at least, only the first stand in the group is ever considered). The stand contains data relating to the whole plot of forest represented by the tree data input file. This includes the x and y bounds to the plot, the age (starts at zero at the beginning of a simulation), the number of years since the plot was logged and the relative intensity of logging (yslscale). There are some functions that can provide data for output, such as area(), ntrees(), sumskidarea(), sumba() and

sumskidlength(). Cumtimber is a running total of the harvested volume of timber in a simulation. The attribute iharvest is used to indicate whether the next harvest should be a primary harvest or secondary harvest for two-tiered logging strategies ("advanced" logging).

3.5.3 Skidtrails

The skidtrail class represents both conventional skidtrails, used to extract logs from the forest, and cleared strips that are used for managed replanting. The attribute "plantedsps" is –1 for skidtrails in which no planting takes place, otherwise it takes the value of the species group to be replanted. The "tended" attribute indicates the number of years after clearing that the strip is to be tended (with undergrowth removed), and "plantedutl" indicates the utilisation group of the replanted trees ("replantstrips()").

Basic skidtrails form a rectangle ("skidshape") either side of a straight line from the base of a felledtree to the "accesspoint", where the logs leave the plot. The age of the skidtrail is monitored because it affects recruitment of new trees. A proportion (usually 1.0, all) of existing trees in the skidtrail area below a certain diameter are removed when the skidtrail is created ("skidtraildamage()"). Advanced skidtrail modules form branched structures, where a skidtrail runs to another skidtrail, instead of to the accesspoint. In these cases, the pre-existing skidtrail is called the "parent", and the "pparent" attribute is a pointer to it. Skidtrails are only candidates for becoming parent skidtrails if they are still open areas of forest. A skidtrail will often split a parent into an old section, that is not re-opened, and a newly-skidded section. The newly skidded section becomes a new skidtrail that is split from the parent, using "splitparentskid()". Whether a skidtrail is still an open area of forest is defined by their "status", which is changed by calc_ages() after "skidpersist" (a parameter) years. Additional advanced skidtrails account for cutting the corners at the angle where two skidtrails join; these are triangular skidtrails.

There are 5 skidtrail constructor functions altogether, accounting for one basic function and the four types of skidtrail mentioned above: standard, rectangular skidtrails; rectangular skidtrails that were formed from re-used (parent) skidtrails; triangular corner skidtrails; and cleared strips used for replanting. They have the same set of attributes, but the attributes have different values for each type.

3.5.4 Gridsquares

Gridsquares are the cells of a notional division of the plot into equal sized rectangles. They are used primarily for the recruitment process: new trees crossing the size threshold for trees represented in the plot. The "newtrees()" swappable function is used with "newtrees*()" modules to represent recruitment. The other gridsquares functions accompany various newtrees functions. "Estphase" represents the number of years that a gridsquare has been suitable for the establishment of seedlings, and for a given year, "estphaseinc()" calculates how that should be changed. "lai()" estimates the leaf area index of a gridsquare using the DBH of all the trees within and around the gridsquare, and using the "foliage()" function. "adjgridsquares()" finds pointers to the gridsquares adjacent to the one in question. "damageintensity()" evaluates an index of damage for a gridsquare, based on the fractional area of the gridsquare that is covered by skidtrails. "avcompind()" estimate the average competition index for a hypothetical 10cm tree in a gridsquare, and "growthrate()" calculates the growth rate of a hypothetical tree in the gridsquare.

The fundamental (rather than functional) attributes of gridsquares relate to the position and size of the gridsquare: col, row, xcentre, ycentre, xwidth and ywidth.

3.6 Model storage, structures, editing

Described here are the classes: swfunc, module, parameter and their implementation.

These classes represent the abstract ideas of swappable functions, modules and parameters, respectively. They were introduced to allow some formality and structure to the process of editing module choices and parameters through the MM. They may seem unnecessarily complicated at first glance, but are low-maintenance and simple to manipulate when necessary. Figure 2 shows a schematic diagram of the swfunc class, expanding the module and parameter classes to show the internal structure.

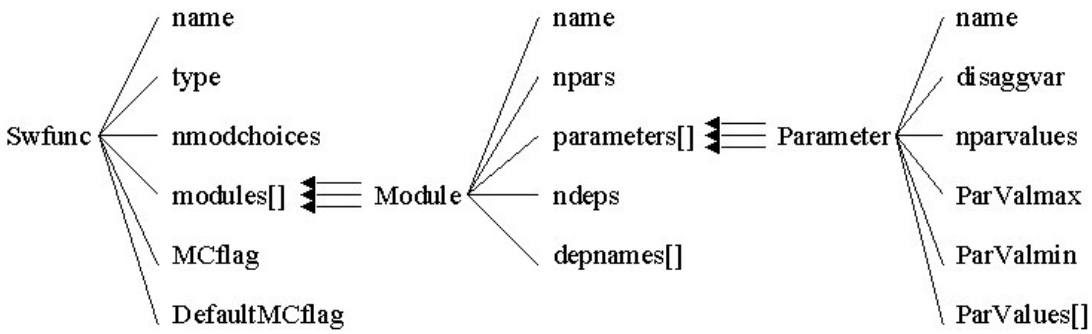

Figure 2: Showing the swfunc, module and parameter classes, and their relationships. Only swfunc objects are declared explicitly in the code, because they contain module objects, which in turn contain parameter objects. Also, the idea of a parameter without a module, or a module without a swappable function, is not conceptually valid.

The swfunc class of objects represent swappable functions. Each model task that can be executed in a number of ways can be thought of as a swappable function. Each possible way that it can be executed, or maybe only some of the ways, are implemented as modules. For example, the livetree swappable function for growth, “dbhincr”. There are many possible ways of calculating the annual growth of a tree, and some of them are represented by modules. Each of those may be thought of as a different model of the way growth behaves. But however the module behaves, it is necessary because it is called by the basic model annual loop (“DoCycle()”, see “Model Implementation”) and so there must be one module, and only one, that is used for a particular swappable function. An swfunc object has an array of module objects, representing the different module choices. The other attributes of an swfunc are: “name”, “MCflag” (the index of the chosen module for this swappable function), “DefaultMCflag”, “nmodchoices” (the number of modules kept in the array) and “type”. “type” is used to differentiate between swappable functions that form part of the ecological model and those from the management options model, for the MM. The one function, “fill()” is used to initialise the swfunc object with particular values.

The module class purpose will be largely apparent from the preceding paragraph. Modules also have dependencies – things they depend on. For example, some modules of growth will use a competition index (“compind”) swappable function, and that is listed in its dependencies so that the MM can give the user suitable options. The “ndeps” and “depnames” attributes describe the number and names of the dependency swappable functions. The module may also have parameters. The parameters are variables that are used in an algorithm or equation in the module – they may be single valued, or arrays. Each module has an attribute, “npars”, indicating the number of parameters that the module uses, and “parameters[]”, an array of parameter objects. There are 6 “fill()” functions, each of which will initialise the module, but they allow for between 0 and 5 dependencies

to be set. Only one fill function will be used for any one module. Each module also has a “name” attribute.

The parameter class exists to store the name, values and disaggregation of a parameter in a module. The parameter may be disaggregated by species group, utilisation group, or harvest. The reason for the latter is to allow some harvesting modules to differ only by a parameter value in alternating harvests. The ecological model parameters typically vary by species group, and management options model parameters typically vary by utilisation group. The parameter class also has attributes to set the maximum and minimum values allowable for a parameter. All data, including default parameter values, are set using the “fill()” function.

3.6.1 Implementation.

A global array of swfunc objects, “swfuncs[]”, is used to access any swfunc, module or parameter. A set of global variables is used to store the parameter values for use by the model during execution. The following section explains how the system copes with changing these values following commands from the MM.

The array is indexed using global constants (from “sfconsts.h”) reflecting the name of the swappable function. Within this, the array of modules, “swfuncs[].modules[]” is indexed by number. Similarly, within each module, the array of parameters, “swfuncs[].modules[].parameters[]” is indexed by number, though the name of the parameter is usually used to check which is which.

The DLL entry point, “DoSfDLLinit()” (sf99.cpp), initialises the swfunc, module and parameter variables in the swfuncs[] array, using the “fill()” function. This is where the names of the swfuncs, modules and parameters are hardcoded. To add a new module it is necessary to edit this section. This section is used to initialise the DLL, before any data or parameter values are stored. The first time the DLL is called, the global variables are created, including the objectcore objects for referencing data about trees, etc.

The choice of which module to use for a given swappable function, and the values of the parameters to be used (see the section “Model Selection” in the MM description), are set using the DLL entry points SetModChoice() and SetParValues(). The former sets “swfuncs[].MCflag” to indicate the module choice (a number, to be used as the index to the “swfuncs[].modules[]” array). The latter sets the values in the array “swfuncs[].modules[].parameters[].ParValues[]” – these are the actual values (but not the variables…) that will be used by the model when it executes.

The values stored in the parameter objects are not used directly by the model as it executes. Before execution, and after SetParValues() is last called, a second DLL entry point, “DoCommitParameters()” (sf99.cpp) is used. This sets the global variables representing each parameter to the value stored in the corresponding parameter objects. This was done for purposes of clarity of model code and speed of execution.

3.7 Model implementation

Described here is Docycle() and the options/flow that follows.

DoCycle() (sf99.cpp) is the function that performs the simulation for one year. It is called each year of simulation by the MM (“runcont.frm” or “multirunf.frm”). In itself it is a simple routine, comprising 9 sequential function calls. If any call returns “false”, the DoCycle is terminated and itself returns “false” with an error message. All functions that may be called beneath DoCycle

return a value to indicate whether an error occurred, and pass back a pointer to a string containing any error message. The error message may be added to, rather than overwritten, for user-information and debugging purposes.

The sequence of calls in DoCycle is as follows:

- Harvesting: check whether this year is a tree harvesting year, and if so, perform harvesting (optharvesttime());
- Thinning: check whether this year is a silvicultural thinning year, and if so, perform thinning (thintime());
- Strip clearing or replanting: check the year, and perform strip clearance, strip replanting or neither, as appropriate (stripstime());
- Output data after harvesting, if requested (DoOutput());
- Simulate individual tree growth: evaluate the diameter increment, then add it to the existing tree diameter (calc_growth());
- Simulate stochastic annual natural mortality (calc_naturalmortality());
- Simulate annual recruitment of new trees to the plot at a diameter of 10cm (calc_recruitment());
- Update forest object ages, object "status" attributes, etc, for another year of simulation (calc_ages());
- Output data annually (or every few years) as requested (DoOutput()).

The silvicultural treatments (harvesting, thinning and strip clearance and replanting) are executed on a fixed timescale by comparing the simulation year (or time since the treatment last happened) with a parameter defining the cycle-length of the operation.

"Optharvesttime()" is complicated slightly because there are two possible harvesting models, at the highest level. The first is a single prescription, applied every few years (a fixed interval). The second is two prescriptions, that alternate with each other. The second prescription is used a fixed number of years after the first prescription has been used.

Growth and natural mortality are calculated and executed on a per-tree basis. Recruitment cannot be treated in the same way, since the recruited trees do not exist in the model beforehand. Recruitment is calculated per gridsquare.

3.8 Data i/o

Described here are the classes and structures: Csymdata, Csymstand, Ctables, Ocoldata, Ocolinfo and Otable and their implementation. Some functionality of these classes is inherited from classes defined in ODBC: CDatabase and CRecordset. These are defined in the file "odbcinst.h".

The default i/o does not require ODBC functions or software, and instead uses standard text formats. This makes the code more robust, less susceptible to external software change and faster to execute. The ODBC functionality within SYMFOR took a long time to program successfully, is poorly documented, database specific, difficult to maintain or debug and slow to execute. None of these problems were predicted before the attempt to code ODBC functionality was attempted. This is an object example of where incorporating the most modern code and maximising user-flexibility is a bad idea. The general rules are: to give flexibility where it is simple to do so; to program only code that is simple to debug; and to program code that does not rely on external code for operation.

3.8.1 Data input

Data input is performed using DoReadDBdata() for tree data, and DoReadDBdataStand() for stand data. The two methods are almost identical, so only the former is described here except where differences occur.

Initially, the random number seed is set using the value generated from the date and time by the MM. This only needs to occur once, and is performed by DoReadDBdata(). Then the process splits: the default route is to use text files ("*.csv", "*.txt") for data input, but the original and more flexible option is to use ODBC routines to access database tables directly.

For text files, the function readtextDBfile() opens the file, reads in the column titles, checks all required fields are present and notes the order of the columns. Each row is read in, and broken into variables using atomise(). A new livetree object is made for each row in the file, and initialised with the data values read in. For tree data, all lines are read in. For stand data, only the first line (of data) is used to create a stand object since only one stand object is allowed in SYMFOR. The file is then closed. Stand data are read in after tree data, and so when the stand data file is closed the input data are checked using filldata(), to complete any gaps and create gridsquares, and checkdata(), to check the validity and consistency of the final data. Data input is then complete.

For ODBC database data input, the process is more complicated. An ODBC database connection object (CDatabase) is created. The opennewDSN() function uses SQLConfigDataSource() (ODBC) to create an ODBC Data Source Name (DSN), with a hardcoded name, "SYMFOR DSN", of the correct database type. [This involves some database-specific variables that are difficult to obtain (when programming) – the easiest way is to create a new DSN using the Windows Control Panel icon "ODBC Data Sources", then use "regedit.exe" from the "Start Menu, Run" to view the Windows registry, searching for the new DSN name.] The Cdatabase object is initialised with the new DSN is assigned to the database connection object using OpenEx(). A SYMFOR recordset object (class Csymdata, Csymdata.h, for tree, and Csymstand, Csymstand.h, for stand data) is created and initialised to the CDatabase object and then opened for a "snapshot" view, making the software link to the database table complete. The "open()" process checks that the required columns are present, and matches them with variables in the recordset object. The MoveNext() function is used to read each row in the table, creating a new livetree for each line. For stand data, an error is returned if there is more than one row since only one stand object is allowed in SYMFOR. The recordset objects are then deleted. Stand data are read in after tree data, and so when the stand data file is closed the input data are checked using filldata(), to complete any gaps and create gridsquares, and checkdata(), to check the validity and consistency of the final data. The CDatabase object is deleted, and data input is complete.

3.8.2 Data output

Data output is programmatically significantly harder than data input, because the variables, type of variables and order are not known until run-time, and several data output tables are allowed simultaneously.

Pointers to the data output table objects, "pOtable[]", and the number of output tables in use at any one time, "nOtables", are stored globally (sf99.cpp).

The DLL entry point, DoCommitOuts() is used to register the user's intention to output to a set of tables. The set of Otable objects is either deleted, deleted and re-written, or appended. New Otable objects (Otable.h) are created as required, using the main constructor function. This has all the necessary information, regarding the location, type and name of database, the name, type and

number of variables to be output, and the details of when to output data to this table. DoCommitOuts() is used whenever the user edits the output settings. The column information (name, type, SYMFOR object variable name) is stored in a special data structure, “Ocolinfo”.

Once a simulation begins, DoSetupDBOuts() is used to initialise each table (Otable.cpp), preparing it for data to be written. There is an option to overwrite existing data, or append to it. Either way, for output to ODBC database tables the same initial process as for data input is followed: A CDatabase object is created, and a new ODBC DSN is created called “SYMFOR DSN” [Note: could this name be used more than once simultaneously, causing problems?]. The connection to the database is then made with OpenEx(). For overwriting previous data, “DropThenAddTableDB()” is used to drop the existing data and create a new table with column names and types. For text formats, DropThenAddTableText() (for overwriting) or OpenTextTableFile() (for appending) opens the file, sets the delimiting variable and writes the column titles.

At many points during the simulation, the function DoOutput() is called, both by DoCycle() and by the MM, to output data according to the user’s specifications. DoOutput compares the output reason (a code, indicating whether the DoOutput call is being made at the start of the run, annually, after harvesting, etc) with that selected by the user for each table. If there is a time restriction involved (such as outputting every 5 years) then the simulation year is also checked. If data should be output to any table in a particular DoOutput call, the Otable class “writedata()” function is called for that table.

Writedata() uses an array of Ocoldata structures (Otable.h) to represent the data. Each Ocoldata structure contains an array of floating point numbers, representing the data in one row of the table. Writedata() fills the array of Ocoldata structures with data using calls to GetDataDLL(). GetDataDLL() was written to return data to the MM for use in visual displays, but it is also used to pass data from forest object storage to the output data structures. Once all the data to be output in this “dump” has been collected into the Ocoldata array, it is written to the table row-by-row. For text format, a standard C-language “sprintf()” command is used. For ODBC databases, an SQL command is made and executed with the ODBC (CDatabase) function “ExecuteSQL()”.

The DLL entry point “DoResetOuts()” is used at the end of a run, and before initialising the output tables. It deletes all the existing output tables, using the Otable function “close()” to delete all dynamically allocated memory.

3.8.3 Database table query

In choosing a database table, either for data input or output by ODBC, the user selects a database as a standard Windows directory or file. The set of tables in the database is then accessed using ODBC with technology similar to that described above, with the DLL entry point function, GetDBtables(). This uses a class called CTables (CTables.h), derived from the CRecordset class, similar to the OTable class but more simple.

3.9 Summary of DLL entry points

Most of these functions are described in the preceding sections, however it may be useful to describe the typical usage of the DLL functions during SYMFOR operation.

Initialisation:

1. **DoSfDLLinit**: Declares the global variables and forest object groups and initialises the possible models: swappable functions, modules and parameters.

Model selection and setting:

2. **GetSwFuncInfo**: returns the name and type of a swappable function.
3. **GetModInfo**: returns the number and names of all modules for a swappable function.
4. **GetChosenModDeps**: returns the names of the swappable functions that a module uses.
5. **GetParInfo**: returns the number and names of all parameters for a module.
6. **GetParValues**: fills a block of memory with the value(s) of a particular parameter.
7. **SetModChoice**: specifies a particular module to use for a swappable function in the DLL.
8. **SetParValues**: Sets the value(s) of a particular parameter in the DLL
9. **DoCommitParameters**: sets the DLL global variables to the values as stored in the parameter class objects.

Output table selection and settings:

10. **GetDBtables**: returns a list of tables that are found to be in an external database, for use in selecting input and output data tables.
11. **DoCommitOuts**: Sets the selected output tables, times and data information in the DLL.

Data input at the start of a simulation:

12. **DoReadDBdata**: input tree data from file.
13. **DoReadDBdataStand**: input stand data from file.
14. **DoReadStandWizard**: input stand data using the MM stand wizard.

Output table initialisation at the start of a run:

15. **DoSetupDBOuts**: open the output data file or database table.

Data retrieval from the DLL by the MM for the visual displays:

16. **GetObjectNames**: returns the names of the forest objects in the DLL (livetree, skidtrail, etc).
17. **GetObjVarNames**: returns the names of variables of forest objects about which data are available.
18. **GetDataDLL**: returns the values of forest object variables for all objects of that type.

Model execution:

19. **Docycle**: simulates one year of forest change.

Data output during a simulation:

20. **DoOutput**: output data to tables as specified.

Resetting variables at the end of a run or before a new run begins:

21. **DoResetOuts**: delete any existing output tables.
22. **DoResetDLL**: zeroes the forest object groups, and deletes all forest objects in memory.

Other functions:

23. **DoReceiveData**: Special function, not used by the MM, but written for auxiliary programs using the DLL to feed tree data to SYMFOR one tree at a time.
24. **DoSetRandseed**: Special function, not used by the MM, but written for auxiliary programs using the DLL to set the random number seed in SYMFOR.

4 On-line help pages

This section describes the structure of the help pages, how they operate and link, and gives some indications of what pages should be changed as a result of changes to the software.

The help pages are structured as a downwards tree. All the help files are contained within the directory “hlp”, and each branch of the tree leads to a subdirectory structure. Where relevant, there are links from one branch across to another, but these do not compromise the tree structure.

The top page, “contents.html”, contains links to 14 branches, and some helpful pointers for those who do not find menus useful. There is also a search engine, whose files are kept in the SYMFOR directory. The most extensive of the branches are the User Manual and Model Components. The user manual contains a branch giving information about each window of the MM, a tutorial and sections for the common SYMFOR activities. Each of the windows in SYMFOR has a “help” button, which uses “autorun.exe” to open the default web browser with the help page for that window. Thus the help pages and manual are linked on-line to the program itself.

The Model Components section contains pages listing each swappable function, module and parameter, and pages for each of the swappable functions, modules and parameters. There is extensive cross-linkage in this section, and no elements are excluded. The aim is to fully document all model components, so that the user can understand and document any model made with the SYMFOR system.

The help pages are kept to the same format by using a style-sheet (“helpstyl.css”). Fonts are a non-standard combination, “arial, helvetica”, that works in the two most common web browsers (Microsoft Internet Explorer and Netscape). To create a new page, the most reliable method is to copy an existing page, and edit it. Netscape Composer is included with the Netscape Communicator suite, and is a simple visual editor: from Netscape navigator, use the “file” menu and the “edit page” option. Help page file names are kept obvious, with each parameter help page given the same name as the parameter, for example.

If modifications are made to the system, or if a bug is found, the web pages should be updated. The “What’s new” page lists significant modifications and can act as a diary of product development. The “Known bugs and errors” page is generally fairly small, since bugs and errors are often corrected almost as soon as they are observed.

Many of the sections of the help pages are not complete, or even started. They were included because of the aim to produce complete documentation for the product, although the documentation itself takes longer to produce.

5 Installation versions

The aim of installation is to put the software onto a system such that the system can operate the software immediately and flawlessly.

5.1 Installation issues and files

Firstly, SYMFOR uses programming languages that create executable (or library) files, each of which requires other libraries to operate. So the software is not completely self-contained. In addition, the software is restricted to Windows operating systems. The implications of this are that

to ensure SYMFOR can run following installation, ALL the necessary library files must be installed along with the SYMFOR-specific files. The external library files required are:

MSVBVM50.dll
StdOle2.tlb
OleAut32.dll
OlePro32.dll
AsycFilt.dll
Ctl3d32.dll
ComCat.dll
Grid32.ocx
MFC40.dll (for the DLL compiled with MSVC++ v5.0)
MFC42.dll (for the DLL compiled with MSVC++ v6.0)
MSVCRT40.dll
ThreeD32.ocx
MSChart.ocx
SysInfo.ocx
ComCtl32.ocx
ComDlg32.ocx

The system may already have a version of SYMFOR installed, in which case the existing files should be overwritten – or at least, it is the user's responsibility to copy the existing files to a different location before re-installation. The SYMFOR-specific files required are:

symfor2000.exe
sf2000.dll
mm.ini
pars.txt
mods.txt
runsets.ini
about.txt
autorun.exe
postp1.exe
search.html
searchengine.exe
index.txt
\standard models\parameters\standardpars.txt
\standard models\modules\standardmods.txt
\data\synthetic\exampletree.csv
\data\stand\examplestand.csv
The help files: "\Hlp"

A third complication is that the auxiliary files "mm.ini", "pars.txt", "mods.txt" and "runsets.ini" store information about the latest settings used. In order to minimise confusion by new SYMFOR users, the initial settings should be sensible and should not require alteration before a basic SYMFOR simulation can be performed. This means that the developer's own SYMFOR directory often can not be used to make an installation version without modification. A parallel directory called "Symfor install", containing the latest versions, can help to keep the installation version separate to local working versions.

5.2 Making an installation version

The following procedure can be used to make an installation version of SYMFOR. Further processes will create a CD installation and a web-download installation. Altogether the software required for this includes: an "application set-up" included with Microsoft Visual Basic v5.0, WinZip Self-Extractor, and a CD welcome package that was written for SYMFOR.

5.2.1 A basic installation directory

1. Edit \symfordev\symfor2000.swt as necessary.
2. Copy "\Symfor" to "\Symfor real" and copy "\Symfor install" to "\Symfor". Check the versions now in "\Symfor" are the latest versions, and are not debug compilations. Check that the parameter sets, etc, contain only the "default" set, and that these are up to date.
3. Ensure that "\Symfor\standard models\modules\standardmods.txt" and the similar file for parameters contain sensible module and parameter sets.
4. Run VB5 Application set-up wizard and choose file "\symfordev\symfor2000.swt" at the first prompt. All other prompts should require only a Return key.
5. Copy "\Symfordev\Setup disks\2000" to "\release\symforCD\2000".
6. Copy "\Symfor" to "\Symfor install" and copy "\Symfor real" to "\Symfor".

5.2.2 The Help Pages

1. Using WinZip, zip the "\Symfor\Hlp" directory.
2. Run WinZip Self Extractor, choosing: a "standard self-extracting exe", "\Symfordev\Setup Disks\helpfile.txt" as a file to prompt the user with, set the default path to install to to "C:\Program Files\Symfor\Hlp", 32-bit Win 95/NT version and "default to overwrite files without prompting". All other options should be left at default settings or blank.
3. Copy "Hlp.exe" to "\release\SymforCD\Hlp.exe".

5.2.3 Creating a CD

1. Follow the instructions as for a "basic installation".
2. Follow the instructions to make the help pages installation.
3. Start "Easy CD Creator", and cancel out of the wizard.
4. From the "file" menu, select the "open CD layout" option and choose "symfor2000.cl3".
5. Create the CD using x1 speed, and "test and record" if it is the first CD (otherwise just record will be fine). Use all other options at their default values.
6. Test the CD in a conventional CD ROM.

5.2.4 Creating an internet download installation

1. Follow the instructions as for a "basic installation".
2. Copy the directory "\release\symforCD\2000" to "\temp\2000".
3. Zip the "\2000" directory, making "2000.zip".
4. Run WinZip Self Extractor on "2000.zip", choosing a "self-extractor for software installation", "\Symfordev\Setup Disks\scriptfile.txt" as a file to prompt the user with, 32-bit Win 95/NT version and "default to overwrite files without prompting". All other options may be left at default settings or blank.
5. Copy "2000.exe" from the "\temp" directory to the web download directory, update the web pages and make a link to the new download version.

5.2.5 Creating a floppy disk installation

Follow the instructions as for a "basic installation", but choose "Floppy disk" rather than "single directory" as a method for distributing the software. Alternatively, choose "disk directories", then

copy the contents of those directories to floppy disks. Note that you cannot copy the directories themselves, because there won't be room on the disk – copy the contents only.

5.3 Testing installation

This is difficult, because whether the installation was successful may depend on many things. As a minimum, the installation version should be run on the nearest computer running Windows. If possible, test the installation on brand new machines running each of the Windows operating systems!

Make sure you are not going to overwrite a "\Symfor" directory by moving any existing directory to another location, such as "\Symfor old". Run the installation of both the software and the help pages. Check that the new "\Symfor" directory has been created. Try running "symfor2000.exe". Try doing a typical run, without selecting any data files or parameter sets that are not default. Check that data are read in by using the displays, and that the operation appears normal. Check the default settings for data input, parameter values and module choices. Check that the help pages exist and are linked from the program correctly.

This is not a procedure that should go wrong, and there are no standard methods for correcting it if something does go wrong. Basically check the directory structure and file content compared to that on the machine where the installation was made – they should be the same.

6 Debugging

Debugging is the process of finding errors or omissions in the software, often as a result of the program having crashed during operation. There are no hard and fast rules of finding bugs, but some general approaches may help. The job of debugging SYMFOR is made more complicated by the use of two programming languages and two sets of code: the MM and the DLL. Sometimes the bug will crash SYMFOR, sometimes it will make the behaviour spurious. The latter is often harder to debug.

1. Note the random number seed, the input data, module choices, parameter sets, output data, displays and run results used in the run where SYMFOR crashed, so they can be used again in diagnosis.
2. Ask some questions:
   Does the error repeat itself?
   Does it also happen when the run is done on another machine?
   Is the latest version of the software being used?
   Are the models/parameter values being used sensible?
   Is there an error message?
3. If it does not appear that the user has made an error (as is usually the case, with input data formats or something), you'll have to start debugging. First, try to make the error reproducible.
4. Try changing things to see what affects it; displays, parameter values, input data sets, etc.
5. When you think you have some idea whereabouts in the code (probably in the MM) the bug is, run the code in the debugger mode (VB: press F5, VC++: Set the active build configuration to "debug" then press F5).
6. Set some breakpoints, perhaps at the start of the routine you think might be causing the problem. If the program crashes soon after opening a particular window, try setting the breakpoint at the start of the "form_load()" function of that window.

Likely bugs include the program trying to write to the (*i*+1)th element of an array with *i* elements, dividing by zero or (in C++) using a pointer that hasn't been correctly initialised to point to a variable.

Some particularly nasty bugs occur in SYMFOR when there is a "divide by zero" in the DLL. This does not seem to be caught by the C++ compiler, which is extremely annoying. It turns up as a divide by zero error in the MM, but when it is debugged there is no sign of a divide by zero. It probably happened in the DLL, and that is what should be debugged. If the bug is affected by the model, it is probably in the DLL. Otherwise it is probably in the MM.

Experience is the most valuable tool for debugging, and while I have tried to pass on some tips here, the approach to finding a bug depends a lot on the symptoms, and there are no general rules. Sorry.

7 Index of filenames

There are many files involved with SYMFOR, to compile it, to distribute it and to run it. The files mentioned in this document are listed in an index here, however there are many files, particularly C++ files where the class is referred to instead of the file, that are not listed.