Designing and implementing computer simulation models for portability and reuse

E. Post

Centre for Advanced Computational Solutions, Lincoln University, Lincoln, Canterbury, New Zealand
poste@lincoln.ac.nz

Abstract: As computers become more powerful it becomes feasible to develop computer models that more accurately approximate the real systems they are simulating. However, such models are increasingly complex, perhaps taking years to develop, so it is important to maximise reuse of existing models and design new models to facilitate reuse. Unfortunately many existing models are not easily reused for various reasons. For instance, the computer language used to write the model may not be portable, or may be a proprietary system requiring a licence or particular operating system, or it may not have been written in such a way that it can easily be extended. Furthermore, computer hardware and operating systems are continuously changing and high-performance computing, such as parallel computing on a Linux cluster, is increasingly available. It is limiting and potentially expensive if a computer model cannot be ported to another system without being rewritten. In these days of reduced research funding it is becoming increasingly important to use such funding more effectively. One way to enable more strategic use of research funding would be to increase the reusability of computer models so as to build on the work of others rather than duplicate it. This is particularly so in cases where the cost of the validation exercise is high. This paper makes a number of recommendations for developing reusable, user-friendly, robust, flexible, extendible and generic computer models. These include such issues as choosing an appropriate language, using appropriate programming techniques, designing for possible portability to another operating system or environment or parallelization. It also discusses the use of a framework and replaceable components for a model of a complex system rather than a monolithic application.

Keywords: *software design; programming; portability; reuse*

1. INTRODUCTION

Scientists have used computers for research since computers first became commercially available. This usage has increased dramatically since personal desktop PCs became commonly available in the 1970's and 1980's and scientists could teach themselves to program and perform their own computing. However, in these last 2 to 3 decades there have been some significant changes in computing.

- Computers have increased tremendously in power available, thus making it possible to achieve nowadays on a desktop PC what could formerly only be done on a supercomputer.
- Computer programs have increased significantly in both size and complexity making it much more difficult to create such a large and complex program, especially if it is incorporating pre-existing units.
- Computer programming has become increasingly more complex and difficult. For instance, the introduction of Graphical User Interfaces (GUIs) and the object-oriented paradigm mean it is no longer as simple for an amateur programmer to write good software as it was using the simpler languages and computer environments of the 1980's.
- It is increasingly expensive to perform laboratory tests or run field trials to validate computer simulation models. Such validation tests, especially for field trials, can easily cost far more than the software itself. If possible it is much more effective to re-use existing models that have already been validated, rather than develop new models still requiring validation.
- It is increasingly difficult to get research funding so it is important to use what funds one has effectively. This means it is important to consider the possibility of re-using software that already exists rather than starting from scratch each time.

These are just some of the factors that need to be considered when developing scientific software. The rest of this paper gives some suggestions to help scientists develop better, more flexible and effective software by making it more easily

possible for others to re-use, thus making more effective use of scarce resources.

2. INVESTIGATE WHETHER WHAT YOU WANT ALREADY EXISTS

Before spending considerable time and resources developing software do some research to see if someone else has already developed such software and will make it available for you to use, either free or for a reasonable fee. Recognize that even if you have to pay for this software it may well cost far less than it would cost for you to develop it yourself, in particular if the existing software has already been validated. Also using existing software will allow you to obtain results far quicker and publish earlier!

3. OPERATING SYSTEMS

First consider which operating system the program may be run under. Scientists work on a range of different operating systems, such as Microsoft Windows, Apple, VMS, various flavours of Unix, including Linux, and others.

If the software you plan to develop is ***ever*** likely to be used by scientists in other institutions then you should seriously consider choosing a programming language or package that is available for the most common operating systems, especially Linux and Windows. Programs developed in such languages under one operating system should then easily be able to be ported and recompiled on another and should require no more than minimal changes.

4. ANTICIPATE POSSIBLE PARALLELIZATION

Not all languages are easily parallelizable, so it is important to consider this question before choosing a language.

Consider whether you (or anyone else) are ever likely to want to run your program on a computer with better performance, such as a parallel computer. Many modern scientific applications are so complex that they can take a very long time to run and it becomes important to parallelize them to run in less time. Also, sometimes a computer simulation model originally developed as a stand-alone program may later be incorporated as part of a larger system that runs on a parallel computer. In these cases it is most likely that the high performance computer will be using a Unix operating system. Thus the issues described in section 3 become important as it may become necessary to port your application to another operating system in order to parallelize it.

Most parallel programs are written in C, C++ or Fortran, often using MPI (Message_Passing Interface), and may also be written in Python. It can be very difficult, although not necessarily impossible, to parallelize programs written in other languages. Post (2002) describes a case of parallelizing a Smalltalk application.

5. CHOOSE AN APPROPRIATE PROGRAMMING LANGUAGE

Then you need to choose which language to use. In this paper we will consider primarily the case where scientific programs are being developed in conventional programming languages, rather than for instance specialised mathematical, statistical or simulation packages.

There are several factors to consider, such as which programming paradigm to use, which language to use and which compiler to use.

5.1. Choice of programming paradigm

In the 1980's most programmers used a procedural or functional approach, using languages such as Basic, Fortran, Pascal or C. However there are inherent difficulties in such languages that greatly restrict the ability to program correctly, in a way that the program can easily be maintained, extended or used, especially with the much larger programs of today.

Therefore, if at all possible use the Object-oriented (OO) paradigm when designing and developing software. Objects map well onto objects and systems in the real world, and the OO approach will facilitate future reuse and extension. If you cannot use an OO language then still design the software with the OO paradigm as far as possible, making all code units modular, passing data into these units with parameters and returning results from each routine.

Suitable object-oriented languages are Python, C++, and in some circumstances Java, Delphi, Smalltalk or C#, but see below for other considerations regarding these last four languages.

5.2. Availability of languages across operating systems

If at all possible use standard languages, such as C++, C, Fortran, or Python that are widely available (even free) for various computer architectures and operating systems. (You cannot necessarily predict that your model will never be used on another operating system.) Of these it is strongly recommended that you use Python, or perhaps C++, if at all possible as they are object-oriented. Avoid using C or Fortran if you can.

Especially if you are an amateur programmer you should probably choose Python as it is simpler to learn and perhaps less likely to cause problems.

Some other languages, such as Java, Delphi, C# and Smalltalk are available for different operating systems, such as for both Windows and Linux. Delphi is known as Kylix for the Linux operating system. Note though that programs written in Delphi are not exactly the same as Kylix programs and may not port completely without trouble, with problems most likely to occur with relation to the GUI. A version of C# runs on Linux in an interpreter. Other issues to be considered are discussed further below

5.3. Availability and cost of compilers and licences

You also need to consider the cost and availability of compilers for various languages or perhaps licences for specialised environments. Researchers often have little money and may not be able to afford to buy commercial compilers. And if, for instance your model needs to run in a specialised environment such as ACSL, then the cost of licences may restrict re-use of your model by others.

For instance, reasonably good quality free Python, C, C++ and Fortran compilers are available for several platforms for standard versions of these languages. However, if you really have to use Fortran then it is advisable to use Fortran 77 for which compilers are widely available at no cost on many platforms. Currently there are no free compilers for Fortran 90 or Fortran 95 for Linux, although there have been in the past and some versions are presently under development.

Delphi, although a very good language, is proprietary and it costs money to pay for the compiler, although not very much. Java is freely available for most common operating systems, although this is usually, but not always, interpreted rather than compiled. Smalltalk interpreters from the same company for different operating systems do exist in versions that are free for non-commercial use.

5.4. Virtual machines

Some languages, such as Java, C# and Smalltalk, are interpreted rather than compiled, although Java compilers do exist. In these languages the interpreted code is executed on a virtual machine. This can mean that you do not usually achieve as good performance as you would with a compiled languages such as C++. Python is usually interpreted, but executes very fast, and can also be compiled. Python can also be parallelized.

In particular, if there is a possibility your program may later be parallelized, then it is better to avoid languages using a virtual machine as it is very difficult to parallelize them, although not impossible. See Post (2002). For performance reasons it is usually preferable to choose a compiled language rather than an interpreted one.

6. AVOID PROPRIETARY OR SPECIALISED ENVIRONMENTS

Avoid using proprietary and specialised environments such as ACSL, Microsoft, Borland. Developing software in such environments may limit the re-usability of the model, either because it becomes too expensive to convert the software to a new environment or because of the cost of licence fees, which may be prohibitive, especially in a parallel processing environment where the licence fee may be per processor.

If you must use a proprietary environment for development, such as for instance Microsoft Visual C++, use standard features and libraries in these languages and avoid using implementation-specific libraries. (The documentation should indicate whether features are standard (usually ANSI) or implementation-specific.) Generally most programs written for Unix will port easily to Windows. However, the reverse is not generally the case because there are so many proprietary products for Windows.

If it is useful to use for instance a specialised simulation or statistical or mathematical environment or language for developing a model because of the features it provides, try and choose an environment which will allow the export of the model in standard C or Fortran code that can be compiled and which will run independently and does not have to run in a proprietary environment. Then this exported code could be incorporated as a component model for another system.

If it is essential to use specialised libraries, such as for instance mathematical or engineering libraries, choose those which are well-known and widely available on different platforms, such as NAG or Portland Group. Otherwise develop your software so that it would be relatively easy to use alternative libraries if necessary.

7. USE THE MODEL-VIEW-CONTROLLER APPROACH

Use the Model-View-Controller concept when developing software, where the user interface is completely independent of the code implementing the actual model, with these two layers connected by an intermediate controller or transaction layer.

This is described more fully in VisualWorks (2001) and many other sources.

The many advantages of this approach are well-known, such as for internationalisation. However, in the modelling context some of the main advantages are that it increases the portability of the code for the model. Most incompatibilities between implementations on different operating systems and architectures occur in the GUI which may involve considerable change when porting an application. However, usually the code for the model itself will port with minimal trouble providing it has been written in a standard language using standard libraries. This separation of the model from the GUI is also important in modelling as quite often it will be necessary to run the model independent of its GUI, such as for instance when it becomes a component of a large system or in parallel processing. If this approach is not taken it can take considerable work to separate model code from GUI code.

8. GENERAL PROGRAMMING TECHNIQUES

No matter what programming language you eventually choose there are several techniques that should be used to make your code more maintainable, flexible, extendable and re-usable.

8.1. Avoid using global data

Not only does global data make it difficult to keep modules independent in the application but it can contribute to difficulty during parallelization.

8.2. Define interfaces between modules carefully

Ensure that all modules (whether methods, procedures, functions, sub-routines or just blocks of code) of your program have a well-defined interface for calling that routine, passing parameters into it, and returning results. Modules should not depend on for instance using global data, as this makes them liable to error if changes are made elsewhere in the program, and less easily re-used as they are dependent on outside information to run correctly.

8.3. Use standard data types

Use standard data types such as integers, IEEE floats and doubles and be sure that interacting applications know whether they are using for instance 1, 2, 4 or 8 byte integers. Also be sure to be aware of whether the code is using either 1-byte or Unicode chars and be consistent. Avoid using implementation-specific data-types such as the Pascal 6-byte Real.

8.4. Consider computer architecture issues

If the model may be ported to a computer of different architecture be aware that there may also be issues with whether integers are little-endian or big-endian, for instance when reading and writing data files. This is also important if parallelizing a program for a heterogeneous system. If you can use MPI for inter-processor communication this will automatically be taken care of, but if you are using sockets you will need to take care of it yourself, although there are functions available for use with sockets that will help with this.

8.5. Use in-code documentation

One of the biggest problems when trying to re-use someone else's code, or even your own after a long time, is that frequently it is poorly documented, or even not documented at all, making it extremely hard to understand.

To increase understandability of your program you should try to choose good descriptive variable and function names, so that one can almost read the code as if it is in a natural language such as English. If your program is likely to be used internationally it is helpful if you use an internationally recognized language for naming variables and functions such as English.

It is also extremely important to document your program well. This is best done in the code as separate documentation is frequently not kept up to date, or may even be lost. Also such in-code documentation can frequently be used by document-generating tools to produce up-to-date technical documents describing the application. It is not necessary to be excessive about using comments. However, the following are good habits to develop:

- At the beginning of the file containing the main function of your program put:
    - the name of your program
    - the name of the programmer and contact details, including email address
    - the date it was started
    - a brief description of what it is about, including any important references to scientific papers
    - Every time the program is revised add further information as in the points above, but instead relating to the revision and why it was done. e.g. bug-fixes or new features or both, and list everything new done to that revision.

- It is also good to allocate version and release numbers to different versions of your program. Every time there are major changes it should be a new version, and every time there are minor changes it should be a new release. e.g. 2.3 is version 2, release 3. If this information is kept in the comments and other people use your program then they can tell you what version they are using if they need changes. It is good practice to use a version control system such as CVS.
- Every module, variable and function should have a comment describing what it is for.
    - If it is a variable there should also be a comment describing any units, and perhaps minimum and maximum valid values.
    - If it is code it should also describe what the parameters are for, any relevant units, and what values are returned. In addition, any piece of code for which it is not immediately obvious what it does, should have comments explaining it.
- If code is implementing equations from some reference work, such as scientific papers, then there should be a comment giving the full reference of the source of the equation.

8.6. Avoid "hard-coding" items that may change

Don't build in such items as file names and paths into your program as these may change between different computers. Rather allow your program to read path and file names in from a text file. This gives greater flexibility in using the program especially if changing between operating systems

Also avoid programming actual initial values of variables where it is likely they may change between runs of the program. Rather allow the program to read in files of initial data. This will allow a user to change start-up parameters without having to re-code the program.

8.7. Initializing data and inter-changing data between modules and applications

In most cases it is advisable to use text files rather than binary files for inputting such initialising data as described in section 8.6. This makes it easier for a user to use a text processor to edit the data files before running the program using them.

It is sometimes also useful to use text files for outputting data that may be read into a spreadsheet or another program. You can separate data items by spaces, tabs, commas or semi-colons if necessary.

In both cases it is important to put comments in the text files to describe each item, or column or row of items, including units used., and valid maximum and minimum values. You may also need to put a full reference to the source of your information for making this decision. It is common practice to indicate that a line in a data file is a comment by making the first character of the line in column 1 a #.

8.8. Avoid duplicating other software

If other software exists that will post-process your data and produce the graphs and statistics you need, then do not waste time (and money) programming these features into your program. Rather just export the result data in a file suitable for import into the other software.

9. COMPLEX SYSTEMS

Twenty to thirty years ago it was a considerable achievement just to write a computer simulation model for a fairly small component. Also, because computers were so slow the models could not be very complex or they ran for too long. However, with the huge increase in computing power it is now possible to run much more complex and detailed models very quickly. In addition science has moved on and many simple systems are now well understood. As a result there is now significant interest in modeling large complex systems.

Complex systems can be programmed as huge monolithic systems. However, the disadvantages there are it is difficult to change a part of the system to reflect new research.

A good alternative is to model a complex system by creating a framework that manages the system but that interacts with component models. For instance, a farm may include management strategies, climate information, animals and pastures (or other food). If this is modeled by using a framework to manage the daily, weekly, monthly or annual cycle of a farm then this framework can interact with different models for the other components, such as using cows or sheep or other animals.

Advantages of this approach are that:

- One can re-use existing models already created by yourself or other researchers
- One can substitute different models, perhaps representing different research, for the same component. e.g. different models of cows.
- One can substitute models of different levels of complexity for different simulation experiments.

To achieve this type of complex system however requires extremely good programming techniques such as described in this paper. However, the potential savings in cost and increased flexibility in using the simulation model for research make it worth considering whether to spend the time and effort to achieve this. Neil et al. (1999) and Sherlock et al.(1999) describe the implementation of a complex system to simulate a dairy farm by using a framework and components.

10. RECOGNISE COMPUTING AS A HIGHLY-SKILLED PROFESSION

In the 1980s it was relatively easy for someone who was not a trained computing professional to teach him or herself to program and then develop their own scientific software. However, computing has increased so much in both complexity and also the potential of what can be done on computers by experts. Therefore if scientists want to make most effective use of computers and get the greatest benefit from them they should strongly consider employing computing experts to do their programming for them. Experience has shown that a computing expert can usually achieve excellent (and better) results in a considerably shorter time than an amateur. This approach would assist a scientist to use time far more productively in doing science rather than dabbling in programming and taking very much longer to achieve results which are probably not as good as those that would be achieved by an expert. Scientists are not expected to build their own very advanced technological laboratory equipment, so why should they be expected to create their own software?

11. CONCLUSION

If new simulation models are developed considering the issues and using the techniques described in this paper then there is a far greater likelihood that such software can be re-used and extended. This should thus lead to less wastage of time and money and more effective use of scarce resources.

Even if you do not expect your software to be re-used it is good programming practice to consider the issues described in this paper and implement the techniques suggested. Then, if someone does want to re-use your software in the future it is in a state where this can be done.

12. ACKNOWLEDGEMENTS

I am very grateful for the help given to me by Rob Sherlock in discussing these issues. I also appreciate all that I have learned over the years from many scientific programmers, both for the good techniques learned, and also for what not to do!

13. REFERENCES

Neil, P.G., Sherlock, R.A. and Bright, K.P. Integration of legacy sub-system components into an object-oriented simulation model of a complete pastoral dairy farm. *Environmental Modelling and Software*, 14, 495-502, 1999.

Post, E. Adventures with Portability, Third LCI International Conference on Linux Clusters: The HPC Revolution 2002, Florida, USA, October 2002

Sherlock, R.A. & Bright, K.P. An object-oriented framework for farm system simulation. MODSIM99 – Proceedings of the International Conference on Modelling and Simulation, Modelling and Simulation Society of Australia and New Zealand Inc., eds. L. Oxley, F. Scrimgeour, and A. Jakeman, Hamilton, New Zealand, 783-788, 1999.

VisualWorks Internet Connectivity Cookbook, Cincom Systems Ltd., Cincinnati, Ohio, 2001. http://www.cincom.com/newsmalltalk/prodinformation/pdf/vwiccb.pdf

References to software websites

ACSL (Advanced Continuous Simulation Language) Home Page http://www.acsl.com/

Cincom Smalltalk Home Page, http://www.cincom.com/scripts/smalltalk.dll/index.ssp

CVS Home Page, http://www.cvshome.org/

Delphi Home Page, http://www.borland.com/delphi/

Java Home Page http://java.sun.com

Kylix Home Page, http://www.borland.com/Kylix/

MPI Home Page, http://www-unix.mcs.anl.gov/mpi/

Numerical Algorithms Group (NAG) Home Page, http://www.nag.co.uk/

Python Home Page http://www.python.org/

Portland Group (PGI) CDK Cluster Development Kit - Software for Linux, http://www.pgroup.com/products/cdkindex.htm