EcoLab Documentation

Russell K. Standish

September 2, 2016

Version: 5.36

Abstract

EcoLab is an object oriented simulation environment that implements an experiment oriented metaphor. It provides a series of instruments that can be coupled together with the user's model (written in C++) at run-time in order to visualise the model, as well as support for for distributing agents over an arbitrary topology graph, partitioned over multiple processors and checkpoint/restart support. EcoLab was originally developed to simulate a particular model (the EcoLab model) of an abstract ecology[7]. However, several other quite different models have been implemented using the software, demonstrating its general purpose nature.

Contents

1 License

EcoLab is distributed as unrestricted public domain open-source software, which you can use as you wish. The netcomplexity module makes use of *nauty*, which has license restrictions documented in include/nauty.h. Commercial or military applications of EcoLab must make do without netcomplexity functionality.

2 Making EcoLab

Once you have unpacked the gzipped tar file, you should be able to make EcoLab by running "make" or "gmake" in the top level directory. EcoLab needs GNU make (which available by default on Linux, Cygwin/Windows or MacOS, but can be installed from sources on most operating systems if needed). If you have a multiprocessor, or multicore computer, you can speed up the build time by spreading the compilation over multiple processors using the `-j` option to `make`. Currently EcoLab has been developed against TCL/Tk 8.5 and gcc 4.5, but should build with other versions.

EcoLab's Makefile searches for the software it needs, and sets Makefile flags appropriately depending on what it finds. The only software that is an absolute requirement is TCL, however it will also use Tk, BLT (version 2.4.z), Cairo, zlib, readline, XDR, UNURAN or GNUSL, Berkley DB or gdbm if available.

For the plotting widgets, EcoLab will use Cairo by preference - if you wish to use the older BLT-based widgets, specify BLT=1 on the makefile build line. If Cairo is not installed on the system, Ecolab will fall back to using BLT, and if that is not available either, will not provide the plotting widgets.

For parallel programming, it also uses MPI and ParMETIS. EcoLab's Makefile searches for the software in `$HOME/usr`, then `/usr/local` and finally `/usr`. When installing 3rd party libraries, please install them in either `$HOME/usr` or `/usr/local`, usually by specify the `--prefix` option to the "configure" command of the package.

You can override the default settings by passing options to make. In the following table, you can define an option by specifying (eg MPI=1) on the make command line, or undefine it using (eg MPI=).

DEBUGGING	Turns on -g, and assertions
MEMDEBUG	replaces `new` and `delete` with a version that tracks and reports memory usage
PROFILING	logs times executed by each TCL command (§15)
GCOV	Prepares executable for test coverage analysis
VPROF	Prepares executables for use by the vprof[1] tool
MPI	Enable distributed parallel version
OPENMP	Enable the use of OpenMP shared memory parallel constructs
PARALLEL	Enable the use of automatic parallelisation (Intel compiler)
XDR	Enable use of XDR (§13.2) for checkpoints and client-server. On some systems, XDR needs to be disabled for correct compilation in MPI mode. This bug should be fixed in this version of *EcoLab*.
GCC	Force use of gcc compiler.
NOGUI	Disables all GUI functionality.
BDB	Use Berkley database from Sleepycat for the cachedDBM §22) class.
UNURAN	select UNURAN random number library
GNUSL	select GNUSL random number library
PREFIX	installation location for "make install". Defaults to `$HOME/usr/ecolab`.

EcoLab will use the UNURAN non-uniform random number library[2] or the GNU Scientific Library[3] if available. Note that the author uses UNURAN, so GNUSL is typically not tested. If neither library is available, a limited library based on the libc `rand()` function is used instead. The Make flags `UNURAN` and `GNUSL` can be used to override the defaults.

Please note the following point when installing unuran (tested against unuran 1.6.0):

- EcoLab assumes that if PRNG is available on the system, then UNURAN will use it. However, UNURAN will not detect PRNG if it is installed in `$HOME/usr`. If you have a linkage error complaining about missing prng functions, either reinstall UNURAN with the –with-urng-prng option to its configure script, or compile EcoLab with the "`PRNG=`" Makefile option. The former is the preferred option, as it allows runtime configuration of the uniform random generator via PRNG's string interface.

The other options are relatively straight forward. Define `MPI` if you wish to run *EcoLab* in parallel using the MPI message passing library. You will also need to install the ParMETIS[4] library.

[2]http://statistik.wu-wien.ac.at/unuran/
[3]http://www.gnu.org/software/gsl/
[4]http://www-users.cs.umn.edu/ karypis/metis/parmetis/

The standard build will also build in the example models directory. You may also do a `make install`, which will install the *EcoLab* software into the directory given by the `PREFIX` variable in the top level Makefile. By default, this is ~`/usr/ecolab`. You can change this value to something else (eg /usr/local/ecolab) by giving the command `make PREFIX=/usr/local/ecolab install`. After that, you can compile the `models` directory completely independently of the rest of the distribution — use the example Makefile in the `models` directory as a template. If you have installed *EcoLab* in a non-standard location, you may need to modify the `ECOLAB_HOME` Makefile variable to point to the installed directory.

3 Using Ecolab on Windows

The preferred way of using *EcoLab* on windows is to use the Cygwin posix emulation environment[5], which has most the dependent packages available as installable options. However *EcoLab* can also be built using the MinGW compilers, although you will need some sort of posix-like shell, eg Cygwin, or Msys to do the actual build on Windows. EcoLab is, however, validated to build on MXE[6], a cross-compiler environment based on MinGW to build Windows executables using Linux. Check out my fork of MXE[7] which may contain additional packages needed for *EcoLab*, that have not yet been integrated into the MXE master branch.

3.1 Cygwin

Cygwin[8] is a port of the GNU development environment to 32 bit Windows (Windows 95/98/ME and Windows NT/2000/XP), and runs fine in 64 bits too. As such, it provides as near to Posix like environment as is possible in Windows. There are a few hints needed to get Ecolab compiled and running under Cygwin.

- When installing Cygwin, select the following additional packages:

  Devel gcc-g++ and make

  Libs libcairo-devel, libdb4.8-devel, gsl-devel, libreadline-devel, zlib-devel,

  Tcl tcl-tk-devel

  X11 xorg-server and xinit

- Optionally install the PRNG/UNURAN libraries by building them from source code. UNURAN is required for running the jellyfish model.

[5]http://cygwin.com
[6]http://mxe.cc
[7]https://github.com/highperformancecoder/mxe
[8]http://cygwin.com

- Unpack Ecolab and do `make` in top level directory.

EcoLab must be used under X11 on Cygwin. To start the X11 server, type "startx" at the Cygwin console. X11 programs run under this server appear to be running natively under Windows — its quite neat!

Once the X server has started, it will pop another shell window from which you can run Xwindows programs. `cd` to the `models` directory, and run the example ecolab script as `./ecolab ecolab.tcl`. The `#!` mechanism does not work with Cygwin!

3.2 MXE

To build EcoLab using MXE, you first need to build MXE by cloning the mxe repository, and building everything:

```
git clone https://github.com/highperformancecoder/mxe.git
cd mxe
make
PATH=$PATH:`pwd`/usr/bin
```

Now you can build ecolab by building in the toplevel source directory:

```
make MXE=1
```

If you need to build additional third party packages, such as unuran, install these into `$HOME/usr/mxe` It is generally quite easy to build packages for MXE if they use autogen, cmake or qmake Makefile generators.

To run a MXE-built ecolab executable, copy the entire Ecolab source directory with the MXE binaries to your windows system. Then run `install.bat`, which installs the TCL libraries required to support *EcoLab* in the standard APPDATA location on Windows. Then you can run *EcoLab* as usual from any command line tool (Cygwin not required, X-windows not required).

Whilst MXE can be used to build *EcoLab* models for running, using Cygwin is a lot simpler. However, MXE is useful for building standalone applications that use the *EcoLab* library, such as Minsky[9].

4 Using Ecolab under Mac OSX

EcoLab compiles out of the box on OSX using the Mac OSX Dev Tools package, which is available for free download from http://connect.apple.com You will also need a copy of TCL/Tk - the ActiveTCL works well, although building TCL/Tk from source is also possible. If you do the latter, choose the unix build directory, not the macosx one.

[9] http://minsky.sf.net

You need to make a choice whether you want to run EcoLab as a native Aqua application, or as an X-windows application under the XFree86 server software. Specify `--enable-aqua` on the Tk configure line to build for Aqua, rather than X11. On the EcoLabbuild line, specify `AQUA=1` on the make command line.

You may also use MacPorts to easily build the prerequisite packages. EcoLab's Makefiles will search for Macports software located in `/opt/local` prior to searching system locations. Note that if you install the MacPorts version of Tk, it uses X11 only, Aqua mode is unavailable.

4.1 Compiling and running EcoLab for Aqua

Build EcoLab with `AQUA=1`. In the models directory, an "application bundle" is created for each model, for example `ecolab.app`, using the utility script mkmacapp. If you try to run EcoLab scripts in the usual way, you will get an error message:

```
SetFrontProcess failed,-606
```

Instead, you must invoke EcoLab in a rather cumbersome way:

```
ecolab.app/Contents/MacOS/ecolab ecolab.tcl
```

To make it a little easier, the `macrun` script is provided to do this for you — try

```
macrun jellyfish.tcl lakes/OTM
```

5 Structure of the EcoLab Simulation System

EcoLab is built on the following components:

tcl++ which provides bindings to the TCL scripting language,

TCL_obj which exposes the internals of C++ objects to TCL

pack performs serialisation of objects, for checkpoint, and client/server applications

ClassdescMP parallel programming support

Graphcode provides an abstraction of objects moving on a distributed graph.

arrays which implements dynamic arrays used in a number of EcoLabmodels

analysis/Xecolab which provides a number of generic instruments for observing the EcoLab models

random a thin abstraction on random number generators, kindly supplied by UNURAN or GNUSSL (as preferred)

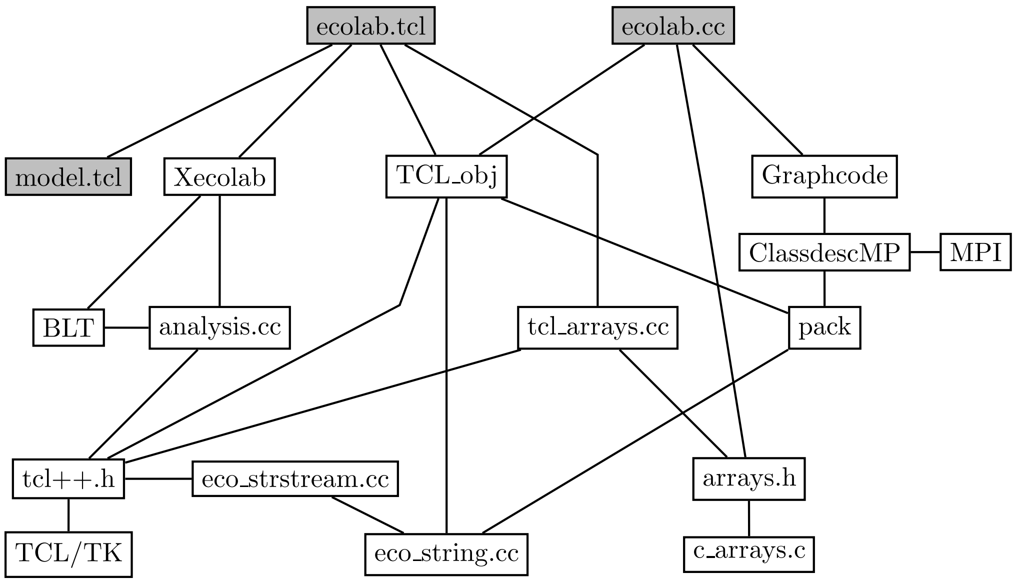


Figure 1: Structure of the *EcoLab* Simulation System for the *EcoLab* model. Shaded boxes are specific to the *EcoLab* model, and are replaced by equivalent modules for other models.

cacheDBM Persistent object map

eco_strstream classdesc stringstream class

eco_hashmap hashmap — using either std::map or TCL's hash map

netcomplexity Network complexity measure

C++ Standard Library (of course!)

The *EcoLab* model itself is defined in the model specific file `ecolab.cc`. To define another model, replace this file by another one with similar functions. An example of this is `jellyfish.cc`. See §8 for more details.

The whole computation is constructed from a TCL[5] script. Example scripts include `ecolab.tcl`, `pred-prey.tcl` and `console.tcl/engine.tcl` for a sample client/server system.

6 Constructing an experiment

An experiment consists of a TCL[5] script, which binds the various elements used (the model system, input parameters, instruments) into an experimental run. An example experiment is `ecolab.tcl`. This is an executable script — once you have made *EcoLab*, you can run this script.

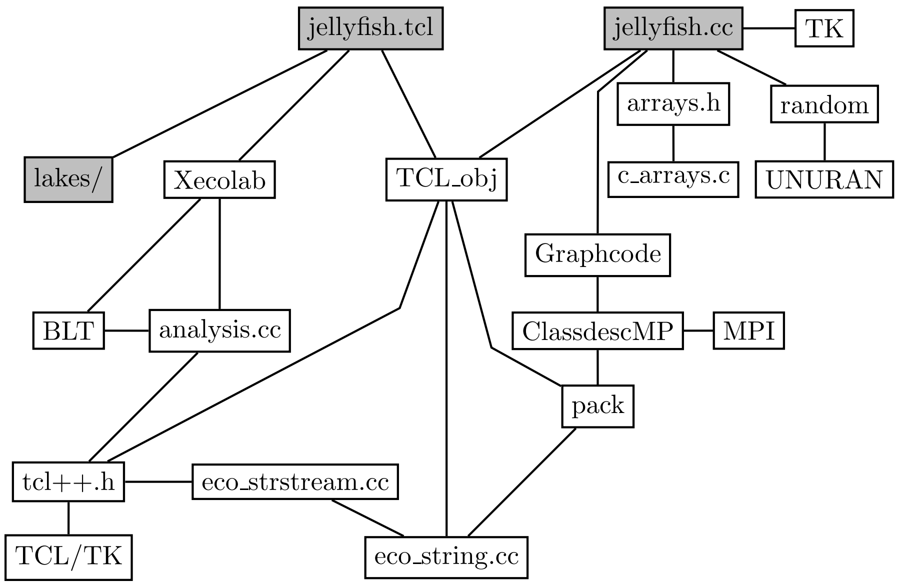


Figure 2: Structure of the *EcoLab* Simulation System for the Jellyfish model. Shaded boxes are specific to the Jellyfish model, and are replaced by equivalent modules for other models.

The script consists of several parts — the first being a simulation loop which steps the model through the generate and mutate operators, then updates the various instruments (`display, plot, connect_plot` etc.) There is a `running` flag which controls whether the simulation is running or not. This is used by the **run** and **stop** buttons to control the execution of the simulation.

The other parts of the experimental script have been broken into separate TCL files — model.tcl contains the input parameters to the model, and Xecolab.tcl has the TCL code relating to the X-windows interface.

6.1 Input parameters

EcoLab exports all the members of the object named in the `make_model` macro to the TCL environment. For example, if the model object is called `ecolab`, accessor TCL commands for the members are created — `ecolab.sp_sep` to access the `sp_sep` member of the ecolab object. Calling the accessor command with no arguments returns the current value of the member. Calling it with an argument sets the value of the member.

The `use_namespace` (§6.3) command can be used to dump all model commands into global namespace. This unclutters the setting of model parameters considerably.

An alternative technique for getting TCL data into your model is to use a `tcl_var` (§15.3). Of the two methods discussed here, the former is recommended.

6.2 The main button bar

The main button bar has five predefined buttons, **quit** (which causes the application to quit), **run**, which calls the `simulate` procedure, and **stop** which sets the `running` flag to zero, suspending the experiment. **Command** calls the (§6.4), allowing you to interact directly with the TCL interpreter. **Object Browser** is a widget allowing you to graphically drill down into the objects defined in the model.

Additionally, there are two user defined buttons, which can have functions bound to them by adding TCL code after the `GUI` command like the following example:

```
GUI
.user1 configure -text condense -command condense
```

6.3 use_namespace

The `use_namespace` *name* TCL command searches for all TCL commands of the form *name.x*, and creates a new TCL command *x* that calls *name.x*. This allows a simple "dump" of *EcoLab* model methods and instance variables into TCL's current namespace. The term "namespace" in `use_namespace` is unrelated to TCL's namespace concept.

Note that `use_namespace` does not override an any existing commands in the current namespace. Imagine what havoc an instance variable called "proc" would make! Thus it may be necessary to refer to an instance variable or method by its fully qualified name fro TCL scripts to get the desired behaviour.

6.4 Command Line Interpreter

The command line interpreter allows you to type any TCL command at the console, rather like the wish tool from Tk. Unlike wish though, the GNU readline library is employed, so expect the usual command and history editing functionality you expect from bash. It is simply invoked by the TCL command `cli`.

If the readline library is not available, then `cli` will not provide command line editing. However, you can still supply a script to *EcoLab*'s standard input.

TCL is not a thread aware system (since it runs on Windows and MacOS), so when the command line interpreter is running, nothing else is — eg GUI widgets or the model. This restriction may be lifted in the future on Posix compliant systems.

6.5 Object Browser

The object browser is a drilldown widget for probing the values of model variables. Since model variables are implemented as TCL commands, this tool lists all TCL commands. Commands containg a "." in their name belong to objects, so these commands are grouped and labelled in blue. TCL namespaces are grouped and labelled in green. Clicking on this blue group opens another window containing just that group of commands. Clicking on a green name drills down into a namespace. Clicking on a red command runs it, and the result is displayed next to it — if this is a Tcl_obj instance variable, the result is the value of the variable. You can specify an argument to the command in the **Args** input box.

Shift clicking allows you to select a range of commands to execute, and Control clicking allows a non-contiguous selection to be made. This is more useful for the **Repeat** option, which repeats the selected commands every second, allowing you to dynamically follow a model's variables.

Right mouse click on any item brings up a menu of options.

- "hide", which removes that command from the display. In this way, you can customize what's in the display. This is particularly useful when using the **Repeat** option.

- "plot" Feed the value of the variable into a plot widget, to track its value over time. Note that the x access in this case is approximate wallclock time — it may have no connection whatsoever on model time. It is, however, a convenient diagnostic for model exploration and debugging.

6.6 Instrumentation

Currently four instrumented widgets are included in the *EcoLab* distribution. The first time these widgets are called, they instantiate themselves in a separate window. This follows the philosophy that initialisation should be a transparent operation. A number of these widgets depend on the BLT toolkit, so it is wise to include BLT at the compilation stage. Each widget is independent, and so multiple instruments can be operating at the same time.

Each such widget creates a new namespace x, the name of which is either supplied by the user, or generated from the arguments. Also created is a window heirarchy starting with $.x$. The widget command returns the namespace name which can be captured for future reference.

One of the uses of this namespace is to call the widget specific `print` command, which dumps a postscript representation of the widget to a named file. For example, a plot window start with `plot nsp ...` can be printed to "plot.ps" using

```
nsp::print plot.ps
```

6.6.1 Plot

Plot a number of TCL variables

Usage:

`plot` *plotname* [`-title` *title string*] *x y1* [*y2...*]

The arguments can be TCL variable names, and the names are used as labels in the graph. If you just supply numerical values, then the variables will be just names arbitrarily x, y1, y2 etc. *plotname* labels the particular widget, so multiple plots can be performed simultaneously.

The plot can be zoomed by selecting a rectangular region using the left mouse button. The right mouse button reverts to the original scale.

plot supports the following methods:

- *plotname*::print *filename* — produce a postscript representation to a file
- *plotname*::clear — clear the plot

6.6.2 Histogram

Usage:

`histogram` *plotname* [`-title` *title string*] *list of values*

This command adds the values to a histogram plot. A record of all data values so far added to the histogram is stored in the file *plotname*`.dat`. This allows automatic recalculation of the histogram's bins whenever a data value goes out of bounds. The number of bins is also dynamically controlled by the scale widget on the side. The histogram is recalculated next time `histogram` is called after the number of bins has been altered. This can be a time consuming business, so it is recommended that the experiment is suspended before changing the number of bins.

There is also the option of plotting the histogram with logarithmic scales.

histogram supports the following methods:

- *plotname*::print *filename* — produce a postscript representation to a file
- *plotname*::clear — clear the plot
- *plotname*::outputdat *filename* — output histogram to a text file
- *plotname*::xlogscale — toggle logarithmic x axis
- *plotname*::ylogscale — toggle logarithmic y axis
- *plotname*::setnbins *nbins* — set the number of bins

6.6.3 Display

Usage:

`display` *model_var identity_var*

This command plots each component of the *model variable* as a function of time. The *identity variable* is used to ensure continuity of the curves, and colouring them to aid identification of species. The index of the component is not a good proxy for this, as the index of a particular species may have changed through a `condense` operation.

Traditional *EcoLab* usage of this command is

```
display ecolab.density ecolab.species
```

The colour of the lines used are controlled by the `palette` TCL variable.(§6.7) This is a list of X-window colours to use.

The namespace for this widget is constructed by appending the 1st argument to the string "display_". Any unacceptable characters are replaced by '_', so it may take a little experimentation to find the correct namespace name. display supports the following methods:

- display_*model_var*::print *filename* — produce a postscript representation to a file
- display_*model_var*::clear — clear the plot

6.6.4 Connect_plot

Usage:

`connect_plot` *interaction density*

This command displays the connectivity of the interaction matrix. The ecologies are ordered to show up the independent subecologies, and if the `palette` variable (§6.7) has been defined, each of the independent subecology is coloured differently.

The integer array *density* is used to mark rows and columns that have become extinct. The are marked in a wheat colour.

There are two buttons which allow the user to zoom in and out by a scale factor of two. One can also zoom in by clicking with the right mouse button at the location you wish to zoom in on. The plot can also be dragged with the left mouse button to change the view.

The namespace for this widget is constructed by appending the 1st argument to the string "connect_". Any unacceptable characters are replaced by '_', so it may take a little experienmtation to find the correct namespace name. display supports the following methods:

- connect_*interaction*::print *filename* — produce a postscript representation to a file

6.7 Palette Variable

This variable is used by the `display` and `connect_plot` and `plot` commands to define a palette of colours for colouring the species in the instruments. If the TCL variable `palette` is assigned a list of X-windows colours (on many systems, a list of such colours is found in `/usr/lib/X11/rgb.txt`), then the palette class can be used like an array within C++, returning the colour name as a string:

```
palette[i]
```

returns the `i%n`th colour, where `n` is the number of colours in the palette list.

6.8 Making movies

Its actually quite easy to make a movie of an Ecolab run. Each of the base widgets of the instruments has a method called `print` defined in the widgets namespace (§6.6), which will output a postscript representation of the widget to a file. See `gen-move.tcl` for an example, which produced these animated GIFs[10].

Once a series of postscript files are created, you can convert them into GIFs using the `pstoimg`[11] utility that comes with LaTeX2HTML[12], using something like the following[13]:

```
cp *.ps /tmp
for i in *.ps; do pstoimg -gif  $i; done
cp /tmp/*.gif .
```

Finally, use `gifmerge`[14] to produce an animated GIF.

```
gifmerge *.gif >movie.gif
```

An alternative is to produce an AVI file using the `mencoder` software that comes as part of the mplayer[15] package. First you need to prepare a collection of jpeg files:

```
for i in *.ps; do
  f=${i%%.ps}
  gs -sDEVICE=ppm -sOutputFile=$f.ppm -dNOPAUSE -dBATCH -g700x400 -r50  $i
  cjpeg $f.ppm >$f.jpg
  done
```

[10] http://parallel.hpc.unsw.edu.au/rks/ecolab-snaps/example-anim-gifs.html

[11] Requires ghostscript, perl and netpbm to be installed

[12] http://parallel.hpc.unsw.edu.au/htmldocs/latex2html

[13] The current `pstoimg` script doesn't seem to work properly in any directory other than /tmp — I will need to fix it and post a copy for EcoLab users

[14] gifmerge is available from a number of open source repositories

[15] http://www.mplayerhq.hu

You can control the final size of the bitmap using the `-g` option to gs, and control the scale with the `-r` option ("pixels per inch"). Mplayer has problems if the bitmap is too large, and if the dimensions are odd.

Finally, you can create an mpeg1 encoding using mencoder:

```
mencoder -mf on:type=jpeg -ovc lavc -lavcopts vcodec=mpeg1video \
           \*.jpg -o movie.avi
```

Consult the mencoder man page for more details on codec options.

6.9 Auxilliary Commands

6.9.1 get_vars/data_server

Syntax:
model.`get_vars` *server port*
model.`data_server` *port*

This pair implements a client server connection. `data_server` is executed on the compute server, and services any requests coming in on the given port. `get_vars` attaches to the compute server, and downloads the compute server's model variable into the client's model variable. The client can the follow on with the usual instruments for analysing the data. The advantage in this approach is that X-window traffic can be avoided, the total amount of traffic between client and server can be controlled by how often these routines are called. An example setup is located in `console.tcl` and `engine.tcl`. An alternative client/server scenario can be contructed using Tcl sockets, and the console2.tcl/engine.tcl give an example of just transferring the $\boldsymbol{n}$ of the predator-prey example. This has a great deal of flexibility, allowing, for example, messages to be propagated from client back to the server to allow user interactivity into the model

6.9.2 checkpoint/restart

Syntax:
model.`checkpoint` *filename*
model.`restart` *filename*

These commands dump the contents of the model's variables into a file, and correspondingly reload the model's state variables from the file, in order to implement checkpoint-restart for a batch, or long running environment. These commands are defined in the (§13.11), and can be overridden if desired.

By setting the variable *model*.`xdr_check` to 1 (default value is zero), the checkpoint is written out using XDR routines, so the checkpoint file can be restarted on a computer with a different processor.

6.9.3 Trapping signals

`trap` *signal command*
`trapabort` [off]

Arrange for *command* to be executed whenever the signal *signal* is received. *signal* may be specified symbolically (eg `TERM` or `XCPU`) or numerically (15 and 24 in previous example).

`trapabort` arranges for the TCL error handler to be called whenever a segmentation violation, illegal instruction or bus error occurs. Often the error is mild enough, that processing can continue — it is used in particular with the Object Browser. Specifying `trapabort off` turns off this behaviour — you definitely need this disabled when running *EcoLab* in a debugger.

6.10 ecolab_version

This reports the version number of the EcoLab system, eg 4.D22. This version number is also available in the file `version.h` as the macro `VERSION`.

7 Creating New Instruments

The best way to proceed is to copy an existing instrument, and modify it to your needs. The interface to the TCL language can be done almost entirely through the `tcl++` class library (§15). Use the `NEWCMD` macro to instantiate a new TCL command. This may be the complete widget, or merely a component of it. The instrument may be encoded entirely in TCL code, accessing the model's state variables in the usual way. Alternatively, you may write C++ code to access the state variables directly through the model's class defintion.

An alternative approach is taken in `analysis.cc`. A variable that has been registered in the `TCL_obj` database can have a reference set up to it using the `declare` macro, defined in `TCL_obj_base.h`. For example, in the ecolab model, `ecolab.density` is the TCL name for the `density` member of the ecolab model. Then

```
declare(density,iarray,"ecolab.density");
```

declares a variable of type `iarray&` that may be used to refer to this member. Argument 3, the string parameter needn't be a constant string, but can be any string, eg a string passed through `argv[]`.

8 Creating a New Model

The code that explicitly defines the *EcoLab* model is contained in `ecolab.cc`. Several examples of a completely different models is provided with the *EcoLab* distribution. Most of these are described in different papers:

shadow Ecolab model with a neutral shadow model, as described in [8] and [9].

newman A variation of Mark Newman's evolutionary model, described in [6]

webworld An implementation of Drossel *et al.*'s Webworld model, described in [2] and [10].

jellyfish A model of Jellyfish migration in Palauan lakes, in collaboration with Mike Dawson. This model will be described in more detail in §12.

netcomplexity A class providing TCL methods for computing network complexities. Used in the study reported in [11]

A model typically consists of an interface file (.h), which is processed by Classdesc, an implementation file (.cc) and one or more experiment scripts (.tcl).

The model should, as far as possible, be implemented as a single object (which may be a container). Let's call the model JoesFolly. The model class (eg `JoesFolly_t`) is defined in the interface file in a regular C++ fashion. It is simplest if all members of the model class are public, however read the Classdesc chapter (§13.1) for how to handle private members.

Consider whether your model maps naturally to the notion of objects related by a network. In that case, you may find that Graphcode (§14) will effectively distribute your model across multiple processors of an MPI parallel job. Both the spatial EcoLab model, and the Jellyfish model are examples of Graphcode deployment.

Your model class needs to be derived from `TCL_obj_t`. This adds a few extra methods to your class, such as checkpoint/restart, and client/server functionality.

```
class JoesFolly_t: public TCL_obj_t
{
  public:
   int an_instance_var;
   double a_method(TCL_args);
};
```

You then define your model object in the implementation file, and pass this object to the macro `make_model`.

```
#include <ecolab.h>
#include "JoesFolly.h"
#include "JoesFolly.cd"
#include <ecolab_epilogue.h>

JoesFolly_t JoesFolly;
```

```
make_model(JoesFolly);

double JoesFolly(TCL_args args)
{
  double x=args, y=args;
  ...
}
```

The supplied Makefile in the models directory can be used as a template for your own project. It contains rules that generate Makefile dependencies for all include files included with `""`. It will launch classdesc to generate descriptors for all class definitions in `JoesFolly.h`. All public instance variables are visible to TCL, as are methods that take no arguments, or a single TCL_args (§13.11) argument.

Of course these instance variables or members are only actual accessible from TCL if their type `T` has a `operator<<(ostream&,T)` defined. Accessing other types of object will result in a runtime error.

Note that Standard C++ requires functions used by templates be declared prior to templates being defined. Since the .cd files are declaring functions like pack, TCL_obj etc., templates that call these functions must be declared after all .cd files are included, otherwise the templates will not pick up the definitions in the .cd files. This is achieved by including `ecolab_epilogue.h` after the all the .cd files have been included in the .cc file. In fact, now *EcoLab* insists on the presence of this file if classdesc is used, and will generate a link time failure if not provided:

```
undefined reference to '(anonymous namespace)::classdesc_epilogue_not_included()'
undefined reference to '(anonymous namespace)::TCL_obj_template_not_included()'
```

9 Error reporting

EcoLab now uses C++ standard exceptions to report errors — the old `longjmp` mechanism has now gone. When you want to report an error from within your user written commands and methods, throw an object of type `error`, which is defined in `tcl++.h`. `error` has a convenient constructor that works a bit like printf, eg

```
  throw error("%d arguments is too many!",nargs);
```

EcoLab will trap all exceptions in user written routines and methods, and return a `TCL_ERROR` result. If its an exception derived from the standard `exception` class (which `error` is), then it will place the value returned by its `what()` method as the result field of the TCL command. Any other caught exception will report “Unknown exception caught”.

What happens when *EcoLab* traps an exception depends on exactly where it occurs — if it occurs as a result of executing a script, the error is reported on standard output, and *EcoLab* exits with a non-zero return value (useful for writing test scripts, for example). If it occurs while the GUI environment is in operation, then a dialog box pops up, offering information to debug what went wrong. If it occurs as part of the command line interpreter (ie when *EcoLab* is reading its commands from standard input) then the error result is reported on standard output, and execution continues.

When the DEBUGGING flag set at make time, the error constructor will call `abort()`, which can be caught by a debugger.

10 EcoLab coding style

There is no particular *EcoLab* coding style in terms of things like choice of identifier capitalisation, indentation, use of whitespace, naming schemes and the like. Non-syntactical information is always misleading, and I can well advise studying the output of doxygen to work what a particular identifier actually is. For singleton classes, I have a habit of following the Java convention of capitalising the class name, and using a lower case for the object. Similarly, for namespaces, I append a `_ns`.

Within header files, I tend to write more compactly, for instance placing an opening brace on the previous line, or putting several simple statements together on one line. In implementation files I tend to space things out a bit more. Obviously, the idea is to only place members with a few lines of definition in the interface file, otherwise migrate the definition to an implementation file.

More important is information that the compiler can use to enforce correctness and performance. The concept of const-correctness is very important in determining flows of data dependencies, similarly exception correctness is important for determining code flow. Use references instead of pointers whereever possible.

Of crucial importance is a concept known as "Resource Acquisition Is Initialisation". If you need to access some resource such as memory, process or file, put the resource acquisition into the constructor of some object, and the corresponding resource release as the destructor. This has numerous benefits, ranging from eliminating resource leaks to ensuring exeception-correctness.

Much C++ code is written in a style I would call C/Java style. Objects are allocated on the heap using `new`, and users of class libraries must ensure that the corresponding `delete` is called to correctly clean up the opbject when it is destroyed. This style of programming leads to a whole host of subtle problems that RAII avoids.

EcoLab code tends to assume that objects are default constructible, copiable, assignable and serialisable (DCAS). These concepts are heirarchical — a class composed of members satisfying these criteria, also satisfies these criteria (or at least can be arranged to saisfy it through automated technques such as Classdesc). Unless absolutely necessary, try to ensure classes introduced in your *EcoLab* model satisfy DCAS. Pointers are not DCAS, so if you need to use a pointer (eg to use a library having objects pointed to), then consider wrapping the pointer in an RAII style. *EcoLab* provides the `ref` class, which is a DCAS type allowing shared references. `ref` is not suitable for polymorphic data, however *EcoLab* provides `poly{}` type for handling polymorphic objects, that is DCAS. The Boost classes `shared_ptr` and `intrusive_ptr` (which are part of the TR1 standard addition to C++) are DCA, but unfortunately are not serialisable per se. Depending on your need, you may be better off using the *EcoLab* `ref` and `poly` data types instead of the boost routines.

11 The EcoLab Model

The EcoLab model is but one model implemented using the EcoLab software. This section documents the model itself, and may be skipped if your intention is to use EcoLab for other models.

We start with a generalised form of the Lotka-Volterra equation

$$\dot{\boldsymbol{n}} = \mathbf{r} * \boldsymbol{n} + \boldsymbol{n} * \boldsymbol{\beta}\boldsymbol{n}. \tag{1}$$

Here $\boldsymbol{n}$ is the population density, the component n_i being the number of individuals of species i, $\mathbf{r}$ is the difference between reproduction and death, $\boldsymbol{\beta}$ is the interaction matrix, with β_{ij} being the interaction between species i and j, * referring to elementwise multiplication and `mutate` is the mutation operator.

11.1 Lotka-Volterra Dynamics

The most obvious thing about equation (1) is its fixed point

$$\hat{\boldsymbol{n}} = -\boldsymbol{\beta}^{-1}\mathbf{r}, \tag{2}$$

where $\dot{\boldsymbol{n}} = 0$. For this point to be biologically meaningful, all components of $\hat{\boldsymbol{n}}$ must be positive, giving rise to the following inequalities:

$$\hat{n}_i = \left(\boldsymbol{\beta}^{-1}\mathbf{r}\right)_i > 0, \forall i \tag{3}$$

The stability of this point is related to the negative definiteness of derivative of $\dot{\boldsymbol{n}}$ at $\hat{\boldsymbol{n}}$. The components of the derivative are given by

$$\frac{\partial \dot{n}_i}{\partial n_j} = \delta_{ij}\left(r_i + \sum_k \beta_{ik}n_k\right) + \beta_{ij}n_i \tag{4}$$

Substituting eq (2) gives

$$\left.\frac{\partial \dot{n}_i}{\partial n_j}\right|_{\hat{\boldsymbol{n}}} = -\beta_{ij}\left(\boldsymbol{\beta}^{-1}\mathbf{r}\right)_i \tag{5}$$

Stability of the fixed point requires that this matrix should be negative definite. Since the $\left(\boldsymbol{\beta}^{-1}\mathbf{r}\right)_i$ are all negative by virtue of (3), each minor determinant of this matrix is equal to a minor determinant of $\boldsymbol{\beta}$ multiplied by a positive number, stability of the equilibrium is equivalent to $\boldsymbol{\beta}$ being negative definite.

A weaker condition is to require that the system remain bounded with time:

$$\sum_i \dot{n}_i = \mathbf{r}\cdot\boldsymbol{n} + \boldsymbol{n}\cdot\boldsymbol{\beta}\boldsymbol{n} < 0,\ \forall \boldsymbol{n} : \sum_i n_i > N\ \exists N \tag{6}$$

As $\boldsymbol{n}$ becomes large in any direction, this functional is dominated by the quadratic term, so this implies that $\boldsymbol{n}\cdot\boldsymbol{\beta}\boldsymbol{n} \le 0\ \forall \boldsymbol{n} : n_i > 0$. Negative definiteness of $\boldsymbol{\beta}$ is sufficient, but not necessary for this condition. For example, the predator-prey relations (heavily normalised) have the following matrix as $\boldsymbol{\beta}$: $\boldsymbol{\beta} = \begin{pmatrix} -1 & 2 \\ -2 & 0 \end{pmatrix}$ which has eigenvalues $3/2, -5/2$. If we let $\boldsymbol{n} = (x, y), x, y \geq 0$, then $\boldsymbol{n} \cdot \boldsymbol{\beta n} = -2x^2$, which is clearly non-positive for all x.

Consider adding a new row and column to $\boldsymbol{\beta}$. What is condition is the new row and column required to satisfy such that equation (6) is satisfied. Break up $\boldsymbol{\beta}$ in the following way:

$$\left(\begin{array}{ccc|c} \ddots & & & \vdots \\ & \mathbf{A} & & \mathbf{B} \\ & & \ddots & \vdots \\ \hline \cdots & \mathbf{C} & \cdots & \mathrm{D} \end{array}\right) \left(\begin{array}{c} \vdots \\ \mathbf{n_1} \\ \vdots \\ \hline n_2 \end{array}\right)$$

Condition (6) becomes:

$$\mathbf{n_1} \cdot \mathbf{A n_1} + \mathbf{n_1} \cdot (\mathbf{B} + \mathbf{C}) n_2 + D n_2^2 \leq 0 \tag{7}$$

Let
$$a = \max_{n=1} \boldsymbol{n} \cdot A\boldsymbol{n}, \text{ and } b = \max_i B_i + C_i.$$
Then a sufficient but not necessary condition for condition (7) is

$$a n_1^2 + b n_1 n_2 + D n_2^2 \leq 0$$

The maximum value with respect to n_2 is $a n_1^2 - (b n_1)^2/4D$, so this requires that
$$b \geq 2\sqrt{aD} \tag{8}$$

11.2 Mutation

With mutation, equation (1) reads

$$\dot{\boldsymbol{n}} = \mathbf{r} * \boldsymbol{n} + \boldsymbol{n} * \boldsymbol{\beta n} + \texttt{mutate}(\boldsymbol{\mu}, \mathbf{r}, \boldsymbol{n}). \tag{9}$$

The difficulty with adding mutation to this model is how to define the mapping between genotype space and phenotype space, or in other words, what defines the *embryology*. A few studies, including Ray's Tierra world, do this with an explicit mapping from the genotype to to some particular organism property (e.g. interpreted as machine language instructions, or as weight in a neural net). These organisms then interact with one another to determine the population dynamics. In this model, however, we are doing away with the organismal layer, and so an explicit embryology is impossible. The only possibility left is to use a statistical model of embryology. The mapping between genotype space and the population parameters $\mathbf{r}$, $\boldsymbol{\beta}$ is expected to look like a rugged landscape, however, if two genotypes are close together (in a Hamming sense) then one might expect that the phenotypes are likely to be similar, as would the population parameters. This I call *random embryology with locality.*

In the simple case of point mutations, the probability $P(x)$ of any child lying distance x in genotype space from its parent follows a Poisson distribution. Random embryology with locality implies that the phenotypic parameters are distributed randomly about the parent species, with a standard deviation that depends monotonically on the genotypic displacement. The simplest such model is to distribute the phenotypic parameters in a Gaussian fashion about the parent's values, with standard deviation proportional to the genotypic displacement. This constant of proportionality can be conflated with the species' intrinsic mutation rate, to give rise another phenotypic parameter $\boldsymbol{\mu}$. It is assumed that the probability of a mutation generating a previously existing species is negligible, and can be ignored. We also need another arbitrary parameter ρ, "species radius", or `ecolab.sp_sep`, which can be understood as the minimum genotypic distance separating species, conflated with the same constant of proportionality as $\boldsymbol{\mu}$.

In summary, the mutation algorithm is as follows:

1. The number of mutant species arising from species i within a timestep is $\mu_i\alpha_i n_i/\rho$. This number is rounded stochastically to the nearest integer, e.g. 0.25 is rounded up to 1 25% of the time and down to 0 75% of the time.

2. Roll a random number from a Poisson distribution $e^{-x/\mu+\rho}$ to determine the standard deviation σ of phenotypic variation.

3. Vary $\mathbf{r}$ according to a Gaussian distribution about the parents' values, with $\sigma\alpha_0$ as the standard deviation, where α_0 is the range of values that $\mathbf{r}$ is initialised to, ie α_0=`ecolab.repro_max`− `ecolab.repro_min`

4. The diagonal part of $\boldsymbol{\beta}$ must be negative, so vary $\boldsymbol{\beta}$ according to a log-normal distribution. This means that if the old value is β, the new value becomes $\beta' = -\exp(\log_e(\beta)+\sigma)$. These values cannot become arbitrarily small, however, as this would imply that some species make arbitrarily small demands on the environment, and will become infinite in number. In *EcoLab*, the diagonal interactions terms are prevented from becoming larger than $-r/(.1 * \texttt{INT\_MAX})$.

5. The offdiagonal components of $\boldsymbol{\beta}$, are varied in a similar fashion to $\mathbf{r}$. However new connections are added, or old ones removed according to $\lfloor 1/r \rfloor$, where $r \in (-2,2)$ is chosen from a stepped uniform distribution

$$P(x) = \begin{cases} 0.25(1-g) & \text{if x} \leq 0 \\ 0.25(1+g) & \text{if x} > 0 \end{cases}$$

where $g \in [-1, 1]$ (default of 0) is specified by the TCL variable `generalization_bias`. The values on the new connections are chosen from the same initial distribution that the offdiagonal values where originally set with, ie the range `ecolab.odiag_min` to `ecolab.odiag_max`. Since a in condition (8) is computationally expensive, we use a slightly stronger criterion that is sufficient, computationally tractable yet still allows "interesting" non-definite matrix behaviour namely that the sum $\beta_{ij} + \beta_{ji}$ should be nonpositive.

6. $\boldsymbol{\mu}$ must be positive, so should evolve according to the log-normal distribution like the diagonal components of $\boldsymbol{\beta}$. Similar to $\boldsymbol{\beta}$, it is a catastrophe to allow $\boldsymbol{\mu}$ to become arbitrarily large. In the real world, mutation normally exists at some fixed background rate — species can reduce the level of mutation by improving their genetic repair algorithms. In *EcoLab*, this ceiling on $\boldsymbol{\mu}$ is given by the `ecolab.mut_max` variable.

11.3 Input Parameters

The model's parameters are set by TCL variables in `model.tcl`. The actual data structures of the model are initialised the first time the model's generate step is called. An example input set is:

```
# initial condition
ecolab.species {1 2}
ecolab.density {100 100}
ecolab.create {0 0}
ecolab.repro_rate {.1 -.1}
ecolab.interaction.diag {-.0001 -1e-5}
ecolab.interaction.val {-0.001 0.001}
ecolab.interaction.row {0 1}
ecolab.interaction.col {1 0}
ecolab.migration {.1 .1}


# mutation parameters
ecolab.mutation {.01 .01}
ecolab.sp_sep .1
ecolab.repro_min -.1
ecolab.repro_max .1
ecolab.odiag_min -1e-3
ecolab.odiag_max 1e-3
ecolab.mut_max .01
```

Model variables define a TCL command of the same name as they appear in the C++ source. So in the *EcoLab* model, the C++ object `ecolab` defines a set of TCL commands such as `ecolab.density` that can be used for setting or querying the values of `ecolab`'s members. If an argument is specified, then that argument is used to set the variable's value, otherwise, the variable's value is returned. Array members in the model are initialised by specifying an TCL list argument to the variables name, and return TCL lists when no argument is specified. The above example starts the ecology off with a single predator and prey (based on `pred-prey.tcl`).

11.4 EcoLab Model commands

11.4.1 generate

This implements the basic Lotka-Volterra equations:

$\dot{\boldsymbol{n}} = \mathbf{r} * \boldsymbol{n} + \boldsymbol{n} * \boldsymbol{\beta}\boldsymbol{n}$

with $\mathbf{r}$ being the reproduction rate and $\boldsymbol{\beta}$ being the interspecies interaction. This is implemented as a single line:

```
  density += repro_rate * density + (interaction * density) * density;
```

This command also increments the timestep counter `tstep`.

An optional argument specifies a number of timesteps to run the generate step. This improves the speed by amortising the real to integer conversion operation over a number of timesteps. The downside is that computation may fail if the problem is ill-conditioned (offdiagonal elements of $\boldsymbol{\beta}$ too large with respect to the diagonal elements).

11.4.2 condense

This command compacts the systems of equations by removing extinct species where $n_i = 0$.

11.4.3 mutate

This applies the point mutations to the system. The precise algorithm is described in §11.2.

11.4.4 migrate

This operator implements migration within cellular *EcoLab*. This updates density values according to the difference with the 4 nearest neighbours: $\boldsymbol{n} += \boldsymbol{\gamma} * (0.25(\boldsymbol{n}_n + \boldsymbol{n}_e + \boldsymbol{n}_s + \boldsymbol{n}_w) - \boldsymbol{n})$, where the n, e, s, w index the north, east, south and west neigbouring cells.

11.4.5 maxeig

`maxeig` returns the maximum eigenvalue of $\boldsymbol{\beta}$. If this number is negative, the equilibrium point is stable, if positive, it is unstable. As reported in Standish (1994)[7], the mutation drives the maximum eigenvalue slightly positive, then instabilities act to push the eigenvalue back to zero. This command requires LAPACK).

11.4.6 lifetimes

`lifetimes` records the timestep when a species passes a threshold (hardwired at 10) in the `create` iarray. If a species has yet to pass the threshold, or has gone extinct, the value in `create` is zero. Upon return, this routine returns the lifetime of the species that have gone extinct. This can then be passed to a histogram routine, or written to a file.

11.4.7 random_interaction

Calls (See §16.2) to randomly initialising the nonzero pattern of the offdiagonal elements. The average number of nonzeros per row is `conn`, and the standard deviation of the number of nonzeros is `sigma`.

11.4.8 set_grid

Synopsis
  `ecolab.set_grid` *x y*
Set up an $x \times y$ grid in spatial *EcoLab*. See `ecolab_spatial.tcl` for an example using this.

11.4.9 get

Synopsis
  `ecolab.get` *x y*
Create TCL method for accessing the internals of cell *x y*. The new commands look like array elements, eg

```
ecolab(1,0).density
```

11.4.10 forall

Synopsis
  `ecolab.forall ecolab.`*command args*
  Run *command* on all cells.

11.5 Spatial Variation

The ecolab model can run in multicellular mode by calling `ecolab.set_grid` from TCL, specifying the dimensions of the grid.. See `ecolab_spatial.tcl` for an example.

Only population density varies between the cells — all other variables are members of the ecolab variable so can be set or queried in the usual way.

The usual ecolab model methods (generate, mutate, condense and lifetimes) can now be called, but operate on the entire grid. A new `migrate` is defined to handle migration between cells. You can also call a method of the ecolab cell on all cells using the `forall` command. For instance, to set all cells to same initial density, use:

```
ecolab.forall ecolab.density [constant $nsp 100]
```

Access to the individual cells can obtain by creating a TCL_obj to refer to it by using the `ecolab.get` method, which creates commands like `ecolab(`x,y`)` to refer to the cell. These can be fed to visualisers in the usual way.

11.5.1 Parallel Execution

Since the cells are pins in a graphcode Graph (§14), they are distributed over parallel processes if available.

Remember to call the `gather` method to ensure node 0 is updated before running a visualiser on global data.

ecolab.gather Bring processor 0's data up to date with the rest of the grid

ecolab.distribute_cells Broadcast processor 0's data out to the rest of the grid.

12 Palauan Jellyfish model

This is a model being developed by Mike Dawson and Russell Standish to model the behaviour of jellyfish in a number of lakes on the island of Palau. Jellyfish are photosynthetic animals, so have have a preference for the sun and avoiding shadows. In this model, the jellyfish are represented by agents that have a position and velocity. If the jellyfish moves into shadow, or bumps into the side of the lake, it will reverse its direction. From time to time, the jellyfish will change direction and speed. The random generators governing these are selectable at runtime through the experimental script. Also, jellyfish do bump into each other. In the model display, a jellyfish will flash green if it bumps into another one.

To run the jellyfish model, run the jellyfish.tcl script (located in the models directory), specifying the lake as an argument, eg:

```
jellyfish.tcl lakes/OTM
```

You can choose whether to compile the 2D version of the model or the 3D version, by (not) defining the preprocessor flag `THREE_D` (see Makefile).

The lake itself is represented by a Tk pixmap, with the blue component representing water. The lake shapes were scanned into a GIF file, and edited with a run-of-the-mill paint program to produce the lakes.

Visualising the lake involved creating a Tk canvas widget, displaying the lake image in it, then overlaying it with shadow lines extending from the pixels lying on the boundary, and finally representing the jellyfish with arrow symbols (to indicate position and velocity).

This model illustrates the use of probes. Mouse clicks in the canvas region are bound to a short method that determines which agent is closest to the mouse position. It then colours that agent red (for tracking purposes), creates a TCL_obj representing that agent, and returns the name back to TCL. TCL then calls the object browser (§6.5) on that TCL_obj. In all, 14 lines of C++ code, and 3 lines of TCL code. The result is very effective.

The jellyfish model is written to be run in parallel using Graphcode (§14). The strategy is effectively a *particle-in-cell* method. The lake is subdivided into a Cartesion grid of cells, and each Jellyfish only needs to consult the cell that it is in, as well as neighbouring cells to determine if it will collide with any other jellyfish.

13 Classdesc

13.1 Object Reflection

The basic concept behind this technology is the ability to know rather arbitrary aspects of an object's type at runtime, long after the compiler has thrown that information away. Other object oriented systems (for example Objective C) use dynamic type binding in the form of an `isa` pointer that points to a compiler generated object representing the class of that object. This technology can also referred to as *class description*, as one only needs to generate a description of the object's class, then ensure the object is bound to that description, hence the name *classdesc*[3].

In C++, it is not necessary to incur the overhead of an `isa` pointer, as one can bind an object's type to an overloaded instance of a function call at compile time.

A *descriptor* is a template function `D`

```
template <class T>
void D(D_t&  t, const classdesc::string& d, T& a);
```

The *D t* argument allows for state to be maintained while the descriptor is recursively applied to the data structure. If state is not needed for the descriptor, a "null" object should be provided for passing through.

Users can specialise the descriptor to handle different types. The classdesc descriptor, however does not specialise the descriptor directly, but rather specialises a functor template:

```
template <class T> struct access_D
{
   void operator()(D_t&, const string&, T&);
};
```

There are two advantages of using functional classes:

1. Partial specialisation is available for template classes, but not template functions

2. It is simpler to specify friend access to a functional class for those descriptors needing to access (see §13.1.6)

The generic descriptor then calls `operator()` of the appropriate access class, eg:

```
template <class T>
void pack(classdesc::pack_t& t, const classdesc::string& d, T& a)
{classdesc::access_pack<T>()(t,d,a);}
```

but may optionally perform some additional pre/post-processing if sensible for the particular descriptor.

The `classdesc` package comes with several descriptors already implemented:

pack/unpack , which implements *serialisation*

xml_pack/xml_unpack , which serialises to/from an XML description of the object.

json_pack/json_unpack , which serialises to/from a JSON description of the object.

dump , which writes an ascii representation of the object to a `std::ostream` object

javaClass , which generates a Java interface to C++ objects using JNI.

`pack` will be documented in more detail later, but a simple overview is that the `pack_t` object is a simple reference to some binary data:

```
struct pack_t
{
  char *data();
  const char *data() const;
  size_t size();
} buf;
```

A call to `pack(buf,""",foo)` pushes a binary representation of the object `foo` (regardless of its type) into buf. The inverse operation is called `unpack`. Syntactically, we may also use the `<<` operator for the same purpose:

```
buf << foo << bar;
fwrite(buf.data(),buf.size(),1,f);
pack_t b1(buf.size());
fread(b1.data(),b1.size(),1,f);
b1 >> foo1 >> bar1;
```

This code has made a copy of `foo` and `bar`, but with the data going via a disk file.

13.1.1 Using Classdesc: Method 1, inlining

The inlining method is conceptually the simplest, and practically the easiest method to use. Start with the following rules in your *GNU Make* `Makefile`:

```
.SUFFIXES: .c .cc .o .d .h .cd
ACTIONS=pack unpack
OBJS= ...
```

```
include $(OBJS:.o=.d)

.c.d:
	gcc $(CFLAGS) -w -MM $< >$@

.cc.d:
	gcc $(CFLAGS) -w -MM -MG $< >$@

.h.cd:
	classdesc $(ACTIONS) <$< >$@
```

The rules ending in .d automatically generate Makefile dependency rules for all the header files used in the project, and the `include` introduces these rules into the Makefile. This feature is not available with all `make`s, but is with *GNU make*. Since it is a relatively trivial exercise to install GNU make if its not already available, it makes sense to use the features of this tool.

The `-MM` option to gcc instructs the preprocessor to generate Makefile dependency lines of the form:

```
xxx.o: yyy.h zzz.h
```

for all header files `yyy.h, zzz.h` included with the `#include "..."` form, not the `#include <...>` form. This is usually what one wants, rather than generating large numbers of dependency lines for system headers that don't change. The `-MG` option tells the compiler to assume that files it can't find will be generated in the current directory. This is important, because we're going to include .cd files, which are automatically generated by `classdesc` by the .h.cd rule. Some native compilers support similar automatic dependency generation, however the behaviour differs in subtle ways from gcc. It is usually simpler to rely on gcc being available, as with GNU make. Note that gcc is not necessarily being used for code generation — one can still use the native compilers for that.

With these rules defined in the Makefile, all you need to do use a statement of the form `buf << foo;` is to place the statement

```
#include "foo.cd"
#include <classdesc_epilogue.h>
```

somewhere after the `#include "foo.h"` line, including the `foo` class definition. Any changes to the foo definition will update everything automatically.

It is mandatory that `classdesc_epilogue.h` is included, it is not an optional feature. You will now get a link failure if you do not included this file when needed:

```
undefined reference to '(anonymous namespace)::classdesc_epilogue_not_included()'
```

EcoLab users should use the file `ecolab_epilogue.h` instead.

13.1.2 Using Classdesc: Method 2, building a library

For most purposes, generating inline *action* definitions suffices for most purposes. However, if you have a lot of different classes for which you need *descriptors* defined, then compile times may become excessive. An alternative is to generate descriptor definition files for each class, and compile these into a library. This is achieved by the following rules:

```
.SUFFIXES: $(SUFFIXES) .h .cd .cdir .a
.h.cd:
        rm -rf $*.cdir
        mkdir -p $*.cdir
        classdesc -workdir $*.cdir -include ../$< $(ACTIONS) <$< >$@

.cd.a:
        $(MAKE) $(patsubst %.cc,%.o,$(wildcard $*.cdir/*.cc))
        ar r $@ $*.cdir/*.o
```

The `-workdir` option requests `classdesc` to write out the definition files into a new directory (`$*.cdir` expands to `foo.cdir` in the foo example). Function declarations are written out on standard output, which in this case is redirected to `foo.cd`.

The `-include` directive tells classdesc to insert the line `#include "../foo.h"` into the definition files, so that the definitions can be compiled.

The next (rather complicated) line compiles each of the definition files. The reason for recursively calling make, rather than the compiler directly, is that GNU Make is able to compile the directory in parallel, reducing compilation times.

See the polymorph example, which uses this technique.

13.1.3 Synopsis of classdesc

Syntax:

> `classdesc [-workdir` *directory*`] [-include` *include-file*`] [-I` *directory*] [-nodef] [-respect_private] [-typeName] [-onbase] *descriptors...*

`classdesc` takes as its input the preprocessed model definition file, that contains all class definitions available to the model. It outputs a functor definition that recursively calls itself for each of the members of that class. If `-workdir` is specified, the functor definitions are written to the specified directory for later compilation into a library. The (pre-expanded) source header file should also be included via the `-include` switch so that all necessary types are defined. If `-workdir` is not specified, the functor definitions are output as inline declarations on standard output.

Normally, classdesc reads its input from standard input, but some operating systems have trouble with this (eg Windows, but not Cygwin). An alternative is to specify the input file using the `-i` flag.

If `-I` switches are specified, the specified directory will be added to the search path for the action base include files.

If a type has been forward declared (eg `class foo;`), but not defined in classdesc's input source, a dummy definition is emitted of the form `class foo {};`. The purpose of this is to ensure that the action routines for types containing pointers to such declared types will compile. This behaviour can be turned off by specifying the `-nodef` option to classdesc.

By default, classdesc will generate descriptors that access private and protected members. If a class has private or protected members, a (see §13.1.6) declaration needs to be given to allow access to the private members. Alternatively, it may be undesirable to expose the internals of an object, for example if the object is being exposed to a different programming environment. In such a case, the `-respect_private` can be used to suppress the accessing of private or protected members.

By default, enums are treated as integers. Sometimes it is desirable to treat them (see §13.6), and this is managed by the `-typeName` option.

`-onbase` turns on the ability to distinguish between an action called on a base class and one called on a member. If your descriptor does not require this distinction, simply provide a method that calls to the original descriptor:

```
template <class T>
void pack_onbase(pack_t& x,const string& d,T& a)
{::pack(x,d,a);}
```

By default, -onbase is disabled as enabling it will break existing code. -onbase will become the default in Classdesc.4

13.1.4 Limitations to classdesc

`Classdesc` will work with any syntactically correct C++ code, and attempt to do the best it can. It ignores anything that is not a struct/class definition, or an enum definition. Classdesc does not preprocess the code presented to it — if you depend on the preprocessor in your class definitions, you must filter your code through the preprocessor first,[16] defining the macro `_CLASSDESC` to ensure pragmas are seen by the Classdesc processor.

[16]Use of the preprocessor is not to be recommended, however, as the contents of included files are also passed to classdesc, leading to a large number of inlined functions being emitted, and correspondingly long compilation times.

Unfortunately, overloaded member functions cannot be resolved to a distinct member pointer, so are quietly ignored by classdesc. This is not an issue with serialisation, of course, as all member functions are ignored, but has implications for descriptors such as `TCL_obj` that export object functionality.

Raw pointers also cause problems in that there is no information at runtime about how many objects a pointer points to, or whether it is reasonable to extend the array with memory taken from the heap. Support for the various uses of pointers is discussed in §13.2.1.

Another issue that occurs in reference to types defined in the current namespace with template parameters occuring as part of a specialisation. For example:

```
namespace frozz
{
  class Bar {};
  template <> class Foo<Bar> {};
}
```

In this case, the classdesc processor does not know which namespace Bar is defined in, (as its more forgetful than your average C++ compiler), so you will get a compile error that Bar is unknown. The workaround in this case is to full qualify type where necessary, ie replace the above code with

```
namespace frozz
{
  class Bar {};
  template <> class Foo<frozz::Bar> {};
}
```

13.1.5 supported #pragmas

#pragma omit *action typename* Do not emit a definition of *action* for this type. It is up to the programmer to supply a definition for this type. The typename needs to be fully qualified with its namespaces.

#pragma treenode *typename* Asserts that pointers to this type refer to a single object or are `NULL`, and that following the links from this object will not return to the same object (no cycles). True of trees, and of limited validity with *directed acyclic graphs.*

#pragma graphnode *typename* As above, except that the graph is allowed to contain cycles. It generally requires more expensive algorithms to traverse a general graph than to traverse a tree.

13.1.6 CLASSDESC_ACCESS

Because serialisation requires access to the private parts of a class definition, by default `classdesc` will emit actions for all members of class, whether private or public. Either all members of the class need to be declared `public`, or a `friend` declaration needs to be inserted so as to allow the action function to have access to the private members. The best way to do this is to put a `CLASSDESC_ACCESS` declaration in the class's definition, as in the following example:

```
class foo
{
   int x, y
   CLASSDESC_ACCESS(foo);
 public:
   int z;
}

template <class T>
class bar
{
   T x, y
   CLASSDESC_ACCESS(bar);
 public:
   T z;
}
```

The macro takes a single argument — the typename of class in question. The program `insert-friend` parses it input, and outputs a copy with these declarations appropriately inserted. This may be used to automate the process, but is not 100% reliable.

The CLASSDESC_ACCESS_TEMPLATE is now deprecated, and is synonymous with CLASSDESC_ACCESS.

The header file `classdesc_access.h` defines versions of these macros for the `pack/unpack` actions — use this file as a guide for writing your own macros if you implement other actions.

On the other hand, for descriptors like `isa`, and for exposure descriptors like `TCL_obj` in *EcoLab*, it is not desirable to process private members. For these purposes, use `-respect_private` command line flag of `classdesc`.

13.1.7 Excluding particular members from the descriptor

Finer grained crontrol over whether members are included or not in the descriptor definition is given by means of the `Exclude` template class. For all intents, an `Exclude<T>` variable is drop-in replacable for a variable of type `T`. However, provided your descriptor base definition supports it, such variables are simply ignored by the descriptor. Classdesc provided descriptor base files ignore variables of this type.

13.1.8 STL containers

Container support for standard STL containers is handled automatically by the standard descriptors. To extend the support to custom containers, it is necessary to inform classdesc that your custom type is a container, and that is done via two type traits: `is_sequence` and `is_associative_container`, which handle these two concepts from the standard library. The standard sequence containers are vector, deque and list, and the standard associative containers are set, map, and the multi_ and unordered_ variants of those.

To enable support for your custom container, eg one that conforms to the sequence concept, define a type trait for your custom container somewhere prior to (See §13.1.1) being included. For example:

```
namespace classdesc
{
  template <> struct is_sequence<MyContainer>: public true_type {};
}
```

If you wish to support containers in your own custom descriptor, you should ideally not refer to specific types, but use these type traits to write truly generic code. This involves quite advance metaprogramming skills, but you can use the file `pack_stl.h` as an exemplar.

13.2 pack/unpack

The pack/unpack function implements "serialisation", or the conversion of an object into a binary form that can be saved to a file or transmitted over the network to a process running in a different memory address space. The obvious strategy of simply sending `sizeof(`*object*`)` bytes starting at address &*object* doesn't work, as the object may contain members that are implemented using pointers — eg dynamic strings or arrays. The obvious strategy will simply copy the contents of the pointer, which will not be valid in the destination's address space.

Using the recursive approach, simple data types can be serialised in the obvious way, or even transformed using into a machine independent form such as XDR. Data types such as arrays or strings can be handled in a type dependent fashion by supplying appropriate specialised definitions.

In this case, the `pack_t` type implements a buffer into/from which the data is packed/unpacked. It has public members `char *data; int size` which point to the buffer and its size, that can then be used for further functionality such as the checkpoint/restart functionality of *EcoLab*.

There are 4 methods of using the pack function on an object —

1. `template <class T> pack_t& pack_t::operator<<(T&)`
2. `template <class T> ::pack(pack_t&, string, T&)`

3. `template <class T> ::pack(pack_t&, string, is_array,  T*, int)`
4. `void pack_t::packraw(char *,int)`

The `classdesc::string` argument is not used, just pass `""` to it. The third method above is a utility routine for packing arrays of objects, `is_array` is a dummy type, so just pass the object returned by the default constructor `is_array()`, then the final two arguments are the array pointer and size respectively. One could easily explicitly loop over the array elements using the first two methods. The last method is used to pack arbitrary byte data into the buffer. It differs from `::pack(pack_t&, string, is_array, char*, int)` in that the latter packs a series of characters, which are usually 32 bit quantities. It is thus both more efficient than the latter, as well as providing a means to unpack data stored in packed char format.

`xdr_pack` is derived from `pack_t`. Instead of packing to native data representation, it uses XDR data representation, which is machine independent. The `XDR_PACK` macro symbol must be defined to enable this functionality, otherwise `xdr_pack` is synonymous with `pack_t`, allowing the code to be employed on machines that do not provide XDR functionality.

`unpack_t` is typedef'd to `pack_t`, and `unpack_base.h` is a link to `pack_base.h`.

In order to use the streaming operators `<<` and `>>` you need to include the file `pack_stream.h`, after all the corresponding `pack()` definitions. This is automatically done by including `classdesc_epilogue.h`

13.2.1 Pointers

`<rant>`

> Pointers are evil!. *Pointers are a dangerous programming construction, and widely abused in the C++ programming world. They often lead to obscure programming errors, and in particular to* memory leaks*, which are notoriously hard to debug. If there is an alternative method that encapsulates pointers, or avoids their use altogether, then that should be used. Unfortunately, pointers are vital to programming in C, and many of the practices are imported into C++.*

`</rant>`

Whilst the above might be considered a little extreme, it is worthwhile listing what pointers are used for, and considering what alternatives there might be. In Classdesc, objects are usually assumed to have the following properties: default constructable, copyable, assignable and serialisable. All the simple traditional C data types except for pointers satisfy these properties. Compound types (structs and classes) whose members satisy these properties also satisfy them, with classdesc automagically extending the serialisability property.

Pass by reference In C++, the reference operator `&` makes obsolete the use of pointers for returning values from function subroutines, and improves type safety. Of course this use of pointers is of no concern for serialisation.

Dynamic Memory In C++, we have the `new` operator to return an array of objects from the heap. However, one must be careful to `delete` the array of objects once finished with, otherwise a memory leak may result. In this case, encapsulation can help. If the pointer is encapsulated with a class definition, then the `delete` can be called automatically when the object's destructor is called, which happens automatically when an object goes out of scope. The simple case of allocating some dynamic memory from the heap can be most readily performed using the standard library container `vector<T>`.

Strings `char *` variables can be replaced by the standard library `string` type.

Dynamic references Dynamic references are used a lot to represent graph structures, or to provide access to objects declared outside the current scope. For some purposes, C++'s static reference type (`T&`) is suitable, but is limited to being initialised only at object construction time. Also, any reference loops will cause serialisation to enter an infinite loop and crash. C++ offers more possibilities in the form of "smart pointers", that guarantee destruction of the referenced object once the reference object goes out of scope. The standard C++ library provides `auto_ptr`, but this is noncopyable, pretty much defeating the purpose of smart pointers. The Boost library (http://www.boost.org) provides several different sharable smart pointers, that can be used. Classdesc provides its own concept, (see §13.3.1) that is a type of dynamic reference. It should be noted that linked lists can be handled easily with standard library containers.

Legacy APIs Many C-based legacy APIs use pointers for pass by reference functionality, strings, or for anonymous references (to avoid publishing the full specification of an object). These APIs can be easily encapsulated to ensure any allocated pointers are appropriately cleaned up.

Global references An object that needs to be destroyed before main() exits, yet needs to be referred to globally throughout the program cannot be implemented as a global object (which is destroyed after the program exits). Instead, it has to either be a global pointer, which is initialised when the object is created, or the entire program must be implemented as a method of an object which is created in main().

Runtime polymorphism Since the actual datatype might vary, only references to the object can be handled. Traditional pointer-based polymorphic systems are not copyable, assignable nor serialisable, as copying a point to an allocated object invariably leads to double free() errors (in the case of destructors cleaning up pointers) or to memory leaks (in the case destructors don't do anything). Traditionally, copying is performed by means of a `clone()` virtual method, which is also how this is done in Java. EcoLab provides an (see §13.3.5) base class in `Poly.h` which provides an interface for cloning, and interfaces for serialisation can be found in the `PolyPack`, `PolyXML` and `PolyJson` headers and a simple runtime type identification system. To create a reference, the smart, modern way to do this is via the `shared_ptr` smart pointer class, which is found in the TR1 library of your compiler, or in Boost if your compiler does not do TR1.

13.3 Graph serialisation

Dynamic references can be serialised, provided a few properties are known about the data structure they make up. There is no way of knowing whether standard pointer actually points to a real object, nor how many. However, since collections of objects are more conveniently handled by standard containers, and since no object can pointed to by the value 0 (or NULL), we can determine these things if the programmer follows a protocol whereby a pointer either references at a single object, or is NULL. Using a smart pointer additionally enforces this protocol. We call this the treenode or graphnode protocol, depending on whether the referenced data structure has cycles or not.

By default, packing a pointer raises an exception. However, this behavior is changed either by specifying a given type obeys the treenode or graphnode protocol using the `treenode` pragma or `graphnode` pragma respectively. Alternatively, the `pack_t::ptr_flag` can be set to the values `TREE` or `GRAPH` respectively.

What happens in this case, is that special graph serialisation algorithms defined in `pack_graph.h` are called that ensure graph objects are serialised or deserialised correctly. For deserialisation, new objects must be created to store the node contents. References to these objects are placed in the `alloced` member of the `pack_t` buffer object. These newly created objects are destroyed when the buffer object is destroyed, unless a copy of the `alloced` vector is made first. Conversely, the objects can be destroyed without destroying the buffer by clearing the alloced vector. Individual objects can be destroyed by simply erasing them (assuming you know which ones!).

pack_graph is a recursive depth-first algorithm, that could potentially blow up the stack if the recursion depth is not limited. The recursion limit can be specified using `pack_t::recur_max`. pack_graph restarts the algorithm once the recursion limit is reached.

The `pack_graph` algorithm can also be applied to smart pointers or other reference types. An example is the `ref` smart pointer provided with Classdesc. For your smart pointer class T, you will need to provide an `Alloc<T>` class with an `operator()(pack_t* buf, T& x)` that returns a newly allocated object referenced by x. The `buf` object is there if you wish to use the alloced mechanism.

13.3.1 Ref

Ref is a reference counted shared smart pointer implementation. When all references to an object are destroyed, the object it references is also destroyed. It is somewhat equivalent in functionality to Boosts `shared_ptr` or `intrusive_ptr`. Boost, however, is a big library, so the decision was made to avoid any dependencies of Classdesc or EcoLab on Boost.

Ref performs entire object life cycle management. Initialising it with an object makes a copy, it does not pass control of the object to the ref. As such, it requires that its target type has a default constructor. Assigning, or copying a ref object duplicates the reference to a single object. Dereferencing works as with traditional pointers.

Synopsis:

```
template <class T>
class ref
{
public:
  ref(); // unitialised refs are NULL
  ref(const ref& x);
  ref(const T& x);  //copies x
  ref& operator=(const ref& x);
  ref& operator=(const T& x);  //copies x
  T* operator->();  //dereference operators
  T& operator*();
  void nullify();  //set reference to NULL
  bool nullref();  //returns true if invalid
  operator bool (); //returns true if valid
  bool operator==(const ref& x);
  bool operator==(const T* x); //true if x points to this's target
  bool operator==(const T& x); //true if x==this's target
};
```

Packing a ref object automatically invokes the `pack_graph` algorithm.

13.3.2 Converting code using traditional pointers to using ref

Let `p` be a pointer and `r` be the ref it is being changed to.

This table details common idioms that need changing to convert the pointer to a ref:

`p=NULL;`	`r.nullify();`
`p=new T;`	`r=T();`
`p=new T(x,y,z);`	`r=T(x,y,z);`
`p==NULL, p!=NULL etc`	`r, !r etc`
`delete p;`	`Remove this statement, it is superfluous`
`p->*m();`	`(*r).*m() No Member pointer dereference`
`p++, p+1, etc.`	`Illegal.  Consider using a container type.`

Assignment and copying refs are more expensive than the equivalent pointer operations due to the reference counting mechanism. Therefore, consider using C++ references whereever possible:

```
void foo(const ref<int>& x); instead of void foo(ref<int> x);
{                                       {
  const ref<int>& y=...;     instead of    ref<int> y=...;
```

A certain amount of care must be taken if you need to declare the ref as non-const. If it is just the target that needs updating, it is fine to use a `ref<T>&` variable. However, if the ref itself needs updating, then use `ref<T>` instead.

13.3.3 Roll your own

Sometimes, you need to develop you own dynamic data types, whether a smart pointer, or a STL container like type, that requires a pointer as part of its implementation. You will need to provide your own hand crafted serialisation routines for these, and use the `omit` pragma to prevent classdesc from emitting an automatic definition (or simply arrange things so that classdesc is not run on the definition file). You are advised to keep the pointer encapsulation suitably minimalist, so as to minimise the amount of manual code of serialisation routines.

13.3.4 Synopsis of `pack_t`

```
struct pack_t
{
  char *data;
  size_t size;
  size_t pos;
  Ptr_flag ptr_flag;
  std::vector<PtrStoreRef> alloced; //allocated data used for cleaning up
  pack_t(size_t sz=0);
  pack_t(const char* filename, const char* mode); //pack to file
  pack_t& reseti();
  pack_t& reseto();
  pack_t& seeki(int offs);
  pack_t& seeko(int offs);
  void packraw(char *x, int sz);
  void unpackraw(char *x, int sz);
};
```

`data` points to the beginning of the buffer maintained by `pack_t`. `size` refers to the current position of the input stream (ie the size of current valid data). `pos` refers to the current position of the output stream. It is an error to assign values directly to `pos`. It is OK to assign a value to size when setting up a `pack_t` variable for unpacking. Do not update `size` whilst packing. It is OK to assign a pointer value to `data` for unpacking only, however one should note that `delete` is called on the pointer during destruction, so in general you should reset `data` to NULL before the `pack_t` variable goes out of scope, if you don't want the object deleted (for instance if you've set it to the address of a static array).

`size` and `pos` can be reset to 0 using the `reseti()` and `reseto()` routines respectively. `seeki()` and `seeko()` allows arbitrary positioning of the streams — the seek offset in this case is relative to the current position.

The constructor takes an integer argument which specifies the size of an initial buffer. For example:

```
pack_t b(N); b.size=N;
fread(b,N,1,f);
b>>foo;
```

is a common idiom for reading some data in from a file.

`packraw` and `unpackraw` allow arbitrary byte data to be pushed onto the buffer and taken off. This involves an extra copy operation, but is the safest way of manipulating the buffer directly.

13.3.5 Polymorphism

C++ has two notions of polymorphism, compile-time and runtime. Compile-time polymorphism (aka generic programming) is implemented in terms of templates, and allows the provision of code that can work on many different types of objects. On the other hand, runtime polymorphism involves the use of virtual member functions. Whereever generic programming can solve a task, it is preferred over runtime polymorphism, as virtual member functions introduce procedure call overhead, and inhibit optimisation. Furthermore, the use of a copyable, assignable and serialisable class like `shared_ptr` introduces additional overheads.

Nevertheless, there are situations that cannot be solve with compile-time polymorphism, for example a container containing objects of varying types. The smart, modern way to do runtime polymorphism is via a smart pointer, such as `shared_ptr`, found in TR1. To use `shared_ptr` in a DCAS fashion, your object heirarchy must implement the following interface (provided as an abstract base class `PolyBase`), and the `PolyPackBase` .

```
  template <class T>
  struct PolyBase: public PolyBaseMarker
  {
    typedef T Type;
    virtual Type type() const=0;
    virtual PolyBase* clone() const=0;
    /// cloneT is more user friendly way of getting clone to return the
    /// correct type. Returns NULL if \a U is invalid
    template <class U> U* cloneT() const;
    virtual ~PolyBase() {}
  };

  template <class T>
  struct PolyPackBase: virtual public PolyBase<T>
  {
    virtual void pack(pack_t&, const string&) const=0;
    virtual void unpack(unpack_t&, const string&)=0;
  };
```

Any type may be acceptable for the type identifier system, but needs to be orderable if using the `Factory` class. Typically, ints, enums or strings are used for the type class. A nice implementation is to use the typeName function to return a string representation of the type:

```
  string type() const {return typeName<T>();}
```

The `create()` method is a static factory method that allows you to create an object of the type specified. This is not part of the `PolyBase` interface, but needs to be provided by the base class of the object heirarchy. Its signature is

```
  static object* create(const Type&);
```

The `Factory` class may used for this purpose: If the base class of your class heirarchy is `object`, and you are using strings for your runtime type identifier, then declare a factory object as

```
Factory<object,string> factory;
static object* object::create(const string& n)
{return factory.create(n);}
```

The only other thing required is to register the type heirarchy. This is most conveniently and safely done at factory construction time, and indeed the `Factory` class requires you provide a custom default constructor, but type registration can happen at any time via the `Factory::registerType<T>()` method, which registers type `T`. The factory method requires that all objects in teh class heirarchy are default constructible, but other than that makes no assumptions other than it must have a `type()` method.

To assist in deriving classes from `PolyBase`, the `Poly` template is provided.

```
template <class This, class Base=object> struct Poly;
```

The first template argument `This` is the class you're currently defining, and `Base` is the base class you are deriving from, which may be `object`, or may be another class higher in the hierarchy. This provides an implementation of the clone method. For each of the serialisation descriptors, there is a similar template, so `PolyPack`, `PolyXML` and `PolyJson`.

```
template <class T, class Type>
struct PolyPack: virtual public PolyPackBase<Type>
{
  void pack(pack_t& x, const string& d) const;
  void unpack(unpack_t& x, const string& d);
};
```

These can be used in a "mixin" fashion by means of multiple inheritance, eg.

```
template <class T>
struct Object:
  public Poly<T,object>,
  public PolyPack<T,string>,
  public PolyXML<T,string>
{
  string type() const {return typeName<T>();}
};
```

One thing to be very careful of is your inheritance heirarchy. Multiple inheritance can easily cause a "no unique final overrider", because the implementations of the various virtual function come in from different classes that are mixed in. In the examples directory, are two different solutions to this problem - the first is providing a custom implementation template class, by manually copying the mixin definitions, and the second actually uses the mixin definitions through inheritance, but annotates each class with the base template after the class is defined. The two solutions are shown in UML in figure 3.

13.3.6 Packing to a file

Instead of packing to a buffer, and subsequently storing data to a file, you can directly pack to a file by passing the filename and openmode arguments to pack or `xdr_pack`'s constructor. The arguments are identical to that of fopen:

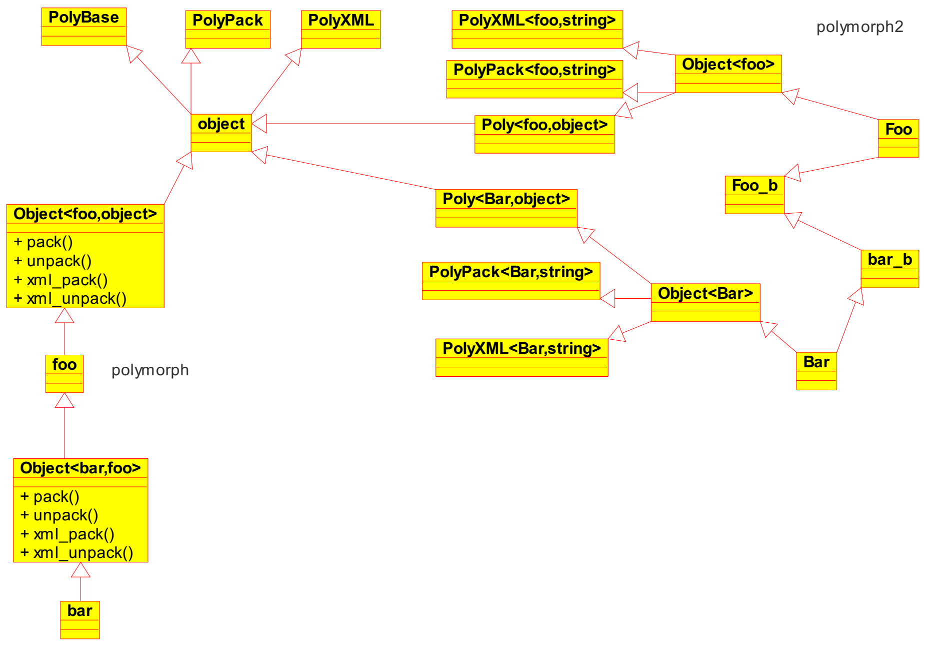


Figure 3: Diagram of the two different example polymorph implementations for a non-flat class heirarchy.

```
 {
  pack_t checkpoint("foo.ckpt","w");
  checkpoint << bar;
 }
```

The advantage here is large checkpoints can be written to disk without intermediate buffering in memory. The file is closed when the `pack_t` object is destroyed.

13.3.7 BinStream — binary streaming to a pack_t

BinStream is an adaptor class that allows streaming of POD (plain ordinary data) types to/from a pack_t type for efficiency reasons. POD types can be basic data types like ints/floats, structs of such, arrays of such and standard containers of such.

You can either use a BinStream class, which takes a pack_t reference (can be an xdr_pack object as well) in its constructor. Alternatively, you can construct the pack_t object directly with the BinStream using the template BinStreamT. The template argument refers to the pack_t type.

Examples:

```
struct foo {a,b};
vector x(10,2);
pack_t p;
foo a;
int i;
BinStream bs(p);
bs << 1 << a << x;
bs >> i >> a >> x;
BinStreamT<xdr_pack> bst("foo.dat","w");
bst << 1 << a << x;
```

13.4 isa

`isa` is an action to determine whether a particular class is derived from another type. To use this, create an `isa` action in the usual way using classdesc:

```
classdesc isa <header.h >header.cd
```

Then `isa(e,Y())` will return whether `e` is of a type derived from type `Y`.

The functionality of `isa` is better achieved using the TR1 type traits feature: `is_base_of`.

13.5 dump

`dump` is a descriptor that produces a human readable description of an object on a stream.

```
template <class T>
void dump(std::ostream* out, const string& desc, T& arg);
```

13.6 Symbolic enums

By default, enums are treated as though they are integers. This works well for serialisation, but if the data is meant to be read by a human, it is desirable to display the enums in symbolic form.

In order to do this, classdesc will emit descriptors using `Enum_handle<E>`, where `E` is an enum, which wraps an enum variable. In particular, the `Enum_handle` will return a string symbolic representation of the enum, or assign the appropriate value to the enum variable when assigned a string constant representing the symbolic value of the enum:

```
template <class T> //T is an enum
class Enum_handle
{
public:
  Enum_handle(T& arg); // wrap enum arg
  operator std::string() const; //symbolic form of enum
  operator int() const; //integral value of the enum
  const Enum_handle& operator=(T x);
  const Enum_handle& operator=(int x);
  const Enum_handle& operator=(const std::string& x); //symbolic assignment
};
```

Classdesc handles writing the dictionaries needed to perform this conversion to and from symbolic constants. See the `xml_pack` descriptor for an example of its use.

Access to the enum reflection data is via the EnumKeys class

```
template <class T>
class EnumKeys
{
public:
  int operator()(std::string key);
  std::string operator()(int val);
  size_t size() const;
  iterator begin() const;
  iterator end() const;
  Siterator sbegin() const {return begin();}
  Siterator send() const {return end();}
  Viterator vbegin() const {return begin();}
  Viterator vend() const {return end();}
};
```

```
template <class T> const EnumKeys<T>& enum_keys();
```

So `enum_keys<enum Foo>()("bar")` returns the numerical value of the enum constant `bar` and `enum_keys<enum Foo>()(bar)` returns the string value `"bar"`.

The various iterators allow iteration, or population of containers:

```
const EnumKeys<Foo> e(enum_keys<Foo>());
map<Foo,string> m(e.begin(), e.end());
vector<string> s(e.sbegin(), e.send());
vector<Foo> v(e.vbegin(), e.vend());
```

13.7 typeName

The `template <class T> std::string typeName<T>()` function returns the symbolic name of the type T. Typename data for user-defined classes and structs is emitted when the -typeName flag is given to classdesc.

13.8 Functional reflection

Classdesc provides metaprogramming reflection support for functional objects with the `function.h` header. Provided in the `classdesc::functional` namespace are the following metaprogramming templates:

`Arity`	`F` has `Arity<F>::V` arguments
`Return`	`Return<F>::T` is the return type of `F`
`Arg`	`Arg<F,i>::T` is the type of the ith argument of `F`
`is_member_function_ptr`	`is_member_function_ptr<F>::value` is true if F is a member function pointer
`is_nonmember_function_ptr`	`is_nonmember_function_ptr<F>::value` is true if F is a ordinary function pointer
`is_function_ptr`	`is_function_ptr<F>::value=true` if F is either a member function pointer or an ordinary function pointer
`void apply(R* r, F f, Args args);`	apply function f to an array of arguments

Metaprogramming datatypes will have either a static data member `V` or a typename `T`. In sympathy with classdesc's fairly compact style, we use these compact names rather than Boost's conventions of `value` and `type`. The `is_` structure use value so as to be compatible with the use of enable_if (supplied in the classdesc.h header).

The apply function takes and array of arguments of type Args. Args can be a user defined type, but must be convertible to all the types used by F. An example type is defined in javaClass_base.h. The return object is passed as parameter r. This allows for the possibility of void returns, in which case apply ignores what is passed here (can pass NULL if F is known to have a void return type).

13.9 classdescMP

classdescMP is a class library supporting the use of the MPI library. It consists of three main concepts, MPIbuf, MPIslave and MPISPMD.

Warning: Do not use MPISPMD as a global variable. Both of these types call MPI_Finalize() in their destructors. Some MPI implementations require MPI_Finalize() to be called before main() exits (although MPICH is quite happy to call MPI_Finalize() after main() exits). For maximum portability, declare the MPIslave or MPISPMD object as a local variable in main(), and initialise a global pointer to refer to this object elsewhere in the code.

13.9.1 MPIbuf

MPIbuf is derived from pack_t, and so arbitrary objects can be placed into an MPIbuf in just the same way as a pack_t. If the HETERO preprocessor symbol is defined, then MPIbuf is derived from xdr_pack instead, so MPIbufs can be safely used on a heterogenous cluster — thus obviating the need to use the MPI compound type mechanism (MPI_Type* series of functions).

Specification:

```
 class MPIbuf: public pack_t
{
public:
  MPI_Comm Communicator;
  int myid();   /* utility functions returning rank and number in */
  int nprocs(); /* current communicator */
  bool const_buffer;  /* in send_recv, all messages of same length */
  int proc, tag; /* store status of receives */

  MPIbuf(): pack_t() {Communicator=MPI_COMM_WORLD; const_buffer=false;}

  bool sent(); //has asynchronous message been sent?
  void wait(); //wait for asynchronous message to be sent

  void send(int proc, int tag);
  void isend(int proc, int tag); //asynchronous send

  MPIbuf& get(int p=MPI_ANY_SOURCE, int t=MPI_ANY_TAG);
  void send_recv(int dest, int sendtag,
                 int source=MPI_ANY_SOURCE, int recvtag=MPI_ANY_TAG);
  void bcast(int root);

  MPIbuf& gather(int root);
  MPIbuf& scatter(int root);

  MPIbuf& reset();
  bool msg_waiting(int source=MPI_ANY_SOURCE, int tag=MPI_ANY_TAG);
};
```

The simplest additional operations are `send` and `get`. `send` sends the buffer contents to the nominated processor, with the nominated message tag, and clears the buffer. `get` receives the next message into the buffer — if processor or tag are specified, the message is restricted to those that match. `get` returns the value of `*this`, so the message can be unpacked on one line, eg:

```
buffer.get() >> x >> y;
```

`get` places the source and message tag for the received message in `proc` and `tag`.

`send_recv` does a simultaneous send and receive, sending the buffer to the nominated destination, with nominated tag. If the flag `const_buffer` is set, then all messages must be of equal length. The prevents the need to send the message sizes first.

`bcast` performs a broadcast.

`gather` concatenates the MPIbufs from all nodes onto the MPIbuf on the root node. If `const_buffer` is set, then the more efficient `MPI_Gather` is used, otherwise the buffer sizes are gathered first, and `MPI_Gatherv` used.

`scatter` scatters an MPIbuf from the root node to all the nodes. The data destined for each node must be separated by `mark` objects, as in:

```
cmd << A << mark() << B << mark(); cmd.scatter(0);
```

Again, if the data to be scattered is of identical size for each node, set the `const_buffer`, and the more efficient `MPI_Scatter` will be employed instead of `MPI_Scatterv`.

By default, all operations take place in the `MPI_COMM_WORLD` communicator. This behaviour can be changed by assigning a different communicator to `MPIbuf::Communicator`

Messages can be sent asynchronously using `isend()`. `sent()` can be used to test whether the message has been passed, and `wait()` can be used to stall until the message has been sent. `wait()` is always called prior to the MPIbuf object being destroyed.

13.9.2 Manipulators

In a way analogous to iostreams, manipulators are actions that can be pushed onto the buffer. In the case of MPIbuf, four manipulators are defined, `send`, `isend` and `bcast`, and `mark` that allows a message to be composed and sent on one line, eg:

```
MPIbuf() << i << j << send(p,tag);
MPIbuf() << i << j << isend(p,tag);
MPIbuf() << i << j << bcast(0);
```

`mark` is used for separating the data items to be `scatter`ed.

13.9.3 MPIbuf_array

`MPIbuf_array` is a convenience type for managing a group of messages:

```
  class MPIbuf_array
  {
  public:

    MPIbuf_array(unsigned n);
    MPIbuf& operator[](unsigned i);

    bool testall();
    int testany();
    vector<int> testsome();
    void waitall();
    int waitany();
    vector<int> waitsome();
  };
```

The `testall()`, `testany` etc methods perform the MPI equivalent call on the group of messages.

`MPIbuf_array` is useful for managing an all-to-all calculation, as per the following typical example:

```
    {
        tag++;
        MPIbuf_array sendbuf(nprocs());
        for (unsigned proc=0; proc<nprocs(); proc++)
          {
            if (proc==myid()) continue;
            sendbuf[proc] << requests[proc] << isend(proc,tag);
          }
        for (int i=0; i<nprocs()-1; i++)
```

```
          {
            MPIbuf b;
            b.get(MPI_ANY_SOURCE,tag);
            b >> rec_req[b.proc];
          }
    }
```

Note that the outer pair of braces that all messages have been sent and received in the group. Using an explicit tag is useful to prevent message groups from interfering with each other.

13.9.4 MPIslave

```
template<class S>
class MPIslave
{
public:
  int nprocs, myid;
  vector<int> idle; /* list of waiting slaves, valid on master */
  MPIslave();
  ~MPIslave() {finalize();}
  void init();
  void finalize();
  MPIbuf& operator<<(void (S::*m)(MPIbuf&))
  template <class T> MPIbuf& operator<<(const T& x);
  void exec(MPIbuf& x);
  MPIbuf& get_returnv();
  void wait_all_idle();
  void bcast(MPIbuf& c=cmd);
};
```

MPIslave is an implementation of a master-slave application. An MPIslave object must be created within an MPI parallel region (eg created by MPISPMD). The slave code is implemented as a class (S say), with each method implementing a (see §13.9.5) being of type `void (MPIbuf&)`. Declaring a variable of type `MPIslave<S>` will set up an interpreter on the slave processor. The remote processes are closed down once the variable is finalised, or goes out of scope.

See the file `mandelbrot.cc` in the `mpi-examples` for an example of how this works.

`operator<<` is provided as a convenience — one can compose a message of the form:

```
MPIslave<S> slave;
slave << &S::foo << x << send(1);
```

without needing to declare additional `MPIbufs`.

For managing a list of idle slaves, the `idle` vector is employed, and is manipulated through the `exec` method, which dispatches a command to be executed on a slave, and the `get_returnv()` which returns the returned result as an MPIbuf, and places the processor from which it received a value back on the idle list. This technique is only valid for slave methods returning a message (even if its a null message). The `wait_all_idle` method waits for all slaves to return.

The `bcast()` method is a convenience method for sending the contents of MPISlave::cmd, or the optional MPIbuf argument to all slaves.

13.9.5 Remote Procedures

Ideally, remote procedures should be callable by a syntax similar to:

```
val=proxy->rp(arg1,arg2);
```

where proxy is an object on the calling thread that can forward the call to a remote object, call method `rp` on that remote object with arguments `arg1` and `arg2`, return a value `val` if necessary.

Unfortunately, C++ syntax requires that the `->` operator return an object that has a member `rp`, which is of little use if the desired object exists in a different address space. This could be addressed using the `->*` operator, which is a true binary operator.

However, a more significant problem comes when trying to transport the arguments, and return a value. It is possible to write template functions that know the types of arguments and return values, and can pack these into the message buffer. Clearly, one needs to write a separate template for each argument count, but this is not too difficult a task in practice, and can be made extensible. The problem comes with transmitting the type information to the remote address space. The obvious solution of sending a function pointer to a template handler function simply does not work, as these have different values in different address spaces. The next obvious solution of sending a pointer to a template member of MPIslave also does not work, although the reason seems obscure. Possibly, it is impossible to obtain a pointer to a template member function. The final remaining solution is to create another classdesc action, however it is known that one cannot overload on the number of arguments to a member pointer argument. Classdesc, at present, is not very good at parsing function arguments.

So a more primitive remote procedure calling interface is required, and the one based on the streams model of `MPIbuf` works well and is reasonably intuitive:

```
MPIslace << &S::method << arg1 << arg2 << send(p);
```

This will run `S::method` on the slave processor, where method must be defined as:

```
void method(MPIbuf& args)
{
  X arg1; Y arg2; args>>arg1>>arg2;
  ...
  args.reset() << result;
}
```

The last statement is only needed if the method returns a value. This is sent as a message to the master.

13.9.6 MPISPMD

```
class MPISPMD
{
public:
  int nprocs, myid;
  MPISPMD() {nprocs=1, myid=0;}
  MPISPMD(int& argc, char**& argv) {init(argc,argv);};
  ~MPISPMD() {finalize();}
  void init(int& argc, char**& argv);
  void finalize() {MPI_Finalize();}
};
```

`MPISPMD` is a simple class that arranges for `MPI_Init` to be called when initialised, and `MPI_Finalise` when destroyed. It use is primarily to construct SPMD style programs. See the `heat.cc` example program to see how it might be used.

13.10 Workarounds

There are times when classdesc simply cannot correctly parse syntactically correct C++, or won't be able to be adapted to do so. One of these situations occurs when a class definition refers to an object in the containing namespace, but the descriptor definition requires the fully qualified version of the name. An example is as follows:

```
namespace foo
{
  struct bar
  {
    enum Foo {x, y, z};
  };


  template <bar::Foo t>
  class Foobar {};
}
```

which is syntactically correct C++, but the generated descriptor looks like

```
template < bar :: Foo t >  struct access_pack<class ::foo::Foobar<t> > {
void operator()(classdesc::pack_t& targ, const classdesc::string& desc,class ::foo::Foob
{
using namespace foo;
}
};
```

The problem is that `bar::Foo` is not visible in the `classdesc_access` namespace where the `struct access_pack` type must be declared.

As a workaround, whenever this situation is encountered, use the fully qalified version of the type, ie as follows:

```
  template <foo::bar::Foo t>
  class Foobar {};
```

13.11 TCL_obj

The `TCL_obj` function creates a set of TCL commands that implement get/set functionality for the class members. For example, with a class definition:

```
class foo: public TCL_obj_t {int a, b; void foobar(int,char**)} bar;
```

`TCL_obj(&bar,"bar",bar);` creates the TCL commands `bar.a` and `bar.b`. To set the value of `bar.a`, use the command `bar.a` *val* from TCL. To get the value, use `[bar.a]`.

Also created is the TCL command `bar.foobar`, which will run respective member function of `foo` when called from TCL.

Any nonoverloaded member function can be accessed from TCL, provided the arguments and return types can be converted from/to TCL objects. In particular, it is not possible at present to call methods that take nonconstant references.

Overloaded method types in general cannot be called, but it is possible to create variable length argument lists by declaring a method with an `(int,char**)`, or a `(TCL_args)` signature. Such methods are not easily called from C++, and generally, one needs to define a set of overloaded functions of a different name (eg capitalised) suitable for calling from C++, as well as the variable length argument list for use from TCL. However, as a special case of an accessor (emulating the setting/getting of an member attribute), one may make use of the Accessor class, which is equally callable from C++ as TCL.

Accessor is not easily usable from within the C++98 language standard (see acessor.h in the test directory), but makes much more sense in the C++11 standard. For example, assume that `Name()` and `Name(const string&)` have been defined as a getter and setter method of the attribute `name`, then one may define a member

```
Accessor<string> name {
    [this](){return this->Name();},
    [this](const std::string& x){return this->Name(x);}
};
```

where the use of lambdas and brace initialisers makes it easy to assign code for the getter and setter components of the accessor. This member will be accessible as an attribute from TCL (just as if name had been defined as a string member), and also callable from C++ as `name()` or `name("someName")` as appropriate.

One downside of the Accessor class is that it is not copy constructible, as copying the accessor will copy a reference to the wrong accessed object. Consequently, if copy construction is required for the object being accessed (eg for DCAS), then a custom copy constructor needs to be provided.

As an alternative to `(int,char**)` arguments for implementing TCL commands, one can also declare the arguments `(TCL_args)`. `TCL_args` variables can be assigned to numerical variables or strings variables, and the appropriate argument is popped off the argument stack:

```
int x=args;
double y=args;
```

assigns the first argument to x and the second to y. This method use the `Tcl_Obj` structure, so the values needn't be converted to strings at all.

The arguments may also be assigned using the `>>` operator:

```
int x; double y;
args >> x >> y;
```

A third style uses the `[]` operator:

```
int x=args[0]; double y=args[1];
```

The number of remaining arguments is available as `TCL_args::count`.

If `operator>>(istream,T)` is defined, then you may also use the `>>` operator to set a variable of type `T`, eg:

```
void foo::bar(TCL_args args)
{
  iarray x;
  args>>x;
}
```

the assignment operator cannot be used for this purpose, unlike simple types, because nonmember assignment operators are disallowed in the standard. Type conversion operators do not appear to work.

For technical reasons, the name of the TCL command is available as `args[-1]`.

The `TCL_obj_t` data type also implements checkpoint and restart functions (§6.9.2), so that any class inheriting from `TCL_obj_t` also gains this functionality, as well as §(6.9.1).

A helper macro that performs the above is `make_model`, which is used in a declarative sense, which also initialises the checkpoint functor.

Associated with each of these TCL commands, is an object of type `member_entry<T>`. Pointers to these objects are stored in a hash table called `TCL_obj_properties`. The STL hash table behaved rather stangely when used for this purpose, so a class wrapper around TCL hash tables was employed instead:

```
template<class T>
struct TCL_obj_hash
{
  struct entry
  {
    entry& operator=(const T& x);
    operator T();
  };
  entry operator[](const char*x);
};
```

So objects of `member_entry<T>*` can be inserted into the hash table as follows:

```
member_entry<T>* m; eco_string d;
TCL_obj_properties[d]=m;
```

but to extract the data, use `memberPtrCasted`

```
if (T* m=TCL_obj_properties[d]->memberPtrCasted<T>())
   ... *m
```

will allow you to access the TCL object `d`, if it is castable to an object of type `T` (is a `T`, or is derived from a `T`).

A utility macro allows these objects to be accessed simply:

`declare`(*name*,*typename*, *tcl_name*)

where *name* is the name of a variable (in reality a reference), of type *typename* that will refer to the variable having the TCL handle *tcl_name*. The macro performs error checking to ensure such a variable actually exists, and that it is of the same type as *typename*.

Objects can be placed into `TCL_obj_properties` by a several different means:

1. `make_model`*(x)*, which places all of the leaf objects of *x* (which must be derived from `TCL_obj_t`) into `TCL_obj_properties`, and also completes the construction of the `TCL_obj_t` object;

2. `register`*(x)*, which places *x* into `TCL_obj_properties`, as well as the leaf objects — can also be called as `TCL_obj_register(`*object*,*object name*);

3. `TCLTYPE`*(typename)*, TCLPOLYTYPE(typename, interface), where *typename* is defined C++ type, and interface is a base class of typename. This creates the TCL command *typename*, which takes one argument, a variable name for it to be referred to from TCL, and creates an object of that type which it registers in `TCL_obj_properties`. If `TCLPOLYTYPE` is used, the base class type is used for registration - so this object can be used wherever a polymorphic type with the specified interface is expected. For example, consider the following code which creates and initialises an object of type distrand and gives it the TCL name PDF (from testdistrand.tcl):

   ```
   distrand PDF
   PDF.min -10
   PDF.max 10
   PDF.nsamp 100
   PDF.width 3
   PDF.Init dist
   .....
   PDF.delete
   ```

   This macro also defines an x.delete procedure for deleting that object, once no longer desired.

A TCL registered object, particularly dynamically created `TCLTYPE` objects can be assigned to a member of type `TCL_obj_ref`. This is particularly useful for random number generators:

```
class Foo: public TCL_obj_t
{
 public:
   TCL_obj_ref<random_gen> rng;

   ...
     rng->rand();
};
```

Then the member `Foo::rng` can be assigned an arbitrary random number generator within the TCL script, such as the PDF example above.

```
distrand PDF
PDF.min -10
...
foo.rng PDF
...
```

Using TCL_obj_ref also allows that object to be serialised, and to be reconnected after a restart, provided the object has been created prior to the restart.

13.12 Member hooks

TCL_obj_t has the following two members, that allow one to assign a hook that is called after every TCL_obj call into C++ code. This can be used, for example, to record what methods have been called on the C++ model. There is one for each type of method signature called from TCL.

```
struct TCL_obj_t
{
    typedef void (*Member_entry_hook)(int argc, CONST84 char **argv);
    Member_entry_hook member_entry_hook;
    typedef void (*Member_entry_thook)(int argc, Tcl_Obj *const argv[]);
    Member_entry_thook member_entry_thook;
};
```

13.13 TCL_obj_stl

The header file TCL_obj_stl provides TCL_obj support for STL containers. If the value_type of an STL container (vector, deque or list) or set is streamable to an iostream, then it is possible to directly access the elements of the container as a simple list:

```
std::vector<int> vec;
make_model(vec);
   ...
vec {1 2 3}
set vec_elems [vec]
```

If the value_type is not streamable, an exception will be thrown. This feature makes the #members functionality of sets redundant.

The following TCL procedures are defined for the following STL containers, which can be used from a TCL script or the object browser to manipulate STL container objects. Procedures that do not call member names are prefixed by the "@" symbol, which is a valid identifier character in TCL, but is not a valid C++ identifier character. This avoids any possible clash of member names.

vector, dequeue, list

@is_vector A "do nothing" command, the presence of which indicates the object is a vector. @elem is more efficient in this case

@is_sequence A "do nothing" command, the presence of which indicates the object complies with the sequence concept.

size returns the size of the vector

@elem takes one argument, the *index* of an element. It creates a TCL command *name*(*index*) that can be used in the usual way to access or modify the element's value.

set, map

@is_set A "do nothing" command, the presence of which indicates the object complies with the set concept.

@is_map A "do nothing" command, the presence of which indicates the object complies with the map concept. @elem can be used to lookup elements by key.

size Return number of entries in the set or map

count Takes a single argument, and returns 1 or 0, according to whether that argument present within the set or map (as a member or key respectively).

#members Returns list of members of a set

#keys Returns list of keys of a map

@elem Returns a TCL command name for accessing individual elements of a set or map. In the case of a set, the command accesses the *i*th element of the set. In the case of a map, the argument can be an arbitrary string (so long as it converts to the key type of the map), that can be used to address the map element. For example, if the map is `map`¡string,string¿+, one can create an element `m["hello"]="foo"` by means of the following TCL commands:

```
m.@elem hello
m(hello) foo
```

13.13.1 Extending TCL_obj_stl for nonstandard containers

`TCL_obj_stl.h` uses the (see §13.1.8) and 13.1.8 type trait. This means that to enable TCL_obj handling of your custom container `MyContainer`, you need to define the appropriate type trait prior to including `TCL_obj_stl.h`, eg:

```
namespace classdesc
{
  template <> struct is_sequence<MyContainer>: public true_type {};
}
```

Also, if your custom container is more of a map, then you also need to define the `ecolab::is_map` type trait for it's `value_type`, eg

```
namespace classdesc
{
  template <> struct is_associative<MyContainer>: public true_type {};
}
namespace ecolab
{
  template <> struct is_map<MyContainer::value_type>:
    public is_map<std::pair<MyContainer::key_type, MyContainer::mapped_type> > {};
}
```

14 Graphcode

14.1 Graph

A *Graph* is a container of references to (§14.2) (called (§14.3)) that may be linked to an arbitrary number of other objects. The objects themselves may be located on other processors, ie the Graph may be distributed. Objects are polymorphic — the only properties Graph needs to know is how create, copy, and serialise them, as well as what other objects they are linked to.

Because the objects are polymorphic, it is possible to create hypergraphs. Simply have two types of object in the graph — *pins* and *wires*, say. A pin may be connected to multiple wire objects, just as wires may be connected to multiple pins.

The objrefs themselves are stored in a maplike object called an (§14.5), which is replicated across all processors.

A short synopsis of Graph is as follows:

```
class Graph: public Ptrlist
{
public:
  omap objects;

  Graph& operator=(const Graph&);
  Graph(Graph&);
  Graph();

  /* object management */
  objref& AddObject(object* o, GraphID_t id, bool managed=false);
  template <class T>
  objref& AddObject(GraphID_t id);
  template <class T>
  objref& AddObject(const T& master_copy, GraphID_t id);

  /* these methods must be called on all processors simultaneously */
  void Prepare_Neighbours(bool cache_requests=false);
  void Partition_Objects();
  void Distribute_Objects();
  void gather();

  void rebuild_local_list();
  void clear_non_local()
  void print(int proc)
};
```

Ptrlist (see §14.4) is a list of references to (§14.3)), pointing to objects stored locally on the current processor.

AddObject In the first form, add an already created object to the Graph. In the second form create a new object of type *T*, and add it to the Graph. *T* must be derived from the abstract base class `object`. You must explicitly supply the type of the object to be created as a template argument:

```
    g.AddObject<foo>(id);
```

In the third form, create a new object, and initialise its data with the contents of argument `master_copy`.

Prepare_Neighbours() For each object on the local processor, ensure that all objects connected to it are brought up to date, by obtaining data from remote processors if necessary. If the network structure has not changed since the last call to this method, set the flag `cache_requests` to `true`, which substantially reduces the amount of interprocessor communication required.

Partition_Objects() Call the ParMETIS partitioner to redistribute the graph in an optimal way over the processors. ParMETIS executes in parallel, and requires that the objects be distributed before this call. One way of achieving this is to make a simple assignment of objects to processors (by setting the `proc` member of each objref), then call `Distribute_Objects()`.

Distribute_Objects() Broadcast graph data from processor 0, and call `rebuild_local_list()` on each processor.

gather() Bring the entire graph on processor 0 up to date, copying information from remote processors as necessary. A `gather()`, followed by `Distribute_Pins()` brings all processors' graphs up-to-date. This is, naturally, an expensive operation, and should be done for startup or checkpointing purposes.

rebuild_local_list() Reconstruct the list of objrefs local to the current processor, according to the `proc` member of the objrefs.

clear_non_local() Nullify all objrefs that don't belong to the current processor. This can be used to save memory usage.

14.1.1 Basic usage of Graph

Graph is designed to be used in a SPMD parallel environment, using MPI to handle messages between processors. A copy of the Graph object is maintained on each process. Each process has a copy of the objref database (of type `omap`), called GRAPHCODE_NS::objectMap. The Graph::objects reference refers to this database. However the payload pointer of each objref will tend to only point to an object if the object is located in the current processes address space, or a cached copy of the remote object is needed for some reason. Otherwise it may be set to NULL to save space.

To call a method foo() on all objects of a Graph g (in parallel), execute the following code:

```
for (Graph::iterator i=g.begin(); i!=g.end(); i++)
   (*i)->foo();
```

If the method foo needs to know the values of neighbouring nodes, then you may call Graph::Prepare_Neighbours(), which ensures that a cached copy of any remotely located node linked to a local nodes is retrieved from the remote node. Thus arbitrary communication patterns can be expressed simply by the form of the network structure of the Graph.

14.2 Objects

object is an abstract base class, from which all objects stored in the graph must be derived. A synopsis of the ABC is:

```
  class object: public Ptrlist
  {
  public:
    /* serialisation methods */
    virtual void lpack(pack_t *buf)=0;
    virtual void lunpack(pack_t *buf)=0;
    /* virtual "constructors" */
    virtual object* lnew() const=0;
    virtual object* lcopy() const=0;
    virtual ~object() {}
    virtual int type() const=0;     /* return index into archetype table */
    /* partition weightings - redefine in derived type if needed */
    virtual idxtype weight() const {return 1;}
    virtual idxtype edgeweight(const objref& x) const {return 1;}
  };
```

14.2.1 Serialisation methods

The first two virtual methods allow Graphcode to access Classdesc generated serialisation routines. Assuming you have declared a class foo as follows:

```
class foo: public object
{
  ...
  virtual void lpack(pack_t *buf);
  virtual void lunpack(pack_t *buf);
}
```

Then you may define the virtual functions as follows:

```
inline void pack(pack_t *,eco_string,foo&);
inline void unpack(pack_t *,eco_string,foo&);
inline void foo::lpack(pack_t *buf) {pack(buf,eco_string(),*this);}
inline void foo::lunpack(pack_t *buf) {unpack(buf,eco_string(),*this);}
```

The definitions for `pack(,,foo&)` and `unpack(,,foo&)` will then be created in the usual way by Classdesc.

It is important that `pack(,,foo&)` and `unpack(,,foo&)` be explicitly declared before use, otherwise a default template function will be linked in which will not work as expected. See the note on using polymorphic objects under Classdesc.

14.2.2 Virtual Constructors

Defining the virtual constructors for your objects type is also a simple matter. Unlike the case of the serialisation routines, they can even be done inline in the class definition:

```
class foo: public object
{
 ...
 virtual object *lnew() {return vnew(this);}
 virtual object *lcopy() {return vcopy(this);}
};
```

14.2.3 Run Time Type Identification

To migrate an object from one thread to another, Graphcode needs to be able to create an object of the correct type in the destination address space. This is achieved by means of a *run time type identification* (RTTI) system. Given a type token `t`, an object of that type can be created by the call:

```
object *o=archetype[t]->lnew();
```

Instead of using C++'s built-in RTTI system, where tokens are compound objects of somewhat indeterminate size, Graphcode implements a simple RTTI system using template programming, in which a type token is a simple unsigned integer. This implies that each type of object to be used with Graph must be registered first, before use. This is taken care for you automatically if you use `Graph::AddObject()` to add your object to the Graph.

Adding the virtual type method to your class is also easy:

```
class foo: public object
{
 ...
 virtual int type() {return vtype(*this);}
};
```

The first `vtype` is called on an object, an object of that type is created (via its `lcopy` method), and added to the archetype vector. The index of that object within the archetype vector become the type token. *It is vitally important that types are added to the archetype vector in the same order on all threads.* Clearly this is a trivial requirement if only one type is used, but slightly more care needs to be taken in the case of multiple types of object.

If you have multiple object types, consider using the `register_type` template to ensure a consistent type registration across the different address spaces.

14.2.4 Node and edge weights

By default, the graph partitioning algorithm used in Graphcode weights each node and link equally. However, it is possible to perform load balancing by specifying a computational weight function on each node, and a communication weight function for each edge. For example:

```
class foo: public object
{
  ...
  virtual idxtype weight() {return size()*size();}
  vitural idxtype edgeweight(const objref& x) {return (*x)->size();}
};
```

14.2.5 Edge list

An object is derived from (§14.4), which contains a list of objrefs that the current object is connected to.

14.2.6 Self linking nodes

If there is any reason for your node to access its objref (eg to find out its GraphID, for example), then you can add the objref to its edge list (say the first item on the edgelist by convention). Then you can refer to things like `begin()->ID`, `begin()->proc` etc.

The `Graph::Prepare_Neighbours()` and `Graph::Partition_Objects()` methods ignore self-linking edges.

14.3 objref

For every object in the Graphcode system, there is an `objref` on every processor referring to it.

Synopsis:

```
class objref
{
public:
  GraphID_t ID;
  unsigned int proc;

  objref(GraphID_t id=0, int proc=myid(), object *o=NULL);
  objref(GraphID_t id, int proc, object &o);
  objref(const objref& x);
  ~objref();

  objref& operator=(const objref& x);
  object& operator*();
  object* operator->();
  const object* operator->() const;
  const object* operator->() const;
  void addref(object* o, bool mflag=false);
  bool nullref() const;
  void nullify();
};
```

ID A unique integer value that identifies the object within a Graph

proc The location of the active copy of the object

`*, ->` Dereferencing an objref allows one to access the object. It is an error to dereference a nullified objref.

addref Add an object to this reference. The `mflag` parameter indicates whether the object is managed (created by `new` or `lnew()`), and can be safely `deleted` when nullified, or is an external object that should not be deleted. An objref can also be instantiated already pointing to an unmanaged object via its constructor.

nullref whether this objref points to an object or not. If the active copy is on this processor, then nullref should be false. Otherwise, it may true or false depending on whether this processor has a cached copy of the object.

nullify Remove the object from the Graph (but leaving the objref in place). It is an error to remove the active copy, without replacing it with another object.

14.4 Ptrlist

Ptrlists work a bit like `std::vector<objref>` — objrefs can be added with `Ptrlist::push_back()`, indexed with standard array indexing `Ptrlist::operator[]`, and iterated over in the usual way with `Ptrlist::iterator`. However, unlike `std::vector<objref>`, only pointers to the objref is stored within `Ptrlist`, not copies.

Synopsis:

```
class Ptrlist
  {
  public:

    class iterator
    {
    public:
      objref& operator*();
      objref* operator->();
      iterator operator++();
      iterator operator--();
      iterator operator++(int);
      iterator operator--(int);
      bool operator==(const iterator& x);
      bool operator!=(const iterator& x);
    };
    iterator begin() const;
    iterator end() const;
    objref &    front ()
    objref &    back ()
    unsigned size() const;
    objref& operator[](unsigned i) const;
    void push_back(objref* x);
    void push_back(objref& x);
    void erase(GraphID_t i);
    void clear();
    Ptrlist& operator=(const Ptrlist &x);
  };
```

Ptrlists can only refer to objects stored in objectMap. Ptrlists can be serialised — Ptrlists must be unpacked within the context of an omapthe objectMap.

If you need to use a backing map, you can declare another omap object and assign objectMap to it. This will create copies of all the objects contained within objectMap.

14.5 omap

An omap is a container for storing §(14.3), indexed by ID.

```
  class omap: public MAP
  {
  public:
    objref& operator[](GraphID_t i);
    omap& operator=(omap& x);
  };
```

There are a few different possible ways of implementing omaps, with differing performance characteristics. Graphcode provides two different models, *vmap* and *hmap* that may be readily deployed by the user, however users can fairly easily provide their own implementation if desired. Different implementations can be selected by defining the `MAP` macro to be the desired omap implementation before including `graphcode.h`. This will declare everything in the namespace `graphcode_vmap` or `graphcode_hmap` as appropriate. Using this scheme, it is possible to have two different omap types in the one object file, by including graphcode.h twice. However, if you do this, you will need to `#undef GRAPHCODE_H` guard variable prior to subsequent includes.

vmap is intended for use with contiguous GraphID ranges. If there are holes in the identifier range, then the iterator will return invalid references for these holes, and the size() method will be incorrect.

If you need to have non-contiguous ID ranges (perhaps for dynamic graph management — note this is not currently supported), then please use the hmap implementation instead (which will have some performance penalty).

MAP must provide the following members:

```
class MAP
{
  protected:
    objref& at(GraphID_t i);
  public:
    MAP();
    MAP(const MAP&)
    class iterator
     {
       iterator();
       iterator(const iterator&);
       iterator& operator=(const iterator&);
       iterator operator++(int);
       iterator operator++();
       iterator operator--(int);
       iterator operator--();
```

```
      bool operator==(const iterator& x) const;
      bool operator!=(const iterator& x) const;
      objref& operator*();
      objref* operator->();
    };
   iterator begin();
   iterator end();
   unsigned size();
}
```

The `at` method is essentially a replacement for `operator[]()`. A simple example of an omap implementation is provided by `vmap`:

```
class vmap: public std::vector<objref>
{
protected:
  objref& at(GraphID_t i)
  {
    if (i>=size()) resize(i+1);
    return std::vector<objref>::operator[](i);
  }
};
```

hmap is a hash map implementation. With all hash maps, performance of the map is critically dependent upon the choice of hash function, which is application dependent. hmap is simply defined as:

```
class hmap: public hashmap<simple_hash> {};
```

You can provide your own omap definition (umap, say), with your own user defined hash function in the following way:

1. Create a file "umap" somewhere in the default search path with the following:

   ```
   #include "hashmap.h"
   struct myhash
   {
     unsigned operator()(GraphID_t i) {...}
   };
   class umap: public hashmap<myhash> {};
   ```

2. Add the new omap definitions to the Graphcode library:

   ```
   make MAP=umap
   ```

3. Include the declarations of the `graphcode_umap` namespace in your application source file:

```
#define MAP umap
#undef GRAPHCODE_H
#include <graphcode.h>
```

14.5.1 Note on using macros to parametrise omap

It might seem obvious that omap could be made a templated type, with a single template parameter being the implementation of the omap. Unfortunately, omaps contain objects, which in turn have Ptrlists (the edge list), which used to maintain a reference to an omap. So objects and objrefs must be similarly parametrised by by the omap implementation type. However the omap implementation type will also need to reference the parametrised objrefs, leading to an implementation type parametrised by itself, an unparsable situation.

The way out of this dilemma is to make omap an abstract base class, which can be made into a concrete implementation as part of the definition of Graph. However, this introduces an extra level of overhead in calling virtual functions when performing object lookup, which is not insignificant. It also dramatically increases the complexity of coding `object::iterator`, which also would need to be an abstract base class.

In view of this, the macro based solution finally chosen, seemed the cleanest, and most efficient means of implementing different omap implementations.

Obviously, this reason is no longer relevant, however there doesn't seem to be any burning reason to change the macro parametrisation to a template argument.

14.6 Building Graphcode

In building graphcode, you configure the options you want by setting Make variables on the command line:

MPI=1 Build the MPI version of Graphcode

MAP= specify which map to use for omap. hmap is the default. You can build a library supporting multiple different omap types by issuing successive make commands:

```
make MAP=vmap
make MAP=hmap
```

DEBUGGING=1 Used to enable assertions, as well as debugger symbols. Optimisation is turned off

PREFIX= specify an install directory when building the `install` target (default `~/usr`).

14.7 Using graphcode in parallel

To use graphcode in parallel, you need to install Classdesc, ParMETIS and MPI. Define the preprocessor symbol `MPI_SUPPORT` to enable the parallel processing code. An example Makefile for the `poisson_demo` example illustrate how this is done.

You will need to arrange the class definitions for your objects, as well as the graphcode.h file to be processed by classdesc. One way of doing this is to `#include` .cd files into one of the C++ source files, and have a `.h.cd` rule in your Makefile, as suggested in the classdesc documentation.

14.8 Examples

Graphcode comes with an example of solving the Poisson equation. Graphcode is also deployed for implementing the spatial Ecolab model, and the Palauan jellyfish model within the *EcoLab* package. The latter model illustrates a dynamic load balancing.

15 tcl++

This section describes a light weight class library for creating Tcl/Tk applications. [5] It consists of a header file `tcl++.h`, and a main program `tclmain.cc`. All you need to do to create a Tcl/Tk application is write any application specific Tcl commands using the `NEWCMD` macro, link your code with the `tclmain` code, and then proceed to write the rest of your application as a Tcl script. The `tclmain` code uses argv[1] to get the name of a script to execute, so assuming that the executable image part of your application is called `appl`, you would start off your application script with

```
#!appl
...Your Tcl/Tk code goes here...
```

then your script becomes the application, directly launchable from the shell.

EcoLab processes any pending TCL events prior to executing the C++ implementation method. This keeps the GUI lively. However, it may cause problems if TCL scripts depend on C++ state that may be changed by event processing. There is a boolean global variable `processEvents`, that disables the background even processing if set to false. TCL commands `enableEventProcessing` and `disableEventProcessing` can be used to set/clear this flag from the TCL interface.

15.1 Global Variables

The following global symbols are defined by `tcl++`:

interp The default interpreter used by Tcl/Tk

mainWin The window used by Tk, of type `Tk_Window`, useful for obtaining X information such as colourmaps

error To flag an error, throw a variable of type `error`. Use `error(char *format,...)` to construct such an object. The format string is passed to sprintf, along with all remaining arguments. The resulting string is what is return by the `error::what()` method. Standard execptions are usually caught, and the error message return by `what()` is attached to TCL's error log, and `TCL_ERROR` return back to the interpreter.

15.2 Creating a Tcl command

To register a procedure, use the `NEWCMD` macro. It takes two arguments, the first is the name of the Tcl command to be created, the second is the number of arguments that the Tcl command takes.

Here is an example.

```
#include "tcl++.h"

NEWCMD(silly_cmd,1)
{printf("argv[1]=%s\n", argv[1]);}
```

This creates a Tcl command that prints out its argument.

15.3 tclvar

Tcl variables can be accessed by means of the tclvar class. For example, if the programmer declares:

`tclvar hello("hello"); float floatvar;`

then the variable hello can be used just like a normal C variable in expression such as

`floatvar=hello*3.4;`.

The class allows assignment to and from `double` and `char*` variables, incrementing and decrementing the variables, compound assignment and accessing array Tcl variables by means of the C array syntax. The is also a function that tests for the existence of a Tcl variable.

The complete class definition is given by:

```
class tclvar
{
 public:

/* constructors */
  tclvar();
  tclvar(char *nm, char* val=NULL);
  tclvar(tclvar&);
  ~tclvar();

/* These four statements allow tclvars to be freely mixed with arithmetic
 expressions */
  double operator=(double x);
  char* operator=(char* x);
  operator double ();
  operator char* ();

  double operator++();
  double operator++(int);
  double operator--();
  double operator--(int);
  double operator+=(double x);
  double operator-=(double x);
  double operator*=(double x);
  double operator/=(double x);

  tclvar operator=(tclvar x);
/* arrays can be indexed either by integers, or by strings */
  tclvar operator[](int index);
  tclvar operator[](char* index);

  friend int exists(tclvar x);
};
```

15.4 tclcmd

A tclcmd allows Tcl commands to be executed by a simple `x <<` *Tcl command* syntax. The commands can be stacked, or accumulated. Each time a linefeed is obtained, the command is evaluated.

For example:

```
tclcmd cmd;

cmd << "set a 1\n puts stdout $a";
cmd << "for {set i 0} {$i < 10} {incr i}"
cmd << "{set a [expr $a+$i]; puts stdout $a}\n";
```

The behaviour of this command is slightly different from the usual stream classes (but identical to `eco_strstream` from which it is derived, as a space is automatically inserted between arguments of a `<<`. This means that arguments can be easily given - eg

```
cmd << "plot" << 1.2 << 3.4 << "\n";
```

will perform the command `plot 1.2 3.4`. In order to build up a symbol from multiple arguments, use the append operator. So

```
(cmd << "plot"|3) << "\n";
```

executes the command `plot3`.

15.5 tclreturn

The `tclreturn` class is used to supply a return value to a TCL command. It inherits from `eco_strstream`, so has exactly the same behaviour as that class. When this variable goes out of scope, the contents of the stream buffer is written as the TCL return value. It is an error to have more than one `tclreturn` in a TCL command, the result is undefined in this case.

15.6 tclindex

This class is used to index through a TCL array. TCL arrays can have arbitrary strings for indices, and are not necessarily ordered. The class has three main methods: `start`, which takes a `tclvar` variable that refers to the array, and returns the first element in the array; `incr`, which returns the next element of the array; and `last` which returns 1 if the last element has been read. A sample usage might be (for computing the product of all elements in the array `dims`):

```
tclvar dims="dims";
tclindex idx;
for (ncells *= (int)idx.start(dims); !idx.last(); ncells *= (int)idx.incr() );
```

The class definition is given by:

```
class tclindex
{
```

```
public:
  tclindex();
  tclindex(tclindex&);
  ~tclindex() {done();}
  tclvar start(tclvar&);
  inline tclvar incr();
  tclvar incr(tclvar& x) {incr();}
  int last();
};
```

16 arrays

This section documents the array and related data types. Arrays are dynamically sized (like std::vector), but are numerical, so can be used in numerical array expressions like `a+b=0;`. Array expressions use the expression template technique[12] to maximise the performance of array expressions.

The code has been optimised for use with Intel's vectorising compiler, and code will be generated using SSE instructions if compiled with icc.

Arrays and associated functions are defined in the `array_ns` namespace.

16.1 array

```
//E is an array expression - eg array<double> or array<double>+array<int>
//S is a scalar data type - eg int
//T must be a C plain old data type, usually numerical

template <class T>
class array
{
 public:
  typedef T value_type;

  size_t size();   //return size of array
  T* data();       //return pointer to array's contents

  explicit array(size_t s=0); //construct array of size s
  array(size_t s, T val);     //construct and initialise array of size s
  void resize(size_t s);      //resize array, data undefined

  array& array(const E& x);   //copy constructor
  array& operator=(const E& x); //assignment or broadcast
  T& operator[](size_t i);    //reference an element
  T& operator[](const E& i);    //vector indexing

  operator+() etc.    //see below
};
```

16.1.1 Array operators

Arithmetical operators `+,-,*,/,%` are defined elementwise between array expressions, as well as broadcastwise if one argument is a scalar. For example if x and y and z are arrays, and a is a scalar,

```
  z=x+a*y    =>  for (i=0; i<idx.size(); i++) z[i]=x[i]+a*y[i];
```

Indexing operators [] can either take a single integer argument, which refers to a single array or expression element, or it can take an integer array expression, which performs vector indexing. So

```
y=x[idx];     =>  for (i=0; i<idx.size(); i++) y[i] = x[idx[i]];
y[idx]=x;     =>  for (i=0; i<idx.size(); i++) y[idx[i] = x[i];
```

Comparison operators `<`, `>` etc, and logical operators `&&`, `||` are also defined in elementwise and broadcast versions.

`operator<<(expression1,expression2)` is a concatenation operator, appending the elements of expression2 to the end of the elements of expression1.

Compound assignment variants also exist:

```
x+=y;    =>   x=x+y;
x*=y;    =>   x=x*y;
   ...
x<<=y;   =>   x=x<<y;
```

16.2 sparse_mat

The `sparse_mat` data type is designed to hold the interaction matrix $\boldsymbol{\beta}$, which is generally a sparse matrix. It is built up of array¡double¿ and iarray¡int¿ components:

```
class sparse_mat
{
 public:
  array diag, val;
  iarray row, col;
  sparse_mat(int s=0, int o=0)
    {diag=array(s); val=array(o); row=iarray(o); col=iarray(o);}
  array operator*(iarray& x);  /* matrix multiplication */
  sparse_mat operator=(sparse_mat x);
  void init_rand(int conn, double sigma);
};
```

`diag` stores the diagonal components of the array, `val` is the packed list of offdiagonal values, with `row` and `col` being the index lists. The only important operator defined for this class is the matrix operation, which is defined as

```
beta*x == beta.diag*x + (beta.val*x[beta.col])[beta.row]
```

but is implemented separately for efficiency reasons.

`init_rand` is a utility routine for randomly initialising the nonzero pattern of the offdiagonal elements. The average number of nonzeros per row is `conn`, and the standard deviation of the number of nonzeros is `sigma`.

It is possible to represent the offdiagonal array differently for efficiency reasons. For example, if it is desired to represent the offdiagonal elements as a dense 2D array, one can create an extra pointer in the underlying implementation of array. Most of the time, this pointer is NULL, but when the `cs_arrays` routine `offmul` is called, it will check this pointer for the `val` array. If it is NULL, it will create the efficient representation, otherwise it will reuse the existing one. Because the only way the array's actual value will get out of synch with the efficient representation is by index assignment, `put_double()` and `put_double_array` (as well as obviously `delete_array()`) will need to be modified to deallocate the efficient representation.

16.3 Global functions

- array `merge`(iarray *mask*, array *a*, array *b*)

  return an array (or iarray) *r* such that where *mask[i]==1*, *r[i]=a[i]*, otherwise *r[i]=b[i]*.

- array `pack`(array *x*, iarray *mask*, [int *ntrue*])

  Construct a new array from elements of *x* that correspond to where the mask is true. *ntrue* is an optional parameter equal to the number of true elements of *mask*. If not given, then `pack` will count the true elements of *mask*. Give this parameter if you are using the same *mask* in multiple `pack` statements.

- iarray `enumerate`(iarray *mask*)

  Return the running sum of mask.

- iarray `pcoord`(int *size*)

  return [0..size-1]

- iarray `gen_index`(iarray *x*)

  generate a list of index values, with each number appearing in the list according to the value passed in its position. For example, if *x*={0,0,1,2,0,1} then the output from this function will be {3,4,4,6}

- void `lgspread`(array& *a*, array *s*)

  Perform lognormal variation of the components of *a*, with standard error given by *s*, ie

  $$a_i = \mathrm{sgn(a_i)\exp(\log|a_i| + s_i\xi_i)}$$

  where ξ_i is normal random variate.

- void `gspread`(array& *a*, array *s*)

Perform normal variation of the components of a, with standard error given by s, ie

$$a_i = a_i + s_i\xi_i$$

where ξ_i is normal random variate.

16.3.1 Reduction functions

- double `max`(array x)
- double `min`(array x)
- double `sum`(array x)
- bool `all`(array x) return true is all of x[i] are nonzero
- bool `any`(array x) return true if any x[i] are nonzero

16.3.2 Array functions

- array `abs`(array x)
- iarray `sign`(array x)
- array `exp`(array x)
- array `log`(array x)
- array`<int>` ProbRound(array`<double>` x) Round up or down with probability given by the difference between real value and the integers either side. For instance, if the argument's value is 0.2, then it has an 80% chance of being rounded down to 0, and a 20% chance of being rounded up to 1. Uses `array_urand`§16.3.3).

16.3.3 Random number functions

array's random number functions allow arrays to be filled with random numbers efficiently in a single call. Most of these functions use the `array_urand` uniform random object (of type urand§19)), which is accessible from TCL. The `fillgrand()` function makes use of the TCL accessible `array_grand` object, which is of type gaussrand§19).

- void `fillrand`(array x)

  Fill x with random numbers from the uniform distribution over $[0, 1]$

- void `fillprand`(array x)

  Fill x with random numbers from the Poisson distribution e^{-x}

- void `fillgrand`(array x)

  Fill x with random numbers from the Gaussian distribution $e^{-x^2/4}$

- void `fill_unique_rand`(array x, int max)

  Fill x with random integers from the range $[0..max]$ such that no two integers are the same.

16.3.4 Stream Functions

- `ostream& operator<<(ostream& s, expr x)`
- `istream& operator>>(istream& s, array& x)`

16.4 Old arrays

The original array library had a somewhat different API. The new array library can be accessed through the old interface by including `#include "oldarrays.h"` The only slight difference that may be noticed is that oldarray::size is not an integer, but an object that converts to an integer. The problem is that overloaded functions may ge confused.

17 tcl_arrays.cc

This module defines number of utility TCL commands for initialising array variables from TCL. Aside from `srand`, they all return a list of values, suitable for initialising an array:

17.1 list generation

srand *seed* Initialises the random seed

unirand *size* [*max* [*min*]] Returns a list (of length *size*) of uniform random numbers, in the range *min–max*. *max* and *min* default to 1 and 0 respectively.

grand *n* [*std* [*mean*]] Returns a list (of length *size*) of normally distributed random numbers, with mean *mean* and standard deviation *std*. *std* and *mean* default to 1 and 0 respectively.

prand *n* [*std* [*mean*]] Returns a list (of length *size*) of Poisson distributed random numbers, with mean *mean* and standard deviation *std*. *std* and *mean* default to 1 and 0 respectively.

unique_rand *size* [*max* [*min*]] Returns a list of unique random integers of length *size* between *min* and *max*, which default to 0 and *size* respectively.

constant *size value* returns a list of *size* elements, each of value *value*.

pcoord *size* returns a list of *size* elements, whose values are the numbers 0. . . *size*.

17.2 Reduction Functions

Syntax:

`max` *tcllist*
`min` *tcllist*
`av` *tcllist*

The maximum, minimum and average of a TCL list.

18 Statistical Analysis

The Analysis module contains some simple statistics support in the form of a TCL exported C++ data type (§3). The class definition is:

```
struct Stats: public array_ns::array<float>
{
  double sum, sumsq;
  float max, min;
  Stats(): sum(0), sumsq(0), max(-std::numeric_limits<float>::max()),
           min(std::numeric_limits<float>::max()) {}

  void clear();
  double av();
  double median();
  double stddev();

  Stats& operator<<=(float x);
  Stats& operator<<=(const array_ns::array<float>& x);
  Stats& operator<<=(const array_ns::array<double>& x);

  void add_data(TCL_args args);
};

struct HistoStats: public Stats
{
  unsigned nbins;
  bool logbins;
  HistoStats(): nbins(100), logbins(false) {}
  array_ns::array<double> histogram();
  array_ns::array<double> bins();

  double loglikelihood(TCL_args args);

  array_ns::array<double> fitPowerLaw(); //< fit x^{-a} - return a and x_min
  double fitExponential()                //< fit exp(-x/a) - return a
  array_ns::array<double> fitNormal();   //< fit exp(-(x-m)^2/2s, return m, s

  array_ns::array<double> fitLogNormal(); //< fit exp(-(log(x)-log m)^2/2s, return m, s
};
```

These classes can be used from the TCL programming environment like this example:

```
HistoStats h


unuran rand
rand.set_gen {distr=pareto(.5,1.7);}

for {set i 0} {$i<10000} {incr i} {
    h.add_data [rand.rand]
}

# obtain average, median, min, max, standard deviation and no. samples
puts stdout "[h.av] [h.median] [h.min] [h.max] [h.stddev] [h.size]"

# fit power law distribution, returning slope and xmin
puts stdout "[h.fitPowerLaw]"

# return log likelihood ratio for power law versus lognormal
array set pl [h.fitPowerLaw]
array set ln [h.fitLogNormal]
set R [h.loglikelihood powerlaw($pl(0),$pl(1))
                       lognormal($ln(0),$ln(1) $pl(1)]
puts stdout "R=$R p=[expr fabs([erfc $R])]"
```

Fitting parameters is achieved using the likelihood method as described in [1].

The log likelihood ratio function returns $\mathcal{R}/\sqrt{2n}\sigma$, where $\mathcal{R} = \ln\prod_i p_1(x_i)/p_2(x_i)$ is the logarithm of the ratio of likelihoods for the two distributions p_1 and p_2.

If the log likelihood ratio is positive, it means the p_1 is more likely to fit the data than p_2, and negative is the other way around. $p = |\text{erfc}(\mathcal{R})|$ is the probability that this conclusion is wrong.[1]

The histogram function allows one to do histograms without using the GUI widget, which is useful for larger collections of data. The parameters to the histogram method are the number of bins (default 100) and whether linear or logarithmic binning is used.

19 random.cc

This module abstracts the concept of a random number generator. The base types are:

```
class random_gen
{
public:
  virtual double rand()=0;
};

class affinerand: public random_gen
{
public:
  double scale, offset;
  affinerand(double s=1, double o=0, random_gen *g=NULL);
  void Set_gen(random_gen*);
  void set_gen(TCL_args args) {Set_gen(args);}
  template <class T> void new_gen(const T&);
  double rand() {return scale*gen->rand()+offset;}
};
```

`random_gen` is an abstract base class representing a random number generator. `affinerand` performs a simple affine transformation on the contained random generator. The `random_gen*` argument to `affine_rand()` may be used to set the random generator employed. If `NULL` (default) is passed for this parameter, a uniform random generator of type urand is created and used. After creation, you can either set the random generator to be something else using `Set_gen`, or create a new random generator of the same type as the passed argument using `new_gen`. The `set_gen` method is callable from TCL, and takes a named `random_gen` object as an argument. The class's destructor will delete a generator created with newgen, but not delete an object passed by setgen. For example, a normal distribution with mean m and standard deviation s can be obtained with the declaration:

```
affinerand(s,m).newgen(gaussrand());
```

The other classes are:

```
class urand: public random_gen
{
public:
  double rand();
  void Seed(int s);
  void seed(TCL_args args) {Seed(args);}
#if defined(UNURAN) || defined(GNUSL)
  void Set_gen(char *);
  void set_gen(TCL_args args) {Set_gen(args);}
  urand(const char* descr) {Set_gen(descr);}
#endif
};

class gaussrand: public random_gen
{
public:
  urand uni;
  double rand();
};
```

`urand` simply returns a uniform random variate in [0,1]. The basic *EcoLab* code contains a uniform generator which is simply an interface to the standard library `rand()` call, and a Gaussian random generator, which is based on the algorithm described in Abramowitz and Stegun (1964) sec. 26.8.6.a(2).

`gaussrand` returns a normal variate with mean 0 and standard deviation 1. Use `gaussrand` coupled with `affinerand` to change the mean and standard deviation:

```
affinerand gen(2,.5,new gaussrand);
```

defines a gaussian generator variable `gen` with standard deviation 2 and mean .5.

As of *EcoLab*.4.D7, replacements for these routines using freely available libraries, unuran and GNUSL are available. The Makefile will select the UNURAN library[17] if available, otherwise the GNU Scientific Library[18] will be selected. If neither of these packages are available, the original basic behaviour is selected. Please read the section on (§19.2).

Both UNURAN and GNUSL provide a text interface to selecting and configuring the uniform random generator. The method `Set_gen` provides a way passing PRNG parameters[19] to the underlying PRNG generator. By default the Mersenne Twister algorithm is used, which is acknowledged as being one of the most efficient random generator available. The algorithms available through PRNG are:

EICG explicit inversive congruential generator

ICG inversive congruential generator

LCG linear congruential generator

[17]http://statistik.wu-wien.ac.at/unuran/
[18]http://www.gnu.org/software/gsl/
[19]http://statistik.wu-wien.ac.at/prng/prng/doc/prng.html

QCG quadratic congruential generator

MT19937 Mersenne Twister

MEICG modified explicit inversive congruential generator

DICG digital inversive congruential generator

The set available through GNUSL are:

mt19937 Mersenne Twister

ranlxs0,ranlxs1,ranlxs2,ranlxd1,ranlxd2 Lüscher's RANLUX at different levels of precision and luxury.

cmrg combined multiple recursive generator by L'Ecuyer

mrg fifth-order multiple recursive generator by L'Ecuyer, Blouin and Coutre

taus,taus2 maximally equidistributed combined Tausworthe generator by L'Ecuyer

gfsr4 Four-tap shift-register-sequence random-number generator by Ziff.

Note that only the generator can be selected through the GNUSL string interface — parameters cannot be set. The GNUSL documentation recommends mt19937, taus or gfsr4 for general purpose simulation. mt19937 is the default.

```
class distrand: public random_gen
{
public:
  int nsamp;  /* no. of sample points in distribution */
  int width;  /* digits of precision (base 16) used from prob. distribution */
  double min, max;  /* distribution endpoints */
  distrand(): nsamp(10), width(3), min(0), max(1);
  void Init(int argc, char *argv[]);
  void init(double (*f)(double));
  double rand();
};

#ifdef UNURAN
class unuran: public random_gen
{
public:
  urand uni;
  UNUR_RAN *get_gen(); //get pointer to UNUR_RAN object for UNURAN API use
  /* specify a random generator according to unuran's string interface */
  void Set_gen(const char *descr);
  void set_gen(TCL_args args) {Set_gen(args);}
  unuran();
  unuran(const char* descr) {Set_gen(descr);}
  double rand();
};
#endif
```

`distrand` returns a deviate from an arbitrary distribution function (which needn't be normalised) supplied to `Init` (or `init`). The instance variables `nsamp=10`, `width=3`, `min=0` and `max=1` should be modified before calling `init`. `Init` provides a TCL interface — it takes one argument, the name of a TCL procedure implementing the distribution. This class implements the method due to Marsaglia[4]. The UNURAN and GNUSL libraries provide other, perhaps better routines for doing the same things.

Finally, `unuran` allows an arbitrary UNURAN generator to be specified using UNURAN's string interface[20]. For example, `gaussrand` is equivalent to `unuran("normal()")`. The string interface is powerful and comprehensive, with a large number of predefined distributions and methods provided, and the ability to specify arbitrary distributions. It obviates the need to use `distrand`, which is a rather obsolete algorithm.

GNUSL does not provide a string interace to its general nonuniform distributions. For the moment, you will need to implement your own object interfaces to which ever routines you want to use. You may use the gaussrand example as a template.

19.1 TCL interface

Each of the concrete random number types have been declared with TCLTYPE (See §3), so random generator objects can be created at the TCL level. See newman.tcl, or test-dist.tcl for an example.

Using the unuran type, one can easily set up arbitrary distribution functions at runtime.

```
unuran rand
rand.set_gen {distr=cont; pdf="x^2"; domain=(0,10)}
```

19.2 Problems with the basic random number library

19.2.1 Basic urand objects are not independent

Because the basic urand class uses the underlying Unix `rand()` call, objects of type urand are not independent. Rather, they effectively refer to the same object. Setting the seed on one object of this class sets the seed for all objects of that class.

[20]http://statistik.wu-wien.ac.at/unuran/unuran/doc/unuran.html

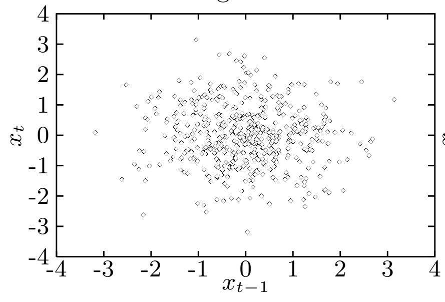
If for example, one wants to perform simulations in parallel, and compare results with different numbers of threads, then the simplest way of arranging this is to ensure that each object in the simulation requiring random numbers gets its own unique stream of random numbers. This can be done by assigning a different urand object to each simulation object, and giving them a distinct seed. Then no matter what the distribution of objects over threads are, the sequence random numbers will be the same from simulation to simulation, allowing meaningful comparisons. The UNURAN and GNUSL libraries both provide independently streamed random numbers.

19.2.2 Basic urand algorithm not efficient

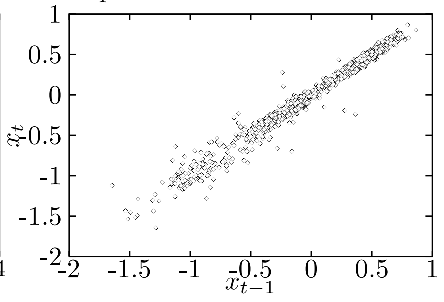
Since the basic urand object depends on Unix `rand()`, it is neither the most efficient algorithm available (the Mersenne Twister is generally better), nor is it necessarily acceptible in all cases. All pseudo-random generators have some kind of structure to their outputs, which may or may not be significant in the simulation. With the UNURAN library or with GNUSL it is possible to select from a range of different algorithms at runtime, making it easy to check that the simulation results do not depend on the type of generator used.

19.2.3 Basic Gaussrand is correlated

The graph of x_t versus x_{t-1} shown below says it all. The correlations of the basic Gaussrand algorithm can well be unacceptable:



UNURAN Gaussian method



Basic Gaussian method

20 Graph library

The *EcoLab* graph library is a library providing a simple and lightweight structure for representing graphs (aka networks). A graph consists of a set of nodes labelled $N = \{0 \ldots n-1\}$, and a set of edges $E \subset N \times N$, which have an optional weight factor attached.

An edge is represented by

```
struct Edge: public std::pair<unsigned,unsigned>
{
    unsigned source();
    unsigned target();
    float weight;
};
```

The abstract Graph interface has the following definition:

```
struct Graph
{
  struct const_iterator
  {
    Edge operator*() const;
    const Edge* operator->() const;
    const_iterator& operator++();
    bool operator==(const const_iterator& x) const;
    bool operator!=(const const_iterator& x) const;
  };

  virtual const_iterator begin() const;
  virtual const_iterator end() const;

  virtual unsigned nodes() const;
  virtual unsigned links() const;
  virtual void push_back(const Edge& e);
  virtual bool contains(const Edge& e) const;
  virtual bool directed() const;
  virtual void clear(unsigned nodes=0);
  const Graph& operator=(const Graph& x);
  void input(const std::string& format, const std::string& filename);
  void output(const std::string& format, const std::string& filename) const;
  template <class BG> Graph_back_insert_iterator<Graph,BG>
    back_inserter(const BG& bg);
}
```

The `begin/end` methods allow one to iterate over the edges. Only a `const_iterator` is supplied, as it is an error to change the value of an edge. One can only reset a graph to the empty graph via clear, and construct the graph incrementally using the `push_back()` method.

The `contains` method allows one to test whether a given edge is in the graph, and `directed` indicates whether the underlying graph structure has directed edges or not. A bidirectional graph otherwise appears as a directed graph where each edge appears twice, once for each direction.

The input/output methods allow for the graph to be read/written from/to a file, in a variety of formats, given by the format parameter. Currently, the following formats are supported:

name	description
pajek	Pajek's .net format
lgl	LGL's .lgl format
dot	Graphviz format
gengraph	Gengraph

Also graphs can be streamed to/from standard I/O streams, and will appear in dot format. In particular, this means that TCL scripts can access Graph objects as strings containing the Graphviz representation. See some of the examples in the `models/netcomplexity_scripts` directory.

`back_inserter` creates an output iterator suitable for use with Boost Graph algorithms. As a simple example, to construct an EcoLab graph from a Boost Graph, do

```
std::pair<BG::edge_iterator,BG::edge_iterator> r=edges(bg);
ConcreteGraph<DiGraph> g1;
copy(r.first, r.second, g1.back_inserter(bg));
```

This interface can be used in both a dynamic polymorphism fashion (ie Graph is an abstract base class) and in a static polymorphism fashion.

The `graph.h` header file provides two concrete graph types - DiGraph and BiDirectionalGraph, which differ just in whether each edge is directed or not.

20.1 Boost

EcoLab has a deliberate policy of not having a dependency on Boost, but to be as compatible with Boost as far as it is possible. The EcoLab graph library has been designed to be interoperable with the Boost Graph Library. The file `test/test_boostgraph.cc` gives some examples of interoperation with Boost.

20.2 Graph Library Functions

`Degrees degrees(const G& g)` returns array of node degrees return as the following structure:

```
struct Degrees
{
  array<unsigned> in_degree, out_degree;
};
```

`void ErdosRenyi_generate(Graph& g, unsigned nodes, unsigned links, urand& uni);`
Generate an Erdös-Rényi random graph with given nodes and links. Links are attached randomly to the nodes, drawn from uni.

`PreferentialAttach_generate(Graph& g, unsigned nodes, urand& uni, random_gen& indegree_dist=default_indegree)`
Barabasi-Albert Preferential attachment algorithm. For each node, an indegree value is drawn from `indegree_dist` (defaults to a constant value of 1). This many links are then preferentially attached to other nodes, according to their outdegree.

20.3 Network complexity

```
class BitRep implements Graph
{
  BiDirectionalBitRep symmetrise() const;
  bool operator()(i,j);
  bool next_perm();
};

class BiDirectionalBitRep implements Graph
{
  bool operator()(i,j);
  bool next_perm();
};

class NautyRep implements Graph
{
  bool operator()(i,j);
  double lnomega() const;
  NautyRep canonicalise() const;
};
```

These classes implement the Graph interface by storing the linklist as a bitset. NautyRep specifically uses the bitset representation of the Nauty package. One can freely convert between these types and others implementing the Graph interface. All of these types support direct setting/testing of the i,j th edge through the operator(i,j).

BitRep and BiDirectionalBitRep allow one to iterate through the permutations (thus cycling over all graph representations of a given edge count). This function returns false when no further permutations exist.

NautyRep has member functions for calling the Nauty library. If you need both the lnomega and canonical representations, then it is more efficient to call them at the same time via the `ecolab_nauty()` function.

```
void ecolab_nauty(const NautyRep& g, NautyRep& canonical, double& lnomega, bool do_canon
```

call Nauty on g, returning a canonical representation canonical and lnomega ($\ln\Omega$). If canonical is not needed, then set `do_canonical` to false.

```
double canonical(BitRep& canon, const Graph& g);
```

SuperNOVA canonical algorithm. Returns $\ln\Omega$ and canonical representation (which needn't correspond to that returned by Nauty).

```
double complexity(const Graph& g);
```

Network complexity. If all links have the same weight, this corresponds to $2\lceil\log_2(n+1)\rceil + \lceil\log_2(n(n-1))\rceil + 1 + \log_2\left(\begin{array}{c} n(n-1) \\ l \end{array}\right) - \log_2\Omega$.

21 eco_strstream class

21.1 eco_string class

```
class eco_string
{
public:
  eco_string();
  eco_string(const eco_string& x);
  eco_string(const char *x);
  ~eco_string();

  eco_string& operator=(const eco_string& x);
  eco_string& operator=(const char* x);
  eco_string operator+(const char* y);
  eco_string& operator+=(const char* y);
  eco_string operator+(const char y);
  eco_string& operator+=(const char y);
  eco_string operator+(const eco_string& y);
  eco_string& operator+=(const eco_string& y);
  operator char *() const;
  int length();
  void freeze(int x);
};
```

Basic string class, broadly compatible with the STL string class, but can be used in the classdesc generated code. The pointer returned by `char *` refers to an ordinary C string, but is no longer valid once the destructor is called. This behaviour can be altered calling `freeze(1)`. Unlike the GNU string class, casting the string to a `char *` does not freeze the string.

21.2 eco_strstream

This is a modified version of ostrstream for *EcoLab* use, with slightly altered behaviour making it more suited for *EcoLab* use.

In particular, `operator<<` add spaces in between its arguments, and a new `operator|` is defined that is similar to ostrstream `operator<<`. This makes it easier to construct TCL commands.

Note that `|` has higher precedence than `<<`, so when mixing the two, ensure that no `<<` operator appears to the right of `|`, or use brackets to ensure the correct interpretaion:

eg.

```
s << a << b | c;
```

or

```
(s | a) << b | c;
```

but not

```
s | a << b | c;
```

In any case, you'll most likely get a compiler warning if you do the wrong thing.

eco_strstream is derived from eco_string class, so has all the string behaviour. The str() member is defiend as equivalent to a (char*) cast for compatibility with ostrstream. However, the contents of the char* are not frozen, unless explicitly specified with freeze() (opposite to ostrstream behaviour).

```
class eco_strstream: public eco_string
{
  public:
    char* str() const;

    template<class T>
    eco_strstream& operator<<(const T x);

    template<class T>
    eco_strstream& operator|(T);
}

inline ostream& operator<<(ostream& x, eco_strstream& y);
```

22 cachedDBM

`cachedDBM` implements a notion of persistent objects. First and foremostly, it has the syntax of std::map, ie it is a template map object, with a key and value type pair. However, by calling the `init()` method, you can attach a database file, so that values saved in the cachedDBM are stored on disk, to be accessible at a later time.

The iterator range `begin()` to `end()` refers to everything stored in the database. An alternative interface that iterates over the database keys is provided by `keys.begin()` and `keys.end()` The database is committed when `begin()` is called, unless the database was opened readonly. In this latter case, there are potentially items stored in the map which will not be iterated over. The alternative iteration methods `firstkey()`, `nextkey()` and `eof()` is an older interface for iterating over `keys.begin()` to `keys.end()`. In this instance, the cachedDBM is not committed when `firstkey()` is called.

Elements stored in the cachedDBM are not actually written to disk until `commit()` is called (or the cachedDBM object is destroyed).

Entries in the database can be removed via `del()`. However, if an item with the same key is in the cache, it will need to be removed via `erase()` as well, otherwise it will be reinserted in the database at commit time.

Only a very simple caching algorithm is employed, but it seems sufficient for many purposes. If the member `max_elem` is set, then this acts as an upper limit to the number of items stored in memory. If you request a new item to be loaded via the [] operator, and it will cause the number of items to be exceeded, the cachedDBM object is committed. At least the oldest quarter of the cache is cleared, and up to half of the cache. So if `max_elem`=100, then one is guaranteed that the previous 50 accessed objects will always be in memory, so can be assigned to a reference.

Classdesc serialisation (XDR serialisation) is used store both keys and value data. The database file are therefore machine independent. As a special exception to serialisation rules, `char*` can be used as key and data types.

22.1 Synopsis

```
template<class key, class val>
class cachedDBM_base : public base_map<key,val>
{
public:
  int max_elem;   /* limit number of elements to this value */
  void init(const char *fname, char mode='w'); /* open database file */
  void Init(TCL_args args); /* TCL access to init() */
  void close();    /* commit and close database file */
  bool opened();   /* is a database attached? */
  bool key_exists(const key& k); /* does the key exist in db */
  val& operator[] (const key& k); /* access element with key k */
  void commit();    /* write any changes out to the file, and clear cache */
  void del(key k); /* delete key from database (but not cache!) */
  key firstkey();  /* return first key in database */
  key nextkey();   /* return next key in database in an iteration */
  int eof();       /* true if all keys have been accessed */
  KeyValueIterator begin() const;
  KeyValueIterator end() const;
  KeyIterator keys.begin() const;
  KeyIterator keys.end() const;
};
```

22.2 TCL access

cachedDBM objects are addressable from TCL. Individual objects stored in the cachedDBM are addressable from TCL by means of the `elem()` method, but also the following members are exported to the TCL interface, allowing scripting access to manipulating the database:

- `cachedDBM::max_elem`
- `cachedDBM::Init` *filename r|w*
- `cachedDBM::close`
- `cachedDBM::opened`
- `cachedDBM::elem` With one argument, get the element value, with two arguments, set it.
- `cachedDBM::commit`
- `cachedDBM::firstkey`
- `cachedDBM::nextkey`
- `cachedDBM::eof`

22.3 Types of database

By default, if EcoLab's Makefile detects the presence of Berkeley DB on your system, it will be used. Otherwise it use the ndbm API. If libgdbm and/or libgdbm_compat (the NDBM compatibility layer in GDBM) these will be added to the linker flags, otherwise it will assume that ndbm is available as part of the standard system library.

You can't mix and match database types. If you have some data stored in one type, and need to access it using a different database type, the utilities convtoNDBM and convtoBDB found in the util directory may be useful.

References

[1] Aaron Clauset, Cosma R. Shalizi, and Mark E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51:661–703, 2009. arXiv:0706.1062.

[2] Barbara Drossel, Paul G. Higgs, and Alan J. McKane. The influence of predator-prey population dynamics on the long-term evolution of food web structure. *J. Theor. Biol.*, 208:91–107, 2001.

[3] Duraid Madina and Russell K. Standish. A system for reflection in C++. In *Proceedings of AUUG2001: Always on and Everywhere*, page 207. Australian Unix Users Group, 2001.

[4] G. Marsaglia. Generating discrete random numbers in a computer. *Comm ACM*, 6:37–38, 1963.

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

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

[7] Russell K. Standish. Population models with random embryologies as a paradigm for evolution. *Complexity International*, 2, 1994.

[8] Russell K. Standish. An Ecolab perspective on the Bedau evolutionary statistics. In Mark A. Bedau, John S. McCaskill, Norman H. Packard, and Steen Rasmussen, editors, *Artificial Life VII*, pages 238–242, Cambridge, Mass., 2000. MIT Press.

[9] Russell K. Standish. Diversity evolution. In Russell Standish, Hussein Abbass, and Mark Bedau, editors, *Artificial Life VIII*, pages 131–137, Cambridge, Mass., 2002. MIT Press. online at http://alife8.alife.org.

[10] Russell K. Standish. Ecolab, Webworld and self-organisation. In Pollack et al., editors, *Artificial Life IX*, page 358, Cambridge, MA, 2004. MIT Press.

[11] Russell K. Standish. Complexity of networks. In Abbass et al., editors, *Recent Advances in Artificial Life*, volume 3 of *Advances in Natural Computation*, pages 253–263, Singapore, 2005. World Scientific. arXiv:cs.IT/0508075.

[12] Todd Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

Index