# EcoLab 4

Russell K. Standish
High Performance Computing Support Unit
University of New South Wales
Sydney, 2052
Australia
R.Standish@unsw.edu.au
http://parallel.hpc.unsw.edu.au/rks

**Abstract.** This paper describes a project to build on the EcoLab Simulation System (currently at version 3), to make a truly general purpose, agent-based, simulation environment using the C++ programming language.

## 1. Agent Based Modeling in C++

In this paper, I describe a project to build on the EcoLab Simulation System (currently at version 3), to make a truly general purpose, agent-based, simulation environment using the C++ programming language. [1] was developed by Langton and co-workers to be such a simulation environment. For a variety of technical reasons, some of which will be discussed later, Objective C was chosen as the implementation language of Swarm. Both Objective C and C++ are object oriented extensions of C, but implement objects in incompatible ways. If you already have a model written in C++, or you wish to use C++ specific features not available in Objective C, and you wish to use the Swarm toolkit, you are forced to write a C language interface (a set of top-level function calls) around your model, and then an Objective C class object to represent that interface. This is a tedious, and error prone procedure. Your model becomes a "black box", that Swarm's *probes* cannot open.

EcoLab 4 will not compete with Swarm. Rather it will complement Swarm's capabilities in a number of ways. Firstly, it can be used in its own right as a C++ simulation system. Secondly, it will enable Objective C objects to appear as C++ stream-like objects that can receive "messages". This might enable the use of a Swarm library, for example the random number generator library. Thirdly, it will enable a C++ object to appear as an Objective C object, that can be "probed".

This paper is organised as follows:

**EcoLab History** describes the evolution of the EcoLab Simulation System to where it is today.

**Object Descriptors** describes the key software advance required to emulate the run-time type enquiry necessary to implement a general purpose simulation system.

**Interfacing with Swarm** describes in more detail how this pseudo runtime type enquiry mechanism can be used to add to and maximise the benefit from the existing Swarm development effort.

[1] Swarm

**Parallel Programming** raises the issues involved in designing a distributed parallel computing model. Many lessons have been learnt from the *EcoLab* 3 distributed programming model, but the system does need to be redesigned from the ground up to allow for "distributed objects".

**EcoLab 4** concludes the paper, describing the key steps that need to be done, and proposing an Internet development project that interested individuals can get involved with.

## 2. EcoLab History

*EcoLab* started life as an implementation of a model evolutionary ecosystem originally developed by David Green and myself (Standish 1994). The *EcoLab* model (as distinguished from the *EcoLab* simulation system) is written in terms Lotka-Volterra population dynamics, coded as vectors of numbers representing the population density, and phenotypic properties of each species. The number of species (hence system size) varies over time through the processes of speciation and extinction, setting this model apart from most evolutionary models that employ a fixed number of species (ie speciation being a replacement process). The very first implementation was done in CMFortran in 1991, which quickly demonstrated the inflexibility of fixed, compile time array sizes, and lack of support for sparse array structures. TCL[2] was adopted in the second implementation, done in C in 1992, to allow for runtime specification of the model's parameters. Unfortunately, with no support for array based syntax in C, the explicit loop constructs required to express the model rapidly became impenetrable and unmaintainable.

In 1993, the author became aware of the C++ language, with its flexible extensibility allowing the implementation of an array based syntax. Also the ability to encapsulate code with objects, including initialisation code to be executed prior to runtime, allowed for a much simpler interface to the TCL interpreter. The former developed into the `array` class library, and the latter into the `tcl++` class library, two of the three main class libraries employed within the *EcoLab* simulation system. This code was released as *EcoLab* 1.0 in late 1994.

In 1996, the mutation operator was improved, and a widget added to visualise ecosystem connectivity, and the resulting code was released as *EcoLab* 2 (version 2.1 was a bugfix release on 2.0). Furthermore, the array class library was separated to allow for parallel C implementations, such as in C* (Thinking Machine's parallel C language). Alas, experimentation showed that the most computationally expensive part of the model was sparse matrix-vector multiplication, which proves impossible to optimise for parallel execution due to communication costs (or equivalently on shared memory computers, cache contention).

Swarm was released to the complex systems modeling community in 1996. Since, in many ways, *EcoLab* was attempting to be more than just an implementation of a particular model, I looked at Swarm to see if the *EcoLab* model could be implemented within it. Whilst Swarm has many unique benefits and features, it has two main drawbacks as far as the *EcoLab* model was concerned. The first was the lack of operator overloading in its implementation language (Objective C), and the second was the

[2]Toolkit Command Language (TCL) is a scripting language designed specifically to be easily added to an application written in C. Since its inception, a number of other scripting languages have had this functionality added, most notably Perl, Python and Guile. More information on TCL can be found from http://dev.scriptics.com or Ousterhout's book (Ousterhout 1994).

lack of a scripting ability.[3] Therefore I resolved to produce another edition of *EcoLab*, incorporating best practice features from Swarm whereever practical.

Work began on *EcoLab* 3 in mid-1997, incorporating the floating window format employed in Swarm, and the [4] for plots and histograms. The other major project was implementing spatial dependence in the model, parallelised by a domain decomposition technique (Standish 1998a). This led to the need to compute a binary representation of the model (or portion of a model), that could be shipped between threads implementing the parallel computation. In Swarm terminology, computing a binary representation in this way is called *serialisation*, and has been available in Swarm since version 2 (ca early 1998). Serialisation provided additional benefits, including the ability to checkpoint/restart the model, and to code a separated client-server architecture, with the GUI client running on a different computer to the model.[5] Serialisation is implemented in *EcoLab* 3 by means of the global class library ("global" for *global* variables of the model). It requires that the model must be implemented in a fairly restricted way, ie as a list of vectors and matrices.

Along with progress to the code base of *EcoLab*, a second "*EcoLab*-like" model was implemented, which is a generalisation of Newman's (Newman 1997) exogenous shock evolutionary model. This model was employed in a study of the connection between Newman's model and the *EcoLab* model (Standish 1999). In a later study, a *neutral shadow* model (Bedau et. al. 1998) of *EcoLab* was used (Standish 2000b). Furthermore, a model economy called *Econolab* (Standish 2000a; Standish 1998b) has been defined and partially implemented. Lessons learnt from implementing these different models were incorporated back into the code base, however, the restriction on the types of objects used in the model has become clear.

## 3. Object Descriptors

C++ has two key disadvantages when it comes to simulation modeling. The first problem, often leveled at the language, is its cavalier attitude to pointers inherited from C. This in turn, makes the implementation of automatic memory management schemes such as *garbage collectors* well nigh impossible. Forcing the programmer to explicitly take memory management into account often causes failures such as memory leaks and corrupted memory, which can be the most frustrating errors to debug. This can be partially overcome by the discipline of using well designed class libraries, and avoiding the use of pointers (and the `new` and `delete` operators) altogether. This places the burden of memory management on the class designer. With the evolution of the *Standard Template Library*, a robust and highly flexible class library, this strategy becomes practical, at least in a number of circumstances.

The second problem is that the compiler throws away most of the information about what an object is at compile time. Serialisation, for example, requires knowledge of the detailed structure of the object. The member objects may be able to serialised (eg a dynamic array structure), but be implemented in terms of a pointer to a heap object. Also, one may be interested in serialising the object in a machine independent way, which requires knowledge of whether a particular bitfield is an integer or floating point variable. By contrast, classes in Objective C are objects in their own right. Each object

[3]Even though TCL/Tk is employed internally within Swarm, there is no user interface to the TCL interpreter

[4]BLT widget toolkit

[5]Both client and server consist of the same executable, but have different TCL scripts.

maintains a pointer to its class object, so the structure can be queried at runtime. This was one of the reasons why Swarm is implemented using Objective C. For one thing, it enables the concept of *probes*, ie at runtime one can probe a member value of a particular object, displaying it as a value, or feeding the result into a graph, or whatever.

Standard C++ does provide a run-time type enquiry mechanism, however this is only required to return a unique signature for each type used in the program. Not only would this signature be compiler dependent, it could be implemented by the compiler enumerating all types used in this particular compilation, and so the signature would be different from program to program!

The solution to this problem lies outside the C++ language, in the form of a separate program that parses your input program, and emits function declarations that know about the structure of the object being operated on. These are called generically *object descriptors*. The object descriptor generator only needs to handle class, struct and union definitions. Anonymous structs used in typedefs are parsed as well. What is emitted in the object descriptor is a sequence of function calls for each base class and member, just as the compiler generated constructors and destructors are formed. Function overloading ensures that the correct sequence of actions is generated at compile time.

For instance, assume that your program had the following class definition:

```
class test1: base_t
{
  int x,y;
public:
  double z[100];
};
```

and you wished to generate a serialisation operator called `pack`. Then this program will emit the following function declaration for `test1`:

```
#include "pack_base.h"
void pack(char *nm, test1& v)
{
   pack(nm,(base_t)v);
   pack("x",v.x);
   pack("y",v.y);
   pack("z",v.z,100);
}
```

Thus, calling `pack("",var)` where `var` is of type `test1`, will recursively descend the compound structure of the class type, until it reaches simple data types that can be handled by the following generic template:

```
template <class T>
void pack(char *desc, T& arg)
{append((char*)&arg,sizeof(arg));}
```

given a utility routine `append` that adds a chunk of data to a repository.[6]

Other types that cannot be meaningfully processed, for example arbitrary pointers (which cannot be meaningfully serialised) can be handled by partial specialisation:

[6]Obviously, ensuring machine independence via XDR or HDF will complicate this scenario somewhat

```
class notimplemented{};
template <class T>
void pack(char *desc, T *arg)
{throw notimplemented();}
```

Unfortunately, it doesn't appear possible to throw a compile time error without sacrificing the clarity templates lend.

Finally, arrays whose sizes are known at compile time can be handled by overloading:

```
/* now define the array version */
template <class T>
void pack(char *desc, T *arg, int ncopies)
{for (int i=0; i<ncopies; i++) pack(desc,arg[i]);}
```

There are some subtleties to all of this. Firstly, if an object's members are protected or private, as they are in the previous example, then the resulting code will fail at compile time. This is not a problem with the user's code, nor the *EcoLab* headers, however it can be a problem with standard library headers. One solution to this problem is to create a second program, similar to the first, that inserts friend statements into the definition of classes. This can be run on the standard header files, and saved in a separate directory, rather like the `fix-includes` script of gcc.

A second problem is that some compound types may need hand coding, for example an array structure that has a size member, and a pointer to the data. One needs to add a `#pragma` line to prevent an object descriptor being emitted for that data type.

## 4. Interfacing with Swarm

As mentioned in §1, Objective C and C++ implement object oriented extensions to C in incompatible ways. In order to embed a C++ implemented model in Swarm, one has to write a "fat" interface to the model in the form of C language functions calls (declared with the `extern "C"` keyword). This is tedious and error prone, and unless one does it comprehensively for every conceivable aspect of the C++ model, the result is that the model is a black box, into which a Swarm probe cannot peer. Such a comprehensive function interface also dramatically pollutes global namespace.

The object descriptor technology described in the previous section provides a solution to this problem. Instead of emitting pack statements, the class analyser emits a function `objc_class` for each data type. This function builds up the internal representation of an Objective C class object. We also provide a `swarmclass` base class from which to derive classes in *EcoLab*. This provides the static `isa` pointer member, which all Objective C objects implicitly define, that points to the class object. This needs to be initialised, which can be handled via a macro definition

```
#define makeswarm(x) x::isa=objc_class(x)
```

The function `objc_class` can take care of defining message handlers corresponding to C++ methods — at least to a point. In C++, methods can be overloaded, but in Objective C they cannot.

The converse problem is that of accessing Objective C objects. It is hopeless trying to access members, without writing a similar parser of Objective C code. However, sending a message to an Objective C object is relatively simple, as these can be constructed at runtime from strings, much as is done presently with the TCL interpreter.

So all we need is for the `swarmclass` base class to define a general message sender routine, rather like the stream class.

## 5. Parallel Programming

*EcoLab* currently supports parallel distribution of computation through distributing the model in a cellular fashion (Standish 1998a). Each cell is conceptually a cell of a spatial grid, but may in fact be any *EcoLab*-like model that loosely interacts with the other cells. The `tcl++` class supports [7] based parallelism by providing macros that cause the C++ code of a TCLCMD[8] to execute on all threads simultaneously, and propagating the TCL variables accessed from the slave threads. The `global` class supports a grid layout of cells of arbitrary dimensionality. This grid is distributed across the threads at initialisation time, and not altered during the calculation.

In practice, there are a couple of problems with the current implementation. Firstly, the global class concatenates all cells within a subgrid into a single list of arrays. This makes sense if the most computationally expensive portion of the code is the update step of the differential equation (called `generate` in *EcoLab*). However, it turns out that most often one needs to work in the cell view, and the resulting work of extracting a single cell and replacing it tends to dominate code times, as well as reducing the readability of the code. What we need is a variant of the `TCLCMD` macro that presents a cell-based view of the model. This will need to be given a neighbourhood template to ensure that neighbouring cells are copied to that thread, and available to the code within the `TCLCMD`. The cell based version of the model will store the individual cells discretely, rather than concatenating them as is current practice.

Secondly, there is no mechanism for rebalancing the load dynamically. For the *EcoLab* model over space, this is not a particularly severe problem, as each thread will contain a number of cells, and over the large sweep of time, the work will be moderately well balanced if the cells are evenly distributed. However, for other types of agent-based models, the work could easily get unbalanced, and/or the agents are mobile within a spatial environment. In this scenario, a capability of migrating objects between threads is essential. In order to access objects located on other threads, proxy objects need to be created to represent the remote object.

### *5.1. Distributed Objects in Objective C*

Whilst the object nature of Objective C appears to offer a simple form of concurrency, a problem occurs in that each object is referred to by a pointer. This implies a single address space in which all threads run, limiting parallelism to the shared-memory (or SMP) variety. Another problem to address is that of the return value of a message invocation. In order to exploit parallelism, the message call must return immediately, either returning a void value, or a bland object value such as “self” (pointer to object receiving message). Similar considerations apply to arguments passed by reference, that are modified by the routine. This problem leads to the concept of a *proxy* — in this instance a placeholder for the return value. One ought to be able to query this proxy to determine if its value has been calculated. Any subsequent use of the return value ought to block until the routine computing the return value has finished.

[7]MPI

[8]The TCLCMD macro installs the block of C++ code following the macro as a command in the TCL interpreter, that can be called from the script being executed.

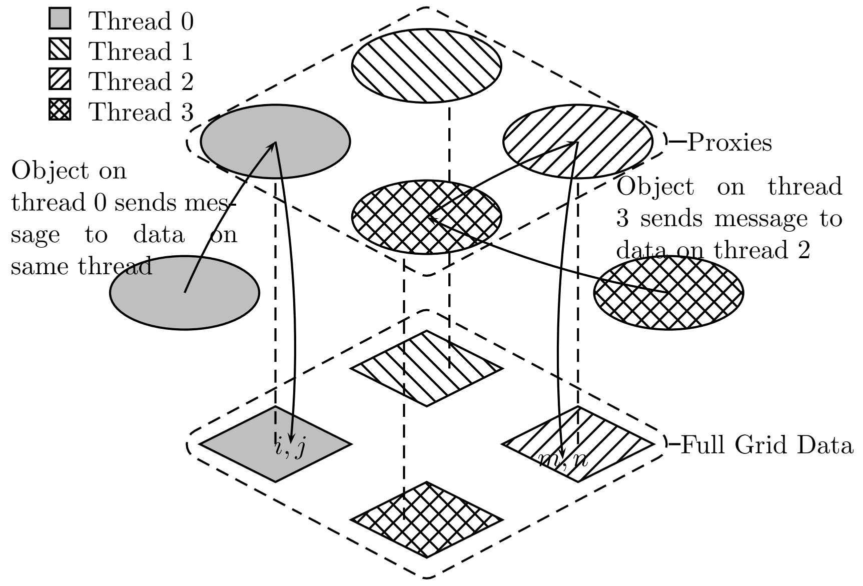
Thread 0
Thread 1
Thread 2
Thread 3
Proxies
Object on thread 0 sends message to data on same thread
Object on thread 3 sends message to data on thread 2
$i, j$
$m, n$
Full Grid Data

Figure 1: Schematic of Distributed Grid Object, illustrating the communication amongst proxies and objects

This concept of a proxy allows one to escape the need to have single address space for the objects to reside. The proxy can refer to an object residing in a different thread's address space. When a message is sent to the proxy, it will forward the message onto the remote object that it is a proxy for. This uses the Objective C `forward` mechanism — if no method exists for a particular message, prior to an error being raised, a `forward` message is sent to the object, which then allows the proxy object to forward the message onto its remote client.

If data is distributed naturally in a grid (eg in the Swarm heatbugs application, the agents live in a 2-D heat field), then proxies are required to access data that exists on other threads. Figure 1 shows how the various proxies and objects will communicate with each other.

## *5.2. Distributed Objects in C++*

C++ present special problems for creating proxies to remote objects because of its static binding to member functions. There is no equivalent functionality to the `forward` mechanism of Objective C. Since we have already described a mechanism for wrapping a C++ object to turn it into an Objective C object in §4, this provides one method for implementing distributed objects. Alternatively, a variant of the Object Descriptors technique described in §3 could be used to generate proxies for objects existing remotely. However, there is also an open source project called [9], which provides for a concept called a *global pointer* (in essence a type of proxy). Remote method invocation occurs by dereferencing the global pointer through the `->` operator. HPC++ is built on top of the Globus parallel communications library. Clearly, the HPC++ option should be explored first, either by directly using HPC++ to compile *EcoLab*, or alternatively studying the system to extract out important design concepts.

[9] *HPC++*

## 6. *EcoLab* 4 project

*EcoLab* 4 will be a major rewrite of the *EcoLab* 3 codebase. Of the 3 main class libraries, `tcl++` will require minor modifications, and `arrays` needs to be reimplemented using C++ templates and/or the [10], technologies that have matured since `arrays` was implemented, but is otherwise independent of most of the issues discussed in this paper.[11] The `globals` class will be replaced completely by the object descriptor technology described in §3. At the time of writing, the source code parser has been written, and tested against an implementation of serialisation. The next step will be to do a partial *EcoLab* model implementation to prove the concept in reality.

There are a number of independent subprojects that can be worked on by interested people. A major task at present is to set up an open source project management system, such as [12], to track changes made by multiple programmers. If you are interested in contributing programming time to this effort, please contact me. Information will also be available from the [13].

## References

[Bedau et. al. 1998] Bedau M. A., Snyder E., and Packard N. H. (1998), "A classification of long-term evolutionary dynamics", in *Artificial Life VI*, C. Adami, R. Belew, H. Kitano, and C. Taylor (eds), pages 228–237, MIT Press, Cambridge, Mass.. .

[Newman 1997] Newman M. E. J. (1997), "A model of mass extinction" J. Theo. Bio. **189** 235–252.

[Ousterhout 1994] Ousterhout J. K. (1994), *TCL and the Tk Toolkit*, Addison-Wesley.

[Standish 1999] Standish R. K. (1999), "Statistics of certain models of evolution" Phys. Rev. E **59** 1545–1550.

[Standish 2000a] Standish R. K. (2000), "The role of innovation within economics" in *Commerce, Complexity and Evolution*, W. Barnett, C. Chiarella, S. Keen, R. Marks, and H. Schnabl, (eds), volume 11 of *International Symposia in Economic Theory and Econometrics*, pages 61–79. Cambridge UP.

[Standish 1998a] Standish R. K. (1999), "Cellular Ecolab" Complexity International, **6** `http://www.csu.edu.au/ci`.

[Standish 1998b] Standish R. K. (1999), "Econolab" Complexity International, **6** `http://www.csu.edu.au/ci`.

[Standish 1994] Standish R. K. (1994), "Population models with random embryologies as a paradigm for evolution" Complexity International, **2** `http://www.csu.edu.au/ci`.

[Standish 2000b] Standish R. K. (2000) "An Ecolab perspective on the Bedau evolutionary statistics" in *Alife VII: Proceedings of the Seventh International Conference*, M. A. Bedau, J. S. McCaskill, N. H. Packard, and S. Rasmussen, (eds), pages 238–242, MIT Press, Cambridge, Mass.

[10] Standard Template Library

[11] Consequently, this reimplementation may not take place for the 4.0 release

[12] Aegis

[13] *EcoLab* website