Index of Swarm documentation (1.0.5)

documenting Swarm 1.0.5

This release of the documentation coincides with version 1.0.5 of the Swarm source code distribution. Though a complete manual for Swarm is still not available, this list of resources should give you a good place to start. Please direct all questions to the official Swarm mailing list: <swarm-support@santafe.edu> (mailto:majordomo@santafe.edu)
.

Table Of Contents:

Introduction

Our home page provides a good introduction to Swarm (what it does and what it aims to be). On the other hand, the apps-intro document aims to give you a more detailed overview of the sorts of things a user needs to do in order to get a simulation up and running in Swarm. The combination of these two documents should help you decide whether Swarm would be a useful tool in the context of your research.

1. 1.0.4 => 1.0.5

   - EZGraph's setGraphWindowGeometryRecordName and GUISwarm's setControlPanelGeometryRecordName have been retired. The macro SET_WINDOW_GEOMETRY_RECORD_NAME can now be used in any geometry archiving context.
   - After adjusting an application per previous item, be aware that the new internal naming conventions used by 'GUIComposite' classes (e.g. EZBin and GUISwarm) will probably differ from the archiving keys in your application. Such widgets will probably lose their saved placements.

2. 1.0.3 => 1.0.4

   - doTkEvents has been moved from ControlPanel to ActionCache. Be sure to direct all doTkEvent messages to the Swarm's instance of ActionCache and not the ControlPanel (a warning message will be generated otherwise).
   - The swarmobject library has been renamed to objectbase. Although the build procedure creates a link to maintain compatibility, applications should include <objectbase.h> instead of <swarmobject.h>.
   - Avoid calling globalTkInterp in all future applications, it will not be supported from version 1.1 onwards.
   - If there exists a <LIBRARY.h> file, e.g. <simtools.h>, that file is the advertised interface to LIBRARY, and it will defined in terms of protocols. Whenever possible, define variables in terms of the protocol they respond to:

   id <ProbeMap> probeMap;

   rather than using a static type:

   ProbeMap *probeMap;

   (One advantage of this is that fewer imports will be needed in your application.)

3. 1.0.2 => 1.0.3

   - Activity Library: Reference Release-Note #3: The AutoDrop option for any concurrent groups is automatically set to the same as the schedule in which it is contained. (Any existing setting of the option on the concurrent group type is ignored.) So, in 1.0.2, if you didn't set the AutoDrop flag for a concurrentGroup initialized under a schedule, it defaulted to NO or false.
   - Random Library: 13, 14

     a. Reference Release-Note #13 This release includes Random v0.7, written by Sven Thommesen. This version adds several new bit generators and distributions and rearranges the library. (Ref. $SWARMHOME/src/random/docs/WHATS.CHANGED.in.0.7 and WHATS.NEW.in.0.7)
     b. Reference Release-Note #14: With the addition of the ability to make use of the default random number generators and distributions such that different runs start with a different seed or with the same seed, the static seed was changed. This means that runs with 1.0.3 will not match runs made with the

- The high-level structural changes in the activity library has led to the renaming of the variable `swarmActivity`. It is recommended that `[self getActivity]` be used in its place (if your app even used this variable, it was probably in the *BatchSwarm*). This is a new message and it takes the place of the `getSwarmActivity` message. But, the obsolete message has been left in place for backwards compatibility.
- The `getCurrentActivity()` macro is gone. If you used the old getCurrentActivity() in your code, it won't work now. Use of this macro was not very widespread, since its main use is to access activity library internals. One of the other macros should be sufficient for any application.

d. The functionality of *Zone*s has been greatly improved and expanded upon. Most of the aspects of the idea behind *Zone*s are now in place. However, there is one incompatibility that must be noted in case your code is fairly old. The dropFrom: message has been removed. Even though it was still present in recent releases, its behavior was identical to drop. Any existing usage should be replaced by a simple drop message without any zone argument. SwarmObject subclasses are now restricted from accessing the zone that was once contained in an instance variable; the message getZone must be used instead.

Objective C Resources

The main technical skill required of a Swarm user is the ability to write Objective C code. Other recommended skills include the ability to use tools such as gdb and emacs. For the record, if you know C and have some form of experience in either C++ or Smalltalk, then learning Objective C should take no more than a day.

Objective C references
(http://www.santafe.edu/projects/swarm/ObjC/objective-c.html)
: Nelson's list of Objective C references on the Web. Includes a 10 minute overview to Objective C that is most of what you need to know about Objective C.

Objective C - Swarm Style
(swarmObjC.html)
: The Swarm system provides a few extensions to the syntax/style of Objective-C. This note describes those features which are important to know as a beginner and those which can be ignored...

Object-Oriented Programming and the Objective-C Language
(http://devworld.apple.com/dev/SWTechPubs/Documents/OPENSTEP/ObjectiveC/objctoc.htm)
: An excellent (online!) book on Objective C. This is for generic NeXT Objective C: Swarm uses GNU Objective C in addition to our own defobj (src/defobj.html) extensions.

debugging tips
(debugging.html)

initial simulations. There is more complete information (the screen shots are a bit dated) about a few of these demo applications in `../examples/` `(http://www.santafe.edu/projects/swarm/examples/index.html)` .

- `Heatbugs, a basic complex system`
  `(apps/heatbugs.txt)`
  `: thoroughly commented!!!`
- `Template application`
  `(apps/template.txt)`
- `Simple model of a stock market`
  `(apps/market.txt)`
- `Mousetrap model of fission, discrete event simulation`
  `(apps/mousetrap.txt)`
- `Tutorial is an app that progressively moves from a simple C implementation of a CA (with bug agents on it) up to a Swarm-style application.`
  `(apps/tutorial.txt)`
- `Hello, World! is another simpler tutorial`
  `(apps/hello-world.txt)`

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Mag 1x: Experimental Procedure in a Computer

At the highest level of abstraction (= the lowest level of magnification), most experiments look like this:

1. Set up the physical system to be studied
2. Set up and calibrate the instrumentation
3. Run the experimental system and record the outputs of the instrumentation
4. Analyze results
5. If not happy with the results:
   - Change experimental and instrumental setup
   - Go to 3
6. Publish paper -> tenure -> fame -> etc.....

The important part of step 6) is that the published paper includes enough detail about the experimental setup and how it was run so that other labs with access to the same equipment can recreate the experiment and test the repeatability of the results. This is hardly *ever* done (or even possible) in the context of experiments run in computers, and the crucial process of independent verification via replication of results is almost unheard of in computer simulation. One goal of Swarm is to bring simulation writing up to a higher level of expression, writing applications with reference to a standard set of simulation tools.

First, let's look at what happens when we port the above stages into the world of a computer. In a computer, you don't just drag the pieces of your experiment in from the outside world and hook them up. You have to create a world with space and time, a bunch of objects in that world (stuff to study and stuff to look at it with), schedules of events over those objects, all sorts of computer widgetry to interact with that artificial world and to manage multiple experimental runs and the data that they generate, and so forth. In other words, in a computer, one usually has to first *create* from scratch all of the bits and pieces of the experimental setup - the virtual equivalent of beakers, bunsen burners, microscopes etc.

Perhaps the most important difference between an experiment in the "real" world and an experiment inside of a computer is the nature of time. In the real world, everything in one's experimental setup is moved forward in time via a very concurrency courtesy of the laws of physics. In a computer experiment, however, the experimenter has to explicitly move every object in his/her artificial universe forward in time, making sure that everything remains within some well-understood state of synchronization. Many fundamental problems in computer science have arisen in the course of trying to understand how to control and use concurrency. Furthermore, most people who implement computer simulations aren't even aware of the subtle, but quite-possibly dominating, impacts of assumptions that they aren't even aware that they are making about concurrency in their model when they code it up and run it.

4. Interact with the experiment via the data produced by the instrumentation objects to perform a series of controlled experimental runs of the system.
5. Depending on what is observed in stage 4), alter the experimental or instrumental "apparatus" and go back to 3).
6. Publish paper *including* detailed specification of the experimental set-up so that others can recreate your experiment and verify your results.

Mag 3x: Swarm Structures

Swarm applications are structured around the concept of the *Swarm*. *Swarms* are the basic building blocks of Swarm simulations: a *Swarm* is a combination of a collection of objects and a schedule of activity over those objects. The collection are like the matter of the Swarm and the schedule is like the arrow of time moving the objects forward.

Model Swarms

For our current demos, Swarm applications contain two swarms. At the core is the *model swarm*, the Swarm that encapsulates the simulated model. Everything in the model swarm corresponds to objects in the world being modeled. For instance, in Heatbugs the HeatbugModelSwarm contains a collection of Heatbug agents, a HeatSpace to represent a physical property of the world, and a Grid2d to store agent position.

> In addition to the object collection, the model swarm also contains a schedule of activity on the model. The schedule defines the effect of passing time on the model. For the simple heatbugs schedule, the execution is simply to update the HeatSpace (diffusing heat across the world) and then telling each Heatbug agent to move itself.
>
> Model swarms consist of a set of inputs and outputs. The inputs to the HeatbugModelSwarm are model parameters: things like the size of the world, the number of HeatBugs, and the diffusion rate of heat. The outputs of the HeatbugModelSwarm are the observables of the model: the individual Heatbugs, the distribution of heat across the world, etc.

ObserverSwarms

The model swarm alone defines the simulated world. But an experiment does not just consist of the objects being experimented upon, it also includes the experimental apparatus used for observation and measurements. In Swarm computer simulations, those observation objects are placed in an *observer swarm*.

> The most important object in an observer swarm is the model swarm that is being studied. The model swarm is one component of the observer, kind of like a little world in a

Building a Model Swarm

The key component of a simulation is the model Swarm. Here is the definition of a HeatbugModelSwarm, from HeatbugModelSwarm.h

```
@interface HeatbugModelSwarm : Swarm {
 int numBugs; // simulation parameters
 double evaporationRate;
 double diffuseConstant;
 int worldXSize, worldYSize;
 int minIdealTemp, maxIdealTemp;
 int minOutputHeat, maxOutputHeat;
 double randomMoveProbability;

 id modelActions; // scheduling data structures
 id modelSchedule;

 id heatbugList; // list of all the heatbugs
 Grid2d * world; // objects representing
 HeatSpace * heat; // the world
}

-getHeatbugList; // access methods into the
-(Grid2d *) getWorld; // model swarm. These methods
-(HeatSpace *) getHeat; // allow the model swarm to
 // be observed.

+createBegin: (id) aZone; // extra methods you
-createEnd; // provide for Swarms
-buildObjects;
-buildActions;
-activateIn: (id) swarmContext;
```

The first section of code says that a HeatbugModelSwarm is a kind of Swarm. HeatbugModelSwarm inherits a lot of behavior from generic Swarm, but also adds new variables and methods.

The new variables are enclosed in the braces in the definition of HeatbugModelSwarm. They are split into three general classes of things: simulation parameters, schedule data structures, and objects in the world. This is a typical sort of model swarm.

Finally, a HeatbugModelSwarm defines new methods. The first few methods are used to allow the model to be observed: a HeatbugModelSwarm will give out its list of Heatbugs, for instance, or its HeatSpace. Observers use these methods to monitor the model.

In addition to the observation methods, there are several Swarm-specific methods for the building of Swarms. These are fairly stereotyped. The `createBegin` and `createEnd` messages are used to create the Swarm object itself. `buildObjects` builds the model objects, and `buildActions` builds the model schedule - more on these later. Finally, `activateIn` arranges for the execution machinery to execute the Swarm itself.

define a getUnhappiness method - the unhappiness is the major measurable aspect of a heatbug, how well optimized it is at the moment. They also have a drawSelfOn method that directs the heatbug to draw itself on the specified graphics widget.

Finally, and most importantly, a Heatbug has a step method. step is where the Heatbugs behavior is defined: every time the Heatbug is told to step it performs its internal calculations, choosing where to move. Each heatbug is told to step when appropriate by the model schedule. The code for step is the real intellectual input into the model, and is worth reading as an example of an agent's behavior.

Building Agents

Now that Heatbugs have been defined, the model swarm needs to create them. This code fragment is from the buildObjects method on HeatbugModelSwarm.

```
// A loop to create a bunch of heatbugs.
for (i = 0; i < numBugs; i++) {
 Heatbug * hbug;
 int idealTemp, outputHeat;

 // Choose a random ideal temperature, output heat from the specified
 // range (model parameters).
 idealTemp = [uniformRandom rMin: minIdealTemp Max: maxIdealTemp];
 outputHeat = [uniformRandom rMin: minOutputHeat Max: maxOutputHeat];

 // Create the heatbug, set the creation time variables
 hbug = [Heatbug createBegin: [self getZone]];
 [hbug setWorld: world Heat: heat];
 hbug = [hbug createEnd];

 // Add the bug to the end of the list.
 [heatbugList addLast: hbug];

 // Now initialize the rest of the heatbug's state.
 [hbug setIdealTemperature: idealTemp];
 [hbug setOutputHeat: outputHeat];
 [hbug setX: [uniformRandom rMax: worldXSize] // random position
 Y: [uniformRandom rMax: worldYSize]];
}
```

The details of this code are best explained in reading the documentation for the libraries. Essentially, we first generate two random numbers: an ideal temperature and an output heat for the new Heatbug. We then create the Hheatbug itself with createBegin and fill in the required parameters of world and heat. Once those are set, we can send createEnd to the Heatbug and it is finished being created. After it's done being created we add it into a list of Heatbugs in the model and set a few parameters on it like the ideal temperature and the initial position.

exactly like building a model swarm.

```
@interface HeatbugObserverSwarm : GUISwarm {
 int displayFrequency; // one parameter: update freq

 id displayActions; // schedule data structs
 id displaySchedule;

 HeatbugModelSwarm * heatbugModelSwarm; // the Swarm we're observing

 // Lots of display objects. First, widgets
 XColormap * colormap; // allocate colours
 ZoomRaster * worldRaster; // 2d display widget
 EZGraph * unhappyGraph; // graphing widget

 // Now, higher order display and data objects
 Value2dDisplay * heatDisplay; // display the heat
 Object2dDisplay * heatbugDisplay; // display the heatbugs
}
```

Again we have input parameters (display frequency), schedule data structures, and resident objects (model swarm, display widgets). The important exception is that HeatbugObserverSwarm is a subclass not just of the generic Swarm class, but specifically a GUISwarm. That implies that the HeatbugObserverSwarm will contain a control panel to allow the user to stop execution, and will also have a special go method to set everything running.

Building a Data Graph

An example of an object inside the HeatbugObserverSwarm is a data graph, the graph of average unhappiness. Here is the code necessary to create that object:

```
// Create the graph widget to display unhappiness.
unhappyGraph = [EZGraph createBegin: [self getZone]];
[unhappyGraph setTitle: "Unhappiness of bugs vs. time"];
[unhappyGraph setAxisLabelsX: "time" Y: "unhappiness"];
unhappyGraph = [unhappyGraph createEnd] ;

[unhappyGraph createAverageSequence: "unhappiness"
 withFeedFrom: [heatbugModelSwarm getHeatbugList]
 andSelector: M(getUnhappiness)] ;
```

The first step is to build an instance of an EZGraph and set its captions. Then a Sequence is created inside that graph (in this case, an AverageSequence). In general, the Sequence requires a target object and a message with which to extract data from that object -- the data is then plotted as one line in the graph.

In the case of an AverageSequence, an entire collection of objects is presented to it. The AverageSequence then extracts data from all the objects in the collection our examples and try to write your own applications. Good luck, and may your simulations be successful!

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

gcc 2.7.x

gcc is the FSF's C, C++, and Objective C compiler. You need gcc 2.7.2 or greater installed on your system, including the Objective C support, libobjc.a. The latest version is available from ftp://prep.ai.mit.edu/pub/gnu/ (ftp://prep.ai.mit.edu/pub/gnu)
.
Note: Swarm will not work with versions of gcc before 2.7.2.

GNU make

Make is used to automate building programs: every OS ships their own version of make with some random set of features. The Swarm makefiles currently use features that are not supported by all makes: GNU make is the only one guaranteed to work. Linux systems probably use GNU make already. The latest version of GNU make should be available on ftp://prep.ai.mit.edu/pub/gnu/ (ftp://prep.ai.mit.edu/pub/gnu/)

gdb

gdb is the GNU debugger: Swarm doesn't require this, but you will probably want gdb on your system to debug programs
(debugging.html)
. Gdb is available on ftp://prep.ai.mit.edu/pub/gnu/
(ftp://prep.ai.mit.edu/pub/gnu/)
.

Prerequisite Libraries

Swarm uses a variety of freely available software libraries. All of these libraries need to be configured and installed independently of Swarm. When you have installed them all, modify Makefile.conf in the Swarm library package to tell the compiler and linker where to find them. Note: it pays to spend the extra time to build shared library versions of these things.

libXpm

libXpm adds pixmap (coloured bitmap) support to X11. libXpm is a common X library, many systems already have it installed. A reasonably modern version is needed: we've used version 3.4f. If you get complaints about "Object" being multiply defined, your version of libXpm is too old. The library is available from ftp://ftp.x.org/contrib/libraries/xpm-3.4h.tar.gz
(ftp://ftp.x.org/contrib/libraries/xpm-3.4h.tar.gz)
.

Tcl/Tk

More common libraries: a scripting language and a GUI toolkit. These libraries are available from ftp://ftp.smli.com/pub/tcl/
(ftp://ftp.smli.com/pub/tcl/)
or from http://www.tcltk.com
(http://www.tcltk.com)

Swarm is set up so that you can run make in subdirectories and expect it to work. If a particular directory doesn't seem to make correctly, try to cd into that directory and run "make" from there to better see what the problem is.

Compiling and Running Swarm Applications

Swarm applications are distributed separately: you will need to download and unpack applications independently. After the application is unpacked, you will have to change the `SWARMHOME` variable in the Makefile. It is setup, by default to point to "../swarm", as if the Swarm libraries were in a sibling directory called "swarm". Just edit that string to point to where you installed the Swarm libraries. No other configuration should be necessary. All you need to do to compile an application is type "make", and it will compile and link.

Once the application is compiled, just run it out of its own directory. A control panel and a couple of parameter windows should pop up; press the "Go" button to watch it run. Congratulations!

Post-compile cleanup

After you've tried out the Swarm libraries for awhile you may want to clean up the intermediate .o files. Swarm is set up so that the only files that you need after compile are `$(SWARMHOME)/Makefile.appl`, `$(SWARMHOME)/Makefile.lib`, `$(SWARMHOME)/Makefile.conf`, and the contents of `$(SWARMHOME)/lib` and `$(SWARMHOME)/include`. If you run `make clean` in `$(SWARMHOME)`, it will throw away all the .o files in the source directories. You can even copy out the Makefiles, `lib`, and `include` to someplace else on your system: no paths are compiled in. Just remember to set `$(SWARMHOME)` to the right directory in your application's makefile. If you need to recompile the library and want to start from a completely clean source directory, `make realclean` will delete the includes and libs created or copied during the compile process. Note that `make realclean` will delete *all* the files in the include and lib directories. So, if you've set your Makefile.conf to point to a directory where you have other include files and libraries, then you don't want to do a `make realclean`.

For debugging purposes it's easiest if your Swarm libraries are compiled with `OPTFLAGS=-g` and you leave the sources around. However, your applications will run faster if you compile with optimization, `OPTFLAGS=-O3 -g`. This will make debugging inside the Swarm libraries strange, but gives noticeable performance improvement. Users of Sparc based machines should also see the notes about `-mv8` in `Makefile.conf`.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Random number library

simtools
(src/simtools.html)
: Collected bits of code necessary for running Swarm

tkobjc
(src/tkobjc.html)
: Graphical Interface library

analysis
(src/analysis.html)
: Various objects that help with data processing

Applications

In addition, there is a set of sample Swarm applications that are distributed separately. Applications are the best current roadmap for the Swarm code: much of what is possible with Swarm is demonstrated in the sample applications. The most important applications are:

template
: Template simulation. The code itself here is trivial, but provides a nice base for new Swarm programmers to start from.

heatbugs
: Heatbugs, a simple complex system. The code here is thoroughly commented for use as a guide to swarm programming.

mousetrap
: Mousetrap is a discrete event simulation of a room full of mousetraps loaded with ping-pong balls. The triggering of one of the traps sets off a chain reaction supposedly similar to fission. It is also thoroughly commented.

market
: Market is an Objective-C wrapped piece of legacy code originally written in C. It is a useful example of how to convert legacy code into Swarm; but, it is not very instructive on Swarm programming in general.

Credits

The Swarm developers are Roger Burkhart <roger@ci.deere.com>, Glen Ropella <gepr@santafe.edu>, Marcus Daniels <mgd@santafe.edu>, and Alex Lancaster <alex@santafe.edu> under the direction of Chris Langton <cgl@santafe.edu>. But credit for the package is spread out over many people. The Swarm developers can be reached collectively as <swarm@santafe.edu>. More information is available on our Web page, http://www.santafe.edu/projects/swarm/

(http://www.santafe.edu/projects/swarm/)
.

Swarm Todo/Wish list

Swarm is an open ended system which is meant to grow in response to the requirements of the user base, either by inhouse development or through user re-contributions. We are therefore maintaining a list of the most popular requests (both in terms of tools and libraries) so that groups of users can recognize common requirements, make more informed suggestions and so forth:

A 'Double' Space

Which could deal with notions of "distance" and answer questions of the form: "which other objects are within X radius of me"? An initial implementation of such a space has been re-contributed by Ginger Booth (http://peaplant.biology.yale.edu:8001/~ginger/) and may serve as a good foundation for such a space.

Complete Batch-Mode Support

Swarm can now run in batch mode, which should allow the user to organize large parameter sweeps over the models s/he is implementing. However, we still need to provide adequate and standardized support for file operations (it should be easy, for example, to load the parameters of an experiment from a file). In order to do this we will provide File objects which will allow users to avoid ad-hoc coding of their file-I/O. This support will be in place well before V1.0.

More Analysis Tools

The averager object can generate the Mean, Max, Min, and Count of a given input stream. We would like to add similar tools to calculate entropies, mutual information and other such measures.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Note that you can't pass arguments to a message, nor can you see the return value. There are also methods xfprint(collection) and xfexec(collection, "message") that print or exec foreach member of a collection.

Preventing Bugs

Defensive programming can help prevent a good number of bugs. When writing code, try to test it incrementally: make small changes whose effects you think you can predict and then test them. Don't outsmart yourself with cleverness: write code correctly first, then go in and hack it up if you need it to be more efficient. Put in sanity checks for conditions that shouldn't go wrong in normal usage, but might if you make a mistake.

-Wall

Swarm currently compiles all code with "gcc -Wall", which tells gcc to emit warnings for a lot of things that it wouldn't normally complain about. Warnings are not (necessarily) errors - warnings will be generated for legal code if gcc thinks that what you're doing could easily be a mistake. You might find this frustrating at first, but it helps catch a lot of common errors, including forgetting to include a prototype or forgetting to return a value from a function. Passing -Wall is good discipline.

nil˙method

Objects in Objective C are essentially pointers to structs. So what happens if you send an object to the pointer 0x0, "nil" in Objective C parlance? Unfortunately, most implementations of Objective C, including gcc, define methods to nil as having no effect. The code

```
aHeatbug = [Heatbug create: aZone];
 aHeatbug = 0; // oops! bug.
 [aHeatbug setIdealTemperature: idealTemp];
```

will not generate any errors.

The reason this is unfortunate is that it's a common bug to trash a pointer accidentally, set it to 0. It would be nice if your program then crashed when you tried to send a message to that mangled object: instead, the message send will fail silently and the program will continue to execute. This can make it hard to find bugs.

There are two ways to make messages sent to nil crash your program. The simplest is to put a breakpoint on `nil_method` under gdb: nil_method is invoked every time a message is sent to nil. Alternately, you can make a copy of the libobjc runtime source and edit nil_method to do whatever you want. The source code in `src/defobj` contains a file `objc.patch` that patches the runtime from gcc 2.7.2.

Emacs and Objective C

Emacs is the world's best editor, infinitely configurable and powerful and free. If you're not used to a Unix text editor and want to write programs, emacs is probably best to learn. The information here will show you how to set up emacs to use an Objective C specific mode for editing source code and how to get emacs to colour your source code for you nicely. Emacs can also help you find compilation errors, run a debugger, and even act like a class browser. Docs for those things are not here, see the Emacs documentation itself.

objc-mode

One of the best things about emacs is that one can load different "major modes" to edit types of files. Major modes making editing files easier by providing structure to your editing. In particular, there are programming language modes that do nice things like indentation for you.

> Starting with emacs-19.30, the default C mode is cc-mode.el, a nice rewrite of the original c-mode.el. Luckily for us, cc-mode.el supports Objective C directly. The basic function to invoke it is `objc-mode`. The following bit of code in your .emacs will cause emacs to automatically enter objc-mode on all files that end in `.m` or .h:

```
(setq auto-mode-alist
      (append '(("\\.h$" . objc-mode)
                ("\\.m$" . objc-mode))))
```

hilit19

Emacs running under X has the ability to colour your source code according to syntax. There are two packages to do this: font-lock and hilit19. I'm told font-lock is better, but I still use hilit19. Once you have hilit19 running in your emacs, you need to teach it how to highlight Objective C code. Here's what I use (cut and paste)

```
;; modified objective c - get defuns, id.
  (let ((comments     '(("/\\*" "\\*/" comment)))
(c++-comments '(("//.*$" nil comment)
("^/.*$" nil comment)))
(strings      '((hilit-string-find ?' string)))
(preprocessor '(("^#[ \t]*\\(undef\\|define\\).*$" "[^\\]$" define)
("^#.*$" nil include))))

    (hilit-set-mode-patterns
     '(objc-mode objective-C-mode)
     (append
      comments c++-comments strings preprocessor
```

Defobj Library

Documentation and Implementation Status

Overview

The defobj library supports the style of object-oriented programming that is used throughout Swarm. It defines a specific style for using the Objective C language that includes its own standard conventions for creating objects and for storage allocation, error handling, and debugging support.

Documentation

The defobj library follows the Swarm library interface conventions

(defobj/library.html)
. It has following standard sections of documentation:

- Usage Guide
(defobj/guide.html)
- Advanced Usage Guide
(defobj/adv.html)

- Interface Reference
(defobj/ref.html)
- Subclassing Reference
(defobj/subclass.html)

- Interface Design Notes
(defobj/design.html)
- Implementation Notes
(defobj/impl.html)

Marcus G. Daniels <mgd@santafe.edu>

Defobj Library: Documentation and Implementation Status

The current interfaces defined in defobj are expected to remain stable, with the exception of those relating to package structure and custom-generated classes. Those facilities are in process of being totally replaced, and will not be documented further until that replacement is complete. Details of support for defining a customized type will also be provided later.

All the interfaces defined in the header file are currently implemented, except for the various types of zones that would hold all allocated storage in a collection of local pages, and that would provide automatic reclaim of unused storage. The current zone implementation supports all defined allocation messages, and also maintains the population of the zone that consists of all objects created directly within it.

The Usage Guide section is only a start, but it does include subsections that discuss the style of Objective C programming adopted for use throughout Swarm. There is a full set of Interface Reference sections, though some details remain to be filled in.

Throughout the documentation, a parenthesized comment that starts with (.. indicates an editorial comment on the current status of implementation or documentation.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

The separation of implementation means that a library publishes its interface entirely without reference to any Objective C classes which implement its objects. Even though classes are often thought of as separating the interface of an object from its implementation, this separation is far from complete. Not only do classes typically contain many internal methods not intended for external use, but they also define a particular storage format for an object defined as instance variables.

A library instead publishes its interface as a set of public object types. These object types may also be supplemented by global object constants called symbols. Both these kinds of definitions normally appear only in the header file of a library. The remaining source files in a library normally contain the classes which implement the object types.

In a library, the files which implement classes (including class header files), need not ever be referenced simply to make use of the implemented capabilities of the library object types. Documentation for the library is normally expressed entirely in terms of the types and symbols published in the library header file. If a feature does not appear in the library header file, it should not be considered part of a supported public interface.

Individual class header files are required to subclass from existing implementations, but interfaces for subclassing are an entirely separate issue from normal public use of an object library. Class inheritance can be a powerful implementation technique, but extension of an existing class framework is typically safe only if performed in explicitly permitted ways. If a library supports subclassing at all, it must carefully state which classes may be subclassed and in what ways. For a library package, this information is supplied outside the library header file. The library header file specifies only the interfaces by which objects are intended to be used, whether implemented by a local class or an external subclass.

Remaining sections of this document explain the declarations which appear in a library header file, and end with a suggested structure of documentation to be provided for a library. The libraries of Swarm mostly follow this structure.

Object Type Definitions

The interfaces to objects defined by a library are specified by object type definitions. An object type defines only a set of messages which may be sent to an object over various phases of its lifetime. An object type makes no commitment to the classes that might be used to implement the object.

Multiple classes may all implement the same messages belonging to a type. The independence of types and classes means that different classes can provide alternate implementations of the same object type. For example, a particular implementing class might be selected to optimize the implementation for a particular case.

Object types are similar to protocols defined by Objective C, and the declarations appearing in a library header file are a minor adaptation of Objective C protocol

```
id <Symbol> North, East, South, West;

#import "grid.xt"
```

An object type is defined by an @deftype declaration. (Note: newer libraries, including objectbase and random, now follow the library interface conventions without using this special @deftype tag. Instead they use just an ordinary @protocol declaration, but otherwise they follow all the structure explained in this document.) The syntax of such declaration is identical to that of an Objective C @protocol definition, except for the entirely uppercase keywords (CREATABLE, CREATING, SETTING, USING) appearing in the GridTurtle example above. All these modifications of Objective C syntax are accomplished by simple preprocessor macros; no extensions to the language compiler are involved.

When this library header file is processed (by a special rule in a make file), an external object id with the name GridTurtle is automatically published. The name of a defined type becomes an ordinary object that accepts specific messages defined by the defobj library. The defobj library
(../defobj.html)
explains the details of such messages; the only purpose here is to explain the basic sections of a deftype declaration.

deftype declarations follow the syntax as Objective C protocols for inheriting messages from each other: a list of names enclosed in angle brackets (e.g., <Create, Drop, ...> above) gives the names of other declared types containing messages to be supported by the new type as well. (These types referenced here are defined by the imported file <defobj.h>.) Like protocols, full multiple inheritance of types is supported. The same messages may be inherited any number of times through any path with no different effect than if inherited or declared just once, so long as no conflicts occur in any of their argument or return types.

The CREATABLE tag appearing in the inherited type list above is a special type which defines no messages of its own, but merely marks the type as one which supports direct creation of instances of the type. Without this tag, the only role of a type is to define messages for inheritance by other types. With this tag, the global type object has a particular implementation that supports object creation using standard messages defined in defobj.

The declared messages of the type may be separated into sections marked by the special uppercase tags such as CREATING, SETTING, and USING above. (Currently, these are the only such tags which may occur.) These sections each define messages belonging to a particular defined "interface" of the object type, which are further combined into distinct "phases" of an object lifecycle supported by defobj messages. Further explanation of the interfaces and phases defined by this example are provided in the Usage Guide
(guide.html)
of the defobj library.

The following line at the end of the <grid.h> header file above is required as part of the standard conventions for coding a library header file: This line adds no information to the header file, but is required as part of the system by which definitions are is no other specific return value is generally *not* followed. If there is no specific return value needed from a message the return type is declared (void).

Documentation Structure

The following standard sections of documentation are suggested for libraries that follow the strict interface vs. implementation separation of library packages, each with purpose and typical contents as given:

- Usage Guide
  Tutorial introduction to a library. Focuses on the most common uses in an order reflecting the needs of a first-time user. Most explanation by means of progressively elaborated examples. Serves as a guided tour of major library capabilities. No attempt at reference-style completeness.
- Advanced Usage Guide
  Continues overview of all significant capabilities of a library, including those which might be needed by advanced users customizing or extending built-in capability. Provides examples of specialized uses extending beyond normal, basic usage, but still using built-in features of the library framework. Helps to simplify the Usage Guide by providing a location for overflow of more advanced features.
- Interface Reference
  Complete, concise summary of all features of a library. Fills the role a Unix "man page" in providing comprehensive definition of library services. Little or no concern to provide path of tutorial introduction. Does not contain extended examples, at most only fragments of examples which serve needs of specification.
- Subclassing Reference
  Documents the rules for writing new classes which subclass from classes that implement the library. Because a library might use complex combinations of classes to implement the range of behaviors defined by its types, the classes in a library are not automatically usable as superclasses of user-defined classes. Each library documents which classes are available for use as superclasses, and the specific rules that must be followed when subclassing from these classes.
- Interface Design Notes
  Explains why choices were made the way they were in the design of the public interface to a library, including other alternatives considered. Serves as a "Rationale" document for the documented interface. May include historical background deliberately left out of the preceding sections. May also include references to other work.
- Implementation Notes
  Explains the structure of classes by which the types of a library are implemented. Summarizes the status of implementation if still incomplete and lists items of possible future work. Provides overview and high-level structure of the implementation in whatever ways would best guide a reader of implementation source code. May

Defobj Library: Usage Guide

Contents

Why Swarm uses Objective C

Swarm uses object-oriented programming not only because this is a good way to build general-purpose software libraries, but because the very concept of an object is at the base of how you build a model in Swarm. To build a simulation in Swarm, the first step to define the various kinds of objects that can inhabit some real or artificial world, and the second step is to define the kinds of events that can occur to these objects, including all the different ways that the objects can interact with each other.

> The basic concept of an object is that it responds to external events, and that the only thing that really matters about an object are the ways it can be observed responding to these events. In an object-oriented programming system, an object is represented by some uniquely identified chunk of data, and the events are the dynamic operations that can be made to occur on these objects.
>
> Different object-oriented programming systems define the operations that occur on objects in different ways. For example, in C++ the operations are called "member functions" and look much like an ordinary function call in the C language. In Objective C, an operation on an object is called a "message" and has a special syntax by which you call it, but a message also has arguments and behaves in many respects like a call to a function.

of the defobj library. But many others are at the heart of how Swarm uses Objective C, and so are dealt with in the rest of this Usage Guide. The entire purpose of the defobj library is to define a standard style for the use of Objective C in Swarm. This style is backed up by a library of foundation classes that Swarm provides to support this style.

Swarm provides its own foundation classes even for such basic operations as creating an object, and other support that user classes ordinarily receive from a builtin Object superclass. A good understanding of the defobj library is essential for Objective C programming in Swarm. Like any programming language, Objective C requires learning not only the rules of the language itself, but also a standard library of initial capabilities that you build from. Objective C doesn't really have a single official standard library (or language definition either, for that matter), but the NextStep foundation libraries (and the NextStep language definition) come very close to this status.

For a variety of reasons explained here and in the Advanced Usage Guide
(adv.html)
, Swarm doesn't use a standard library based on the NextStep foundation interface. You can still learn the Objective C language from other available sources, but you have to be careful to sort out the language level from the standard object and library support they also discuss.

The index page
(../../index.html)
of the Swarm documentation provides a variety of links to other Objective C resources. In particular, it provides a link to a complete on-line reference for the Objective C language as defined by NextStep and implemented by the GNU C compiler. None of the Swarm documentation attempts to duplicate this coverage of the basic language; Swarm assumes that you learn Objective C from other available sources. The NextStep reference(*Object-Oriented Programming and the Objective C Language*
(http://www.next.com/Pubs/Documents/OPENSTEP/ObjectiveC/objctoc.htm)
) is the single best available source, and it can also be ordered as a hardcopy book from the NeXT publications page
(//www.next.com/Pubs/Documents/OPENSTEP)
. Don't try to program in Objective C without it.

Swarm style of Objective C Programming

No matter where you've learned to program in Objective C, don't try to use Objective C in Swarm without understanding the special ways in which Swarm adapts its use of the language and the language runtime system. Your place to find this information is right here in the defobj library. Defobj not only provides the most basic layer of foundation class libraries for Swarm, but also serves as the place where all the rules and guidelines for the use of Objective C in Swarm are gathered together.

Defobj Library: Advanced Usage Guide

(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

- module definition conventions
(module.html)
- create combination messages
(combo.html)
- global portability assumptions
(globalport.html)
- global definitions
(globaldef.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

The DefinedObject type defines a minimum of standard messages, and leaves to other types the definition of message that might or might not apply in any general way to particular objects.

Inherited behavior

The DefinedObject type inherits no behavior from any other type. It is the top of the interface type hierarchy for objects that follow the defobj programming conventions. All objects that follow these conventions should inherit from DefinedObject, either directly or indirectly through other types.

Defined behavior

```
- getZone;
- (BOOL) respondsTo: (SEL)aSel;
- getClass;
```

The getZone message returns the zone in which the object was created. The respondsTo: message returns true if the object implements the message identified by the selector argument. To implement a message means only that some method will receive control if the message is sent to the object. (The method could still raise an error.) The respondsTo: message is implemented by direct lookup in a method dispatch table, so is just as fast as a normal message send. It provides a quick way to test whether the type of an object includes a particular message. getClass returns the class that implements the current behavior of an object.

Defined behavior

```
- (ref_t) addRef: (notify_t)notifyFunction withArgument: (void *)arg;
- (void) removeRef: (ref_t)refVal;
```

These messages add or remove an external reference to an object that is notified whenever the object is changed or relocated. (More details to follow.)

```
- (int) compare: anObject;

- perform: (SEL)aSel;
- perform: (SEL)aSel with: anObject1;
- perform: (SEL)aSel with: anObject1 with: anObj2;
- perform: (SEL)aSel with: anObject1 with: anObj2 with: anObj3;
```

These messages are have implementations in the Object superclass, excppt for the three-argument form of perform:. They have local implementations in defobj, however, just to make subclassing

Name

Create – create a customized instance of a type

Synopsis

```
//
// Create -- create a customized instance of a type
//
@deftype Create <DefinedObject>
CREATING
+ create: aZone;
+ createBegin: aZone;
- createEnd;

+ customizeBegin: aZone;
- customizeEnd;
- customizeCopy: aZone;
@end
```

Description

The Create supertype defines standard messages that provide a general-purpose protocol for creating new objects. These messages may be used either to create a new instance of a type in one message, or to bracket a series of messages that customize available options for an object to be created. The separation of create-time specifications from later behavior of an object gives substantial flexibility to adapt a generic type to particular needs.

These create messages may be implemented either by a type that hides its implementing classes, or directly by a class that adopts these messages as a uniform interface for creating objects. If implemented directly by a class, then the class object serves as the type object in all message descriptions that follow. Otherwise, a type object might be implemented in a variety of ways that guarantee only that published messages on a type are accepted.

In addition to the create messages defined by Create, an object type may support any other messages of any other names or calling conventions. These messages define only a standard method for creating new objects that other types are free to inherit and implement in conformance with a uniform convention. Further conventions are established elsewhere (create combination messages
(combo.html)

createBegin: returns an interim object intended only for receiving create-time messages. If a type was defined by a @deftype declaration, these messages are those appearing in either the CREATING or SETTING sections. Otherwise, the messages valid as create-time messages are defined by the type without any specific syntactic marker.

The createEnd message completes the process of specifying available options for an object being created. Typically it validates that requested options are valid and consistent with one another, and raises an error if they are not. The standard, predefined error InvalidCombination
(Error.html#stderr)
may be raised by createEnd to indicate an invalid combination of requests, or other, more specific forms of error handling may be used.

If all requests received since the initial createBegin: are valid, both individually and in combination with each other, then createEnd determines a finalized form of object that satisfies all requests received and then returns this object. Any additional storage required for the finalized object is taken from the same zone originally passed to createBegin. The object may have whatever implementation is selected to best satisfy a particular request. Different requests may result in entirely different implementations being returned. The only guarantee is that a returned object supports the messages defined for further use of the finalized object. If a type was defined by a @deftype declaration, these messages are those appearing in either the SETTING or USING sections.

On return from createEnd, the id of the interim object returned by createBegin: is no longer guaranteed to be valid for further use, and should no longer be referenced. A variable which holds this such an id can be reassigned the new id returned by createEnd, so that the same variable holds successive versions of the object being created. Following is a fragment that illustrates typical coding practice using createBegin: and createEnd:

```
newArray = [Array createBegin: aZone];
  [newArray setInitialValue: aList];
  [newArray setDefaultMember: UnsetMember];
  [newArray setCount: [aList getCount] * 2 );
  newArray = [newArray createEnd];   // ! note reassignment of newArray
```

By resetting the variable to the returned value of createEnd, there is no possibility that the id value first returned by createBegin: can ever be used again. Sometimes, the id value returned by createEnd might be the same value first returned by createBegin:, if createBegin: was able to allocate the same object returned later by createEnd. In other cases, however, the object returned by createBegin: might be strictly a temporary object that is thrown away after createEnd determines a final object to be created and returned. To protect against changes in implementation, only the final createEnd return value should ever be used again after createEnd has returned.

```
+ customizeBegin: aZone;
- customizeEnd;
- customizeCopy: aZone;
```

Some types accept create-time messages not only when creating a new instance, but to customize a new version of the type itself. Objects created from a customized type will have all options preset that create-time messages sent to the customized type object have already set. If many cycle of customization (by another sequence of customizeBegin:/customizeEnd) depends on the implementation of the original starting type. A type should not be relied on to support more than one cycle of customization unless it is specifically documented to do so.

Any interim object returned by either createBegin: or customizeBegin: supports the getZone and drop messages that a finalized instance may also support. These messages are defined by the Drop
(Drop.html)
type, which is normally inherited by a type to declare these messages on a finalized instance. This type is not inherited by the Create type because the messages would then apply to the finalized instance, not to the interim object. Even though not declared, the messages are available on the interim objects nonetheless. The drop message on an interim object may be used if it turns out that a finalized version is no longer required after creation or customization has already begun.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

of an object lifecycle.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

```
- (void) xfprint;
- (void) xfprintid;
@end
```

Description

A zone is a source of storage for objects or other allocated data. Whenever a new object is created, a zone must be identified from which the storage for its instance variables, or other internal data, is obtained. A program may establish multiple zones to ensure that objects with similar lifetime or storage needs are allocated together, and in general to optimize allocation and reuse of storage.

Zones also maintain a collection of all objects allocated within the zone. This collection, referred to as the "population" of a zone, is a set of all objects which have been created but not yet dropped within the zone. Collections maintained automatically by zones can eliminate a need for other, separately maintained collections in applications that need to keep track of entire populations of objects. Collections of allocated objects can provide support for object query, external object storage, and automatic storage reclamation.

A zone may be used to obtain storage not only for objects, but also for raw storage blocks like those provided by the C malloc function. All objects and storage blocks allocated in a zone remain local to that zone. This means that allocation of storage in other zones does not affect the efficiency of storage allocation within a particular zone. For most zone types, individual allocations may still be freed within a zone, and total storage of a zone may grow and shrink according to aggregate needs. In addition to freeing individual allocations, an entire zone may also dropped. Dropping a zone automatically frees all allocations made within it, including final drop processing on any allocated objects that need it. Release of an entire zone can be much faster than individual release of each object within it.

The Zone type is a fully implemented type that provides default storage management support for objects and other allocated storage. It is also a supertype for other zones that implement alternative policies for use in specialized situations.

(..The current Zone implementation is a temporary one based on a simple pass-through of all allocation requests to the C library malloc function. Complete support of the zone population is already provided, however, as is support for internal storage mapping that will eventually be required for automatic storage reclamation Considerable improvements in storage allocation are possible when the current implementation is replaced by more efficient allocation contained in pages local to each zone. ..)

Inherited behavior

```
- allocIVars: aClass;
- copyIVars: anObject;
- (void) freeIVars: anObject;
```

allocIVars: allocates the instance variable structure for a new object. The initial word of this structure is set to class id passed as its argument. The class also determines the size of the structure allocated. All remaining contents of this structure are initialized to binary zeroes.

copyIVars: creates copies an existing instance variable structure into a new allocation made within the local zone. The existing instance variable structure may be in any zone, but must contain a class pointer in its first word that correctly describes the size of the structure.

freeIVars: releases storage that was previously allocated to hold the instance variable structure of an object. The first word of the object must be a class pointer that correctly describes the size of the structure. Storage allocated by allocIVars: or copyIVars: may be freed only by freeIVars:, and freeIVars: may be used only to free storage allocated by one of these messages.

The object allocation functions are intended for use only within the implementation of an object class. Users of an object type should create a new object only using create: messages or other messages provided for this purpose.

Internal component objects

```
- allocIVarsComponent: aClass;
- copyIVarsComponent: anObject;
- (void) freeIVarsComponent: anObject;

- getComponentZone;
```

These messages allocate, copy, and free the storage that holds the instance variables for an object, but they allocate the object as an internal component of the zone that is not included in the zone population. They are used by classes that allocate additional objects as part of the implementation of another object, and that control the mapping of this storage separately from the zone level objects. getComponentZone returns a specially qualified version of the zone that automatically allocates all its objects with the internal component qualification, even if allocated with allocIVars: or copyIVars:. This qualified zone may be passed as an argument to a create: or createBegin: message so that it will create the new object as an internal component object.

Raw storage block allocation

```
- (void *) alloc: (size_t)size;
- (void) free: (void *)aBlock;

- (void *) allocBlock: (size_t)size;
- (void *) copyBlock: (void *)aBlock blockSize: (size_t)size;
```

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Create combination messages

```
CREATING
+ create: aZone setName: (const char *)name;
```

create:setName: is a combination message defined as a caller convenience. See combination messages
(combo.html)
for a summary of conventions on combination messages.

Create-time options

```
CREATING
- (void) setName: (const char *)name;
```

The Name option may be set only at create time. Its value is a null-terminated character string that remains fixed for the life of the object. The inherited getName message returns this name from a created instance.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

(module.html)
for standard methods of declaring interface symbols in a library header file.)

These predefined warning and event types are typically used not directly by the messages defined above, but by the following macro defined in defobj.h:

```
#define raiseEvent( eventType, formatString, args... )
```

This macro raises an event of the type identified by the first argument. Default handling of the event is to print a formatted message including standard header and trailer lines, and then either to continue (in the case of a Warning) or to terminate the program in the case of Error). The standard header lines include the file, function, and line number where the macro that raised the event was executed.

If the formatString argument is nil, a default message string already established as part of the warning or event type is used to report the error. Otherwise, the formatString argument must be a null-terminated C character string that overrides the default string.

Either the default or replacement message string may contain printf-style formatted field definitions. For each formatted field definition, an argument value must be supplied in the args... section of the macro. (The support of the GNU C compiler for macros with variable argument lists is used by the macro.)

(.. Methods for defining new event types as part of a library implementation are not currently documented. Instead, just use the raiseEvent macro on one of the predefined Warning or Event Types if you want to raise an event using this standard mechanism. Eventually, this standard mechanism will be used to handle warnings and errors under various forms of logging exception handling support.)

Standard Error and Warning Types

The following standard error and warning types are defined by the defobj library. Thesy may also be used by other libraries for errors or warnings that fit within these standard predefined categories:

```
//
// standard errors
//
extern id <Error>
  SourceMessage,          // message in the source defines error
  NotImplemented,         // requested behavior not implemented by object
  SubclassMustImplement,  // requested behavior must be imple-
mented by subclass
  InvalidCombination,     // invalid combination of set messages for create
  InvalidOperation,       // invalid operation for current state of receiver
  InvalidArgument,        // argument value not valid
  CreateSubclassing,      // improper use of Create subclassing framework
  CreateUsage,            // incorrect sequence of Create protocol messages
```

Miscellaneous Operations

Object Types

The following object types are documented below:

- Copy
- GetName
- GetOwner
- SetInitialValue

Summary of Miscellaneous Operations

An assortment of object types in the defobj library serve just to establish a single declaration of messages that many other object types adopt as part of their declared interface. The centralized definition of the messages in these types avoids their repetition throughout many other types. It also provides a single place where basic rules for these messages can be summarized for all other implementations to follow. This reference documentation provides these summaries along with intended purpose of each message.

> None of these types inherits from any other types; their whole purpose is just to package the definition of one or more messages. The type name is typically the same as a message declared by the type, only with the uppercase convention of a type name. These types may be freely mixed into the inheritance lists of other types to build whatever combinations of messages those other types need to define.

Copy Interface Description

Synopsis

```
//
// Copy -- copy all state defined as part of object
//
@deftype Copy
- copy: aZone;
```

Synopsis

```
//
// GetOwner - get object on which existence of object depends
//
@deftype GetOwner
- getOwner;
@end
```

Description

The getOwner message returns another object which is considered as the owner of an initial object. What is considered as an owner depends on its specific object type, but might be a larger object of which the local object is a part, or an object that has exclusive control over the local object. The principal constraint established by an ownership structure is that a given object can have only a single other object as its unambiguous owner.

> Ownership hierarchies arrange themselves in a strict, single-rooted tree. The top-level node of an ownership hierarchy typically returns nil as its owner. If an object is regarded merely as one part of another object defined as its owner, then copying or dropping the owner object should copy or drop the member object as well. Owner and member are neutral terms for a generic relationship sometimes called parent vs. child, but it is up to a particular object type to define specifically what it means by a getOwner relationship.

SetInitialValue Interface Description

Synopsis

```
//
// SetInitialValue - create using initial value from an existing object
//
@deftype SetInitialValue
CREATING
+ create: aZone setInitialValue: initialValue;
+ create: aZone setReadOnlyValue: readOnlyValue;

- (void) setInitialValue: initialValue;
- (void) setReadOnlyValue: readOnlyValue;
SETTING
- (void) setReadOnly: (BOOL)readOnly;
USING
- (BOOL) getReadOnly;
@end
```

Module Definition Conventions

Overview

Some of the basic Swarm libraries are implemented using object definition conventions established by the defobj library

(../defobj.html)
. These conventions include some minor special syntax to declare the public interfaces to Objective C objects. This document explains how to read and interpret the header files and interface definitions published by such libraries. It also explains the typical structure of documentation provided for libraries that follow these conventions.

These conventions are documented under the following subsections:

- library header file
- object type definitions
- GridTurtle example
- global object symbols
- interface design conventions
- documentation structure

Library Header File

For a library that adopts these conventions, a library is not merely a collection of source files that are compiled into a library archive under control of a make file. Instead, Objective C source files in the library are processed in a special way to publish their definitions not only as header files, but as generated objects available at runtime to make full use of the library. A library processed in this way is referred to as as a "module."

The defobj
(../defobj.html)
library documents the full details of special processing performed on a module. For simply using a library defined as a module, the key fact to keep in mind is that the entire public interface to a module is declared in the one header file having the same name as the module itself, plus a trailing .h suffix. Additionally, the header file of a library module normally documents only the public interface to a module, in a way that is completely separated from the implementation of the objects it specifies.

The separation of implementation means that a library module publishes its interface entirely without reference to any Objective C classes which implement its objects.

difference is that the object types of the defobj library are divided into separate interfaces that define distinct phases of an object's life cycle.

GridTurtle example

An example will help illustrate the features of type definitions supported by the defobj library. Throughout documentation of the basic Swarm libraries, a series of running examples will be based on a simple type of object belonging to sample Swarm simulations. This type of object is an agent that move around on a two-dimensional grid, always moving in a current direction that it maintains internally. This agent is like a "turtle" of the original Logo system, except that its position is constrained to discrete integer values of its X-Y coordinates, and its direction is always one of the four orthogonal directions north, east, south, or west.

Following is a complete library header file for a module which defines such an object, called GridTurtle:

```
/*
Name:         GridTurtle.h
Description:  object type for Swarm example programs
Library:      grid
*/

#import <defobj.h>
@deftype GridTurtle <Create, Drop, CREATABLE>CREATING
- (void) setXLocMax: (int)xLocMax;
- (void) setYLocMax: (int)yLocMax;
SETTING
- (void) setXLoc: (int)xLoc;
- (void) setYLoc: (int)yLoc;
- (void) setXLoc: xLoc setYLoc: yLoc;

- (void) setDirection: direction;
USING
- (int) getXLoc;
- (int) getYLoc;

- getDirection;

- (void) move: (int)distance;
- (void) turn: (int)angle;  // angle measured in units of pi/2 (90 deg.)
- (void) print;
@end

id <Symbol> North, East, South, West;
```

Global Object Symbols

The <grid.h> header file above also contains the declaration:

```
id <Symbol>  North, East, South, West;
```

Lines that declare global id variables of type Symbol, EventType, Warning, or Error (using the angle bracket syntax of id variables conforming to a protocol) are processed somewhat like deftype declarations in that they also produce global id variables initialized to support particular messages defined by defobj. These global variables, however, are not used to define or implement other message interfaces, but only to define certain fixed capabilities referenced through their global object names.

If declared as a Symbol, as in the case here, the generated objects have no particular behavior of their own (other than the character string of their name), but only serve to define unique global id constants which may be used as distinct named values in messages. In this example, the current direction of a GridTurtle object is represented by one of the symbol names North, East, South, or West. These values are like the enum constants of the C language, except that they are defined as full Objective C objects, and may be used with further restrictions. An EventType, or a Warning or Error, defines a further subtype of a Symbol constant with further specialized messages documented in defobj.

Interface Design Conventions

A variety of rules on naming and declaration of object types, symbols, and messages are followed by many of the Swarm libraries. These rules help establish a basic consistency on the library interfaces. Following is a list of such conventions that apply to a public library interface, some but not all of which are derived from standard Smalltalk or Objective C coding practice:

- Names of global object constants are capitalized. In the public interface, such names include types and symbols.
- Recapitalization separates words of a compound name (e.g., GridTurtle). Underscores are generally not used.
- Message names start with a lower-case character, and are named using verbs. Nouns that represent gettable or settable components of object state (e.g., Direction of a GridTurtle), are prefixed by get or set to indicate the action being performed.
- The Smalltalk convention that a message return the receiver of a message if there is no other specific return value is generally *not* followed. If there is no specific return value needed from a message the return type is declared (void).

(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Global Portability Assumptions

In a pure object-oriented system such as Smalltalk, the type of an object reference (an object id in Objective C) is the only fundamental data type used within the system. In a hybrid object system such as Objective C, both object id's and other data types are used together. The Swarm libraries make some basic assumptions about the compatibility of native C data types with the id pointer type of Objective C. These assumptions allow some of the more generic abstractions of the libraries (such as collections and actions) to provide one fundamental abstraction that can still be used under an identical implementation to support most of the other C data types.

The basic assumption is that all data types except double may be cast to and from a stored id type without losing the original value. This assumption is valid on all modern 32-bit or greater machines, including all machines on which Swarm is currently running or planning to run.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

func_t provides a type definition for a generic function pointer type (taking no arguments and returning no value). This type name is used in declaring messages that accept or return function pointers of various alternative types.

---

initModule() function

Synopsis

```
//
// initModule() -- module initialization macro
//
#define initModule( module )
```

The initModule function must be called to initialize all the definitions of a library module before any attempt is made to use the library. Initializing a particular library module will also initialize all other library modules on which the library depends. The macro has no effect if the library has already been initialized, so that multiple libraries that all depend on another library can all request initialization of the other library with no ill effect. The argument of the macro is the name of module to be initialized. For example, if just the collections library were being initialized, the macro would be coded:

Global Zones

Synopsis

```
#define globalZone
#define scratchZone
```

The identifiers globalZone and scratchZone refer to two standard zones created by initialization of the defobj library. All other created zones must obtain their storage from one or the other of these two initial zones. (These initial zones are defined as macros only to follow conventions on global external names.)

> globalZone may be used for all storage which must be allocated for the duration of the program, or until explicitly released. scratchZone should be used for storage which is allocated only for use during execution of a single function. It is typically used for objects which are strictly temporary as part of a local computation. (.. Currently, a scratch zone has the identical implementation as the global zone, but eventually a scratch zone may be optimized for temporary allocations, and such allocations may also be freed automatically at various points when functions are guaranteed to be complete.)

The xprint function prints a standard object description to the standard debugging output file. It generates the string to be printed using the describe: message defined by the DefinedObject supertype, but it is defined as a function so that it may be called directly from a debugger. xprintid is similar, but it prints only a one-line description using the built-in default.

The xfprint function is similar to the xprint function, except that it prints a debug description for each member of a collection. The argument of xfprint must support enumeration of collection members using the messages of the Collection
(../collections/Collection.html)
type. xfprintid is similar, but only one line per member is printed.

The xexec function executes a message on an object passed as an argument. The name of message to be executed (the full name including arguments as would be used in naming a selector) must be given as the second argument. This function is equivalent to the "perform:" message of Objective C, but is defined as a function so that it may be called directly from a debugger.

The xfexec function is similar to the xexec function, except that it executes a message on each member of a collection. The first argument of xfprint must support enumeration of collection members using the messages of the Collection
(../collections/Collection.html)
type.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Defobj Library: Interface Design Notes

(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Collections Library

Documentation and Implementation Status

Overview

The object types of the collections library establish a general-purpose foundation to maintain object references or other values as members of structured collections. Customization options consolidate a wide range of basic collection structures into a few core types (Array, List, Set, Map). These types are defined strictly by their interface, not their internal implementation. Specialized options, however, give control and flexibility for efficient, low-level use (such as implementing other libraries).

Documentation

The collections library follows the library definition conventions

(defobj/library.html)
. It has following standard sections of documentation:

- Usage Guide (collections/guide.html)
- Advanced Usage Guide (collections/adv.html)

- Interface Reference (collections/ref.html)
- Subclassing Reference (collections/subclass.html)

- Interface Design Notes (collections/design.html)
- Implementation Notes (collections/impl.html)

Marcus G. Daniels <mgd@santafe.edu>

Collections Library: Documentation and Implementation Status

The collections library has an almost complete set of Interface Reference documents, and the design of the interface is almost entirely final. The implementation of the collections library, however, doesn't yet implement many of the more advanced features of this interface.

Collections library development has concentrated on the specific features needed by the activity library. This library uses collections for its underlying support of actions to be executed in schedules. The collections library itself is still being completed for use as a general-purpose library.

The Array and List types have all basic capability fully implemented. Set and Map have basic messages defined, but currently rely on a crude implementation based on sorted lists. (This implementation is more than adequate for the usually small and relatively static schedule structures in the activity library, and is the most forgiving when change does take place within members being traversed.) Much more efficient implementations based on both balanced trees and hash tables are in the works.

OrderedSet is currently supported only with the low-level option for an internal member slot, and the implemented messages do not match the ones documented in the Interface Reference. Support for groups of duplicate members is missing from both Set and Map.

None of the special options for restricted usage modes on any type of collection (e.g., read-only restriction) has been implemented. Stack and Queue are not implemented, but are nothing more than restricted uses of a List. There is no support for any member type except id or smaller (other member types will depend on a data type facility to be supplied by defobj).

Even though incomplete, the portion of capability that is implemented has been exercised very heavily. The interfaces to Set and Map structures will remain the same even as their underlying implementations improve, so there is no harm in using them. See the GridTurtle test programs
(../grid.html)
for the most complete examples of collections usage.

The collections library follows the documentation structure suggested by the library interface conventions
(../defobj/library.html)
of the defobj library. There are placeholders at least for each section of documentation (some of which merely indicate that the section is not available yet) so that all links should at least link up with something, whether or not there's anything there.

Throughout the documentation, a parenthesized comment that starts with (.. indicates an editorial comment on the current status of implementation or documentation.

The documentation priority for all libraries is to complete at least their interface reference documents, so that there is the equivalent of Unix "man pages" that summarize

Collections Library: Usage Guide

This section of documentation is not yet available. In the meantime, see the GridTurtle test programs
(../grid.html)
for the most complete examples of collections library usage. If you can't find an example there that exercises a message or option that you want to use, chances are it's not implemented.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Collections Library: Interface Reference

Dependencies

Following are the other header files imported by <collections.h>:

```
#import <defobj.h>
```

The collections library follows the library interface conventions
(../defobj/library.html)
of the defobj library
(../defobj.html)
. It also depends on standard supertypes and classes defined by this library. Initialization of the collections library automatically initializes the defobj library as well. Since defobj also requires the collections library, both must always be linked into an application together.

Object Types (with links to reference pages)

```
Collection
 (Collection.html)
          String
 (String.html)
      OutputStream
 (Stream.html)
   Index
 (Index.html)

     /       |          \
    /        |           \
 Array
 (Array.html)
     List,
 (List.html)
           KeyedCollection,
 (KeyedCltn.html)

             ListIndex
 (List.html)
       KeyedIndex
 (KeyedCltn.html)
```

- Map
(Map.html)
- String
(String.html)
- OutputStream
(Stream.html)

---

*Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)*

Last modified: 1998-01-08

Description

A collection is a grouping of object references or other data values which are assigned explicitly as members of the collection. Depending on the subtype, collection members may also be maintained in various associations with each other, such as an ordering of members or an association of members with identifying key values. Major Collection subtypes include Array, List, Set and Map. The Collection supertype establishes common conventions (and associated messages) supported by all types of collections.

> All collections support traversal over their members using a separate object called an Index. All collections also follow common rules regarding the types of data values which may be added as members. The next two subsections summarize these basic features of all collections.

Index objects

An index is a special type of object that references a current position in an enumeration sequence over a collection. An enumeration sequence contains every member of a collection exactly once. Every collection defines an associated type of object for its index. The index type of a collection defines additional ways in which the members of a collection may be processed beyond any messages available directly on the collection. Often the operations of an index provide the most basic and flexible means for manipulating individual members of a collection. An index object into a collection may be created at any time. An index is the basic means to traverse through all members of a collection. Multiple indexes on the same collection may all exist at the same time and may reference the same or different positions. Depending on the collection type, it may be possible to modify a collection through its indexes.

> Once an index is created, the sequence of members in its enumeration sequence is guaranteed to remain the same, provided that no new members are added to the underlying collection, or existing members removed. If a member is located once at a particular position, it is guaranteed to remain at that position as long as the index itself remains.
>
> Many collection types define an explicit ordering over their members. For such collections, the sequence of members referred to by an index will always be consistent with this ordering. An explicit, total ordering also guarantees that all indexes of the same collection have the same member sequence.
>
> If no such ordering is defined, however, some particular sequence of all the collection still becomes associated with each created index. All collection members are guaranteed to be contained somewhere in the enumeration sequence for any particular index, but two indexes on the same collection are not guaranteed to have the same sequence.
>
> The Index type corresponds to the iterator types defined as part of many other object-oriented libraries. The name Index is shorter and emphasizes the more abstract and multi-function role of these basic support objects for any collection. For more background on design of indexes and iterators, see the Interface Design Notes for the collections library.

being created. Members (and keys, too, if applicable) are added to the new collection in the same order as defined by traversing the initial value collection with a local temporary index. Create-time options are also applied as noted under SetInitialValue. Some collection subtypes may have further special rules for initial values; if so, these are documented with the subtype.

[aCollection copy: aZone] is equivalent to [[aCollection getType] create: aZone setInitialValue: aCollection].

Create-time options

```
CREATING
- (void) setMemberType: aDataType;
- (void) setIndexSafety: indexSafety;
- (void) setIndexUpdateHandler: (handler_t)handler : (void *)arg;
SETTING
- (void) setReplaceOnly: (BOOL)replaceOnly;
USING
- getMemberType;
- getIndexSafety;
- (handler_t) getIndexUpdateHandler: (void *)arg;
- (BOOL) getReplaceOnly;
```

MemberType

(.. This option is not currently implemented for any collection types.)

The MemberType option may be used to declare the type of member which a collection contains. Its value must be an objec having one of the ValueType types defined in defobj. (..Currently no ValueType objects are implemented, so MemberType is not supported.)

The member type defaults to the Symbol value Unknown. Specifying a member type has no effect on the collection if the storage size of the data type is no larger than a pointer of id type. The member type, however, may be helpful as documentation, and members can be checked for conformance to the type as part of debug options. If the data type has a size greater than id, the collection automatically allocates and frees space for the member value whenever member values are added or removed. For such a collection, all member values must be passed to the collection not as the value itself, but as a pointer to the value (cast to id as necessary). If a new member is being added, the value will be copied to a new member location contained within the collection. Member values returned by the collection will be returned as pointers to this internal value, not as the value itself. In the case of member values removed from the collection, a nil pointer is returned.

ReplaceOnly

(.. This option is not currently implemented for any collection types.)

This boolean-valued option restricts valid usage of a collection by excluding all operations which add or remove members. For some collection subtypes, a replace-only restriction can obtain many of the same performance advantages as a read-only collection, but without disabling replace operations as well. Just like the ReadOnly option, the ReplaceOnly option may be reset after a collection is created, provided it was not originally set to true.

Collection behavior

```
- (int) getCount;
```

getCount returns the integer number of members currently contained in the collection. All collections maintain their count internally so that no traversal of collection members is required simply to return this value.

```
- atOffset: (int)offset;
- atOffset: (int)offset put: anObject;

- getFirst;
- getLast;
```

An offset is an integer value that gives relative position of a member in the enumeration sequence of a collection. Offsets start the count of the first member at zero, just like C array indexing.

Offsets provide an alternate means to access the members of a collection, without creating a separate index object. Some collection subtypes (such as Array) support fast, direct access by integer member offset, while others support member offsets only as a shorthand for sequential access through every preceding member. Access by offsets is supported on all collections regardless of whether its speed on a particular collection type.

The atOffset: message returns the member at a particular member offset. The atOffset: put: message replaces the member at a particular offset with a new value, and returns the previous member value at this offset. Both these messages raise the error OffsetOutOfRange if the offset is less than zero or greater than or equal to the count of members in the collection.

The getFirst message is equivalent to [aCollection atOffset: 0], and the getLast message is equivalent to [aCollection atOffset: [aCollection getCount] - 1].

```
- (BOOL) contains: aMember;
- remove: aMember;
- (void) removeAll;
```

The contains: message returns true if the collection contains any member value which matches the value passed as its argument. Depending on the collection subtype, this may require traversing

Name

Index – reference into an enumeration sequence for a collection

Synopsis

```
//
// Index -- reference into an enumeration sequence for a collection
//
@deftype Index <DefinedObject, Copy, Drop>- getCollection;

- next;
- prev;
- findNext: anObject;
- findPrev: anObject;

- get;
- put: anObject;
- remove;

- getLoc;
- (void) setLoc: locSymbol;
- (int) getOffset;
- setOffset: (int)offset;

- (int) compare: anIndex;
@end
```

Description

An index is a reference into an enumeration sequence of a collection. Such an enumeration sequence contains all members of the collection in some order. This order will always be consistent with ordering of members in the collection, assuming there is such an ordering. Otherwise, the sequence will still contain all members in some order that remains fixed provided that new members are not added or removed from the collection.

An index is created by a begin: or createIndex: message against a collection. Each major collection type has its own corresponding index type, which supports specialized types of processing against the valid contents of that kind of collection. Once created, an index is a separate object from the collection itself, but it remains valid only so long as the collection itself still exists. Multiple indexes may exist at the same time against the same collection, and each index maintains its own position within an enumeration sequence for the collection.

The next message positions the index to the next valid member after its current position, or to a special position after the end of all valid members. The prev message works similarly, but positions to a valid member preceding the current position, or to a special position preceding all valid members. In addition to repositioning the index, both messages return the new member value to which they are positioned, or nil if there is no such member.

Following is a standard form of loop which may be used to enumerate the members of a collection using an index:

```
index = [aCollection begin: aZone];
  while ( (member = [index next]) ) {
    // do something with member ...
  }
```

When first created, the current position of an index is always set to the special position preceding all valid members. From this initial position, this loop accesses each succeeding member in turn. It assigns each member value to a local variable ("member" in this example) until a nil value is returned.

```
- findNext: anObject;
- findPrev: anObject;
```

findNext: and findPrev: repeatedly perform next or prev until the member value of the index matches their argument. nil is returned if the index reaches the end of valid members without matching the argument. They are only convenience shorthands for loops which could otherwise be programmed using next or prev.

```
- get;
- put: anObject;
- remove;
```

get returns the member value at which the index is currently positioned, or nil if the index is not positioned at a member The put: message replaces the member value at the current index position with its argument. Neither of these messages changes the current position of the index.

The get message provides an alternate way to obtain the current member value in a loop that traverses a collection; its return value is the same as next or prev would return when first positioning to a new member.

The remove message removes the member at the current location of an index, and returns the member value removed. The index position is set to a special position between the members which previously preceded and followed the removed member. If there is no preceding or following member, the index is set to the special location before the start or after the end of all members. After a current member is removed, there is no member at the current index location, but a subsequent next or prev message will continue with the same member that would have been accessed had the current member not been removed.

Both put: and remove raise the InvalidIndexLoc error if the index is not positioned at a current member.

```
- getLoc;
- (void) setLoc: locSymbol;
```

depends on the specific type of the collection, just as noted for the atOffset: message on Collection. In the worst case, this operation is linear in the magnitude of the offset.

```
- (int) compare: anIndex;
```

The compare: message compares the current location of one index with the current location of another index passed as its argument. If the two indexes have the same location, compare: returns zero. Otherwise, compare: returns +1 or -1 according to whether the argument index precedes or follows the receiver index in the enumeration sequence of the collection. If either of the two indexes has an unknown offset, and the location of the other index is anything other than Start or End or an immediately adjacent member, compare: returns the UnknownOffset integer value.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

however, the Array type can be used to wrap an existing C array for external access as an object collection. It can also provide access to an internal C array for direct manipulation using C expressions. These forms of low-level access support hybrid modes of use in which advantages of both low-level manipulation and external object access can be combined.

Inherited behavior

```
@deftype Array <Collection, CREATABLE>
```

The Array type is directly creatable, and supports all standard messages of Collection except removal of individual members. The messages based on an integer offset, either on the collection (atOffset:, atOffset:put:), or an index (setOffset:) all execute in fast constant time. Members of an array are fully ordered according to these integer offsets. Sequential access to members through its members is also fully supported. The Array type disables the remove message inherited from Collection; the message is defined, but any attempt to remove a member will raise an error that the operation is not supported.

While the IndexSafety option of an Array defaults to Unsafe, indexes are inherently safe under all operations with the exception of resizing an array or resetting its member allocation. (The only individual member operation which modifies contents of an array is replacing a member value, and this does not affect existing indexes.) The UnsafeAtMember value of IndexSafety is not supported since individual members cannot be added or removed. A value of SafeAlways makes existing indexes safe even under resizing or reallocation. If an existing index referred to a member location that is no longer contained in the array, the index location is reset to the End location.

The default value of the ReplaceOnly option is true, and cannot be overridden.

The type of index returned by begin: on an array is simply Index. There is no special index type for Array because there are no additional messages beyond those already defined by Index.

Create-time options

```
SETIING
- (void) setCount: (int)count;
- (void) setDefaultMember: memberValue;

- (void) setMemberBlock: (void *)members setCount: (int)count;
USING
- (void) setDefaultMember: memberValue;
- getDefaultMember;

- (void) setMemberBlock: (void *)members setCount: (int)count;
```

being used, the external program is entirely responsible for assuring that the MemberBlock value is a pointer to valid allocated memory containing at least the number of member slots given by setCount:.

With an external member allocation, the array itself will not attempt to either allocate this memory or free it when the array is dropped. Dropping the array only removes its reference to the external allocation.

The pointer to allocated memory is given as a pointer to id type, but if a data type given as MemberType has size greater than id, the pointer must actually point to an array of slots each of which has the size of the data type.

Since an array neither allocates nor frees an external member allocation, the same region of allocated memory may be referenced by multiple arrays, including overlapping member ranges each defined by starting location and count. This flexibility enables alternate subrange views of a single, contiguous initial allocation by means of separately created external collections.

Create combination messages

```
CREATING
+ create: aZone setCount: (int)count;
+ create: aZone setMemberBlock:(id *)members setCount:(int)count;
```

In addition to the standard inherited create messages, the Array type supports two forms of combination messages which both create the array and set its initial options. These are provided both for convenience and to provide the fastest possible array creation for simple cases in which speed might be important. The two cases supported are creating an array with a member count for its internal members already set, and creating an array which refers to an external member allocation. If any more options than these are needed, the normal method of setting options between createBegin: and createEnd messages should be used instead.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Inherited behavior

```
@deftype List <Collection, CREATABLE>
```

```
The List type is supports all messages of Collection.  If created with default options, it
provides no special speedup of accesses by integer offset.
```

```
Unless the EndsOnly option is specified, the default value of IndexSafety is UnsafeAtMember.
This means that indexes are inherently safe under all operations except those which
affect the member at an index or immediately adjacent to an index between members.
If the EndsOnly option is specified, the default value of IndexSafety is Unsafe.  In
this case, indexes could be invalidated by any operation that adds or removes members.
The SafeAlways can be specified for any case to get a guarantee of full index safety.
```

```
ListIndex is a special index type just for list collections.  It provides the only
method to add members within the interior of a list.
```

Create-time options

```
CREATING
- (void) setEndsOnly: (BOOL)endsOnly;
USING
- (BOOL) getEndsOnly;
```

EndsOnly

```
(..  This option is not implemented yet.)  The EndsOnly option restricts a list so that members
can be added or removed only at either end.  This restriction permits an alternate implementation
that both reduces memory usage and enables fast, direct access to members by integer offset.
It obtains key performance advantages of an array while preserving the dynamic extendability
of a list.
```

List behavior

```
- (void) addFirst: anObject;
- (void) addLast: anObject;
```

```
The addFirst:  and addLast:  messages add a new member to either end of a list.  With addFirst:,
the new member becomes the first member of the array.  With addLast:, it becomes the last member.
```

```
- removeFirst;
- removeLast;
```

```
removeFirst and removeLast remove either the first or last member of a list.  They each return
the value of the member just removed.
```

Name

Stack, Queue – restricted forms of lists

```
(..  Neither of these types is currently implemented.  They are nothing more than
restricted cases of a List, so if you need them use a List instead.)
```

Synopsis

```
// Stack -- form of list with add/remove both at start of list
//
@deftype Stack <List>- (void) push: anObject;
- pop;
@end

//
// Queue -- form of list with add at start of list and remove at last
//
@deftype Queue <List>- (void) enqueue: anObject;
- dequeue;
@end
```

Description

A Stack is a subtype of List which restricts the addition and removal of members to the start of a list. Queue is a subtype of List which restricts addition of members to the start of the list and removal to the end. Both types define specially named messages to reflect the specialized usage of their resulting add and remove operations.

Interface Reference

Inherited behavior

```
        @deftype Stack <List>@deftype Queue <List>
...
```

Name

KeyedCollection, KeyedIndex – supertype of Set and Map types

Synopsis

```
//
// KeyedCollection -- supertype of Set and Map types
//
@deftype KeyedCollection <Collection>CREATING
- (void) setDuplicatesOption: duplicatesOption;
- (void) setGroupType: aCollectionType;
- (void) setSingletonGroups: (BOOL)singletonGroups;

- (void) setSorted: (BOOL)sorted;
- (void) setCompareFunction: (idXid_t)aFunction;
-
(void)        setBucketFunction: (bucket_t)aFunction setBucketCount: (int)n;

- (void) setPartiallyOrdered: (BOOL)partiallyOrdered;
- (void) setMemberSlot: (int)byteOffset;
SETTING
- (void) setPartialOrderContext: aKeyedCollectionCollection;
USING
- getDupOption;
- getGroupType;
- (BOOL) getSingletonGroups;

- (BOOL) getSorted;
- (int(*)()) getCompareFunction;
- (int(*)()) getBucketFunction;
- (int) getMaxBuckets;

- (BOOL) getPartiallyOrdered;
- getPartialOrderContext;

- (int) getMemberSlot;

- at: aKey;
- (BOOL) containsKey: aKey;
- (int) getCountAtKey: aKey;

- removeKey: aKey;
```

Inherited behavior

```
@deftype KeyedCollection <Collection>
```

The KeyedCollection type inherits all standard behavior of Collection. The KeyedCollection type is not itself creatable; it only serves as a common supertype for Set and Map collection types.

The default value of IndexSafety for a KeyedCollection is Unsafe, but values of either UnsafeAtMember or SafeAlways can be requested to guarantee a higher level of safety.

Create-time options

(.. None of these options is implemented yet, except for CompareFunction.)

```
CREATING
- (void) setDupOption: dupOption;
- (void) setGroupType: aCollectionType;
- (void) setSingletonGroups: singletonGroups;

- (void) setSorted: (BOOL)sorted;
- (void) setCompareFunction: (idXid_t)aFunction;

- (void) setPartiallyOrdered: (BOOL)partiallyOrdered;

- (void) setMemberSlot: (int)byteOffset;
SETTING
-
(void)        setBucketFunction: (bucket_t)aFunction setBucketCount: (int)n;
- (void) setPartialOrderContext: aKeyedCollection;
USING
- getDupOption;
- getGroupType;
- (BOOL) getSingletonGroups;

- (BOOL) getSorted;
- (int(*)()) getCompareFunction;

- (BOOL) getPartiallyOrdered;
- getPartialOrderContext;

- (int) getMemberSlot;

- (int(*)()) getBucketFunction;
- (int) getBucketCount;
```

Sorted

If this option is true, the immediate members of a keyed collection (included collections of duplicate members, if any) are totally ordered according to an ordering relation defined on key values. The default order for sorting is determined by a standard compare: message on a key value to be added vs. existing keys of the collection. This default method to establish order can be overridden using the CompareFunction option.

The default for the Sorted function is true, unless a value is also given for the option BucketFunction.

CompareFunction

Use the function pointed to by the argument as the method for determining whether two members have the same key value, and also for ordering the keys of the collection if the Sorted option is true. The argument is declared with the following function pointer type:

```
typedef int (*idXid_t)( id, id );
```

The function given will be called whenever one key value needs to be compared with another. Multiple calls to the function typically occur whenever members are added or removed from the collection, until the correct member for insertion or removal is determined.

The compare function accepts two key values as its arguments. Like member types, key types declared with the id pointer type, but may contain any type of value (such as an integer) up to the size of an id. An explicit compare function can support any type of value as a key, regardless of whether it is an object that supports a standard compare: message.

The compare function is called repeatedly by the collection to compare two key values. The function should return zero if the key values are equal, -1 for the first key argument less than the second, and +1 for the first greater than the second. If a keyed collection is not sorted, either -1 or +1 may be returned for unequal keys, regardless of whether one might be taken as greater or less than the other.

BucketFunction

The BucketFunction option specifies a function that is called whenever a member of an unsorted collection is added or removed. It may also be used for a sorted collection to provide faster insertion and removal of members. The responsibility of a "bucket function" is to assign the key value to some integer in a range from zero up to some maximum value. These integers identify "buckets" to which the same key value will also be assigned whenever passed to the function. If the collection is unsorted, these buckets represent hash buckets, and for speed should be relatively sparse (few keys likely in any one bucket, given the expected size of the collection). If the collection is sorted, these buckets represent ordered buckets to which the same range of key values will always be assigned. Such sorted buckets can speed up the search for a key location within a sorted collection. Sorted buckets can also be sparse to minimize the need for further sorting within each bucket. If many keys end up in the same sorted bucket, the sorting process normally used without buckets is still used, so there is less performance impact than with hash keys of an unsorted collection.

getMemberSlot

The MemberSlot option indicates that space has been reserved within each member that allows it to contain a link directly to its position in the enumeration for a collection. If such space has been reserved, special messages can be used that rapidly position an index directly to the member. Operations to remove members are also much faster.

The value of MemberSlot specifies the offset in bytes from the start of each member where the space for its position link has been reserved. The following typedefs are provided to declare the space required for a position link within a member allocation:

```
typedef struct memberData { void *memberData[2]; }            member_t;
typedef struct memberData { void *memberData[2]; id owner; }  dupmember_t;
```

The first type declares the required space for a position link in a collection that does not accept duplicate key values. The second type is required if duplicate keys are permitted.

The offset of the position link from the start of a member may be either positive or negative, in the range of -2048 to +2047. The default value of MemberSlot is UnknownOffset (a large negative value), which specifies that no slot for an internal position link is available within each member.

If a member slot is defined, a memory pointer of some kind must always be used for the value of every member. These members may be object id pointers, but other types of memory pointers are acceptable as well.

The contents of the first two words of position link vary according to the position of a member in a collection. The third word of a dupmember_t link contains the id of the collection in which a member is directly contained. This collection is either an internally created collection containing members with duplicate keys, or the collection to which the member was added if there is no other member with the same key.

KeyedCollection behavior

(.. None of the messages or other behavior relating to duplicate key values are currently implemented.)

```
- at: aKey;
- (BOOL) containsKey: aKey;
- (int) getCountAtKey: aKey;  // unimplemented

- removeKey: aKey;
```

The at: message returns the existing member of the collection which matches the key value passed as its argument, or nil if there is no key value in the collection which matches. If duplicate entries for this key exist, the entire collection of duplicate members created for the key value is returned instead.

The containsKey: message returns true if the key value passed as its argument is contained in the collection, and false otherwise. The getCountAtKey message returns

```
- setMember: aMember;
```

(.. This message is not currently implemented.) The setMember: message repositions the index to the member passed as its argument. This message is valid only if an internal member slot was defined for the collection with the MemberSlot option.

Roger Burkhart <rmb@santafe.edu>
(mailto:rmb@santafe.edu)

Last modified: 1997-12-19 (mgd)

@end

The add: message adds a new member to a set. It returns true if member added matched the key of any member already contained in the Set. Whether or not the member was actually added to the collection or not depends on the setting of DupOption.

The add:setIndex: message adds a new member to a set just like add:, but also has a side effect of setting the position of the index passed as the setIndex: argument to the member which was just added. If an index positioned to the new member is needed anyway, setting the index in the same operation avoids repeating a search for the position at which the member key belongs.

The replace: message replaces a member stored at a given key with another member value that matches the same key. The new value to replace the existing one is passed as the argument. replace: returns the member value that was replaced, or nil if the collection contained no member with a matching key.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

Map – collection of associations from key objects to member objects

Synopsis

```
//
// Map -- collection of associations from key objects to member objects
//
@deftype Map <KeyedCollection, CREATABLE>CREATING
- (void) setKeyType: aDataType;
- (void) setKeySlot: (int)byteOffset;
USING
-         getKeyType;
- (int) getKeySlot;

- (BOOL) at: aKey insert: anObject;
- (BOOL) at: aKey insert: anObject setIndex: anIndex;
- at: aKey replace: anObject;

- (BOOL) insertGroup: aKey;

- createIndex: aZone setKey: aKey;
@end

@deftype MapIndex <KeyedIndex>- getKey;
- setKey: aKey;

- get: (id *)key;
- next: (id *)key;
- prev: (id *)key;

- replaceKey: aKey;
end
```

Description

Map is a subtype of KeyedCollection in which the key value associated with each member is independent of the member itself. Whenever a new member is added to the collection, a key value to be associated with the member must be supplied also. A Map defines a mapping from key values to member values.

(.. This message is not currently implemented. It will eventually give a fast way to create an index already positioned at an entry with a particular key value.)

MapIndex behavior

```
- getKey;
- setKey: aKey;
```

The getKey message returns the key value associated with the current location of the index. It returns nil if the index is not currently positioned at a member. The setKey: messages repositions the index to an entry having a key value that matches its argument. If there is more than one entry matching this key value, the index is positioned to the first entry that matches.

```
- get: (id *)key;
- next: (id *)key;
- prev: (id *)key;
```

These messages are similar to those defined by the Index supertype, except that they combine the return of a member value with an output value for an associated key value. In each case, the key value is passed as an (id *) output argument.

```
- replaceKey: aKey;
```

(.. This message is not currently implemented.)

> replaceKey: replaces a key value with another key value that still compares equal to a key value already present. It may be used to replace an allocation of one object, already contained in a Map, by a different object allocation. replaceKey: returns nil if no existing key value matches its key value argument; otherwise it returns the key value which was replaced by its argument.

Marcus G. Daniels <mgd@santafe.edu> (mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Create-time options

```
        CREATING
        + create: aZone setC: (const char *)cstring;
        SETTING
        - (void) setC: (const char *)cstring;
...
```

String behavior

```
        - (const char *)getC;
        - (void) appendC: (const char *)cstring;

        - (int) getCount;
        - (int) compare: aString;
        @end
...
```

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

```
...
```

Special create: message

```
        + create: aZone setFileStream: (FILE *)fileStream;
...
```

OutputStream behavior

```
        - (void) appendC: (const char *)cstring;
        @end
...
```

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Collections Library: Interface Design Notes

Design Objectives

A collections library is one of the most important foundation services for object-oriented programming. Most object-oriented systems provide at least the start of a general-purpose collections library. The GNUSTEP, project, for example, provides the libobject library (currently in alpha test at ftp://alpha.prep.ai.mit.edu/) which includes a collections library along with other services intended to parallel those of the OpenStep framework developed by Next.

> Swarm has implemented its own collection library to meet the specialized needs of its agent simulation framework. Major objectives of the Swarm collections library include the following:
>
> ... (.. more to come, obviously)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Activity Library

Documentation and Implementation Status

Overview

The activity library is responsible for scheduling actions to occur within a simulated world, and for making these actions actually happen at the right time in the right order. It provides the foundation of dynamic, object-oriented simulation within Swarm.

> Actions consist of messages to objects, calls to functions, or groups of actions in some defined order. The activity library guarantees that all these actions, and the state changes they produce, occur at predictable points in time. Time is defined by the relative order of actions, and may also be indexed by the discrete values of a world clock.

Documentation

The activity library follows the Swarm library interface conventions

(defobj/library.html)
and has the following standard sections of documentation:

- Usage Guide
(activity/guide.html)
- Advanced Usage Guide
(activity/adv.html)
- Interface Reference
(activity/ref.html)
- Subclassing Reference
(activity/subclass.html)
- Interface Design Notes

Activity Library: Documentation and Implementation Status

The activity library is in process of being converted to draw even more of its support from underlying types and classes in the defobj and collections libraries. The basic interface has already been changed to reflect this consolidation, so the programming interface for currently available function is expected to remain stable.

The activity library follows the documentation structure of the Swarm library interface conventions
(../defobj/library.html)
. There are placeholders for each section of documentation so that all links should at least link up with something, whether or not there's anything there. The Interface Reference section is already fairly complete, and there is a start toward a more complete Usage Guide section. The Advanced Usage Guide and other sections will come at a later time.

Throughout the documentation, a parenthesized comment that starts with (.. indicates an editorial comment on the current status of implementation or documentation. In this version, however, an attempt has been made to remove all discussion of future capabilities and to document instead only the capability that is already there.

Source for code examples in the Usage Guide is included in the text where it occurs. Source consists of fragments taken from example programs. The example programs themselves may be downloaded from the web pages and compiled on your own system, using links provided for that purpose. (.. currently there are no code examples in the Usage Guide, which so far is only a general overview of the activity library ..)

There is also a directory of test programs (GridTurtle test programs
(../grid.html)
, contained within the documentation release directory) that provides additional code examples of many basic features of the defobj, collections, and activity libraries. These code examples are very rough and subject to change, however, since their main use is with each new release of the libraries to test various new or existing features. They are sometimes the only source, however, for examples of some library features that have not yet been used anywhere else.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

types controls and tracks a current state of processing within these representations.

Both these representations are more abstract than typical object representations, since they deal not with any constant state which can be statically analyzed, but with shifting patterns of messages to be fired to generate such state. While the basic structure of a Swarm simulation is that of a discrete event simulation, its activity representation is also more complicated than most such systems.

There are two basic reasons for the potential complexity of a Swarm activity representation. One is that Swarm supports a decentralized representation of activity to be generated, both to reflect the nature of much of the behavior that it simulates, and so that execution may be distributed across multiple parallel processors. To avoid any need for synchronizing its decoupled activities more often than necessary, Swarm enables a model designer to avoid overspecifying constraints on the patterns of message sends which might potentially be valid. The partial order representation it adopts for a distributed and decoupled plan of activity is inherently more complicated than the single centralized event list adopted by many discrete event simulation systems.

The other reason for potential added complexity is that Swarm's representation of intended activity may be broken into many separate, modular components, which can be bound together in various ways to create larger components. The Swarm composition structure, when fully implemented, will be fundamentally as powerful as the modular abstractions found in most programming languages, with the added complexity of controlling the time at which various events occur.

In spite of this high potential complexity, none of these features is needed for many simple models, such as those that contain only a small variety of basic behaviors, or which define all their behavior to occur at regular, repeating timesteps. The Swarm representation is extremely rich to enable it to scale to large, multi-level models with a variety of dependent behaviors built into the model, and also to facilitate running models on massively parallel machines. Many of the features which seem complex, moreover, are also especially well-suited to building modular and reusable library components. It is anticipated that many of the more advanced features will find their heaviest usage in pre-built libraries that hide internal complexity from applications that use them.

No matter how complex the structures built from them, all the activity library components finally result in the direct execution of messages sends to objects in a model. Because of this direct execution, very precise meaning can be defined for each of them, in terms of message sends that can or must occur. Every component of a structure to be executed, and every event of its execution, can also be accessed using the interfaces of an object-oriented representation. This means that tools can be built to understand and interact with any components to be executed, and also to trace and debug activity as it occurs. None of these tools have been built yet as part of Swarm, but many of the activity library interfaces are present to support them, not because normal use of the library requires the added components.

Model execution components

The main responsibility of model execution is just to perform the actions specified by an action plan, in an order of execution consistent with any requirements of the plan. Each action plan may specify as few or as many constraints as it likes over the possible order in which its actions could be performed. Given these specifications, the execution objects are entirely responsible for selecting each action to be performed and then performing them.

There are two simple cases of ordering that account for most all usage, including that of the current Swarm demo programs. One is to permit the actions to be performed in any order, including concurrent execution if hardware and software support were available to do this. The other is to require them to be performed in some specific sequence, one after another, so that the effects of one action could depend on actions which preceded it. This sequence could be either fixed and predetermined, or dynamically established each time the action plan is performed. If a dynamically determined sequence is needed, perhaps even selecting which actions are to be performed at all, users can provide custom subclasses for an action plan and an execution object that performs it. A builtin option is to generate an entirely random sequence each time a plan is executed.

Each time an action plan is being processed, a special kind of object called an activity is created entirely automatically by the runtime processing machinery. These activity objects implement the internal machinery of a virtual processor which has the ability to execute action plans. To get any activity started on a model, a processor is first initialized to run a single top-level plan. All other activity during the lifetime of a model must occur as a result of actions initiated by this plan (which may include the startup of other plans).

Because the activity objects are internal to the processing machinery, an application can usually just ignore their existence. They come and go dynamically as various action plans are started and completed, all arranged in a stack or tree of current activities controlled by a single top-level processor. The activity objects are potentially useful, however, to obtain information about the context in which an action is currently running, or to build debugging or tracing tools to understand actions being performed.

One of the kinds of context information available from an activity object is the current time of a clock value incremented as a schedule is processed. A schedule is a kind of action plan in which all actions occur at specific points in time explicitly established within the schedule. As a schedule is executed, the activity object keeps a current time clock, which holds a global time from the start of all model execution, regardless of the time when the schedule itself was started. An activity object that processes a schedule is called a timeline activity, because its time continually increases from a global base time regardless of times contained in the schedule.

New activities are created whenever an action plan being processed contains an action to perform another action plan, or to start another action plan for autonomous execution. If one action plan performs another, its own processing is stopped until a new activity

Activity Library: Advanced Usage Guide

(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

ActionType – specification of an executable process

Synopsis

```
//
// ActionType -- specification of an executable process
//
@deftype ActionType <ProcessType>- activateIn: swarmContext;
@end
```

Description

An action type is a type of process that may be initiated as a unit of execution by an external request. A typical action has a well-defined duration determined by a fixed set of actions that execute within it. Externally initiated interaction typically occurs only at the start or end of the overall process. A typical action is executed in its entirety once an external request that initiates it has occurred. Some actions may also have internal events that cannot begin or complete until other actions from a containing environment have also begun or completed their execution. Such ordering constraints can be defined either within an action type or as part of a dynamic context of execution.

Executable actions include both actions compiled in a host language (such as C functions or Objective C messages) and compound actions built at runtime for interpretation by the Swarm abstract machine.

(.. For now, the only subtype of ActionType is CompoundAction. Types for compiled actions such as functions and messages have not been defined yet. ..)

Inherited behavior

```
@deftype ActionType <ProcessType>
```

ActionType inherits standard parameterization support from ProcessType. (.. Currently, no such support is defined. ..)

Name

CompoundAction – collection of actions to be performed in some order

---

Synopsis

```
//
// CompoundAction --
//   a collection of actions to be performed in any order consistent with a
//   set of ordering constraints
//
@deftype CompoundAction <;ActionType, Collection>CREATING
- (void) setDefaultOrder: aSymbol;
- (void) setAutoDrop: (BOOL)autoDrop;
USING
- getDefaultOrder;
- (BOOL) getAutoDrop;
@end
```

Description

An compound action is the supertype of ActionGroup and Schedule. A compound action defines an executable process that is composed from the execution of a set of actions in some defined order.

Inherited behavior

```
@deftype CompoundAction <ActionType>
```

CompoundAction is not directly creatable. One of its subtypes must be created instead. ActionPlan inherits the basic ability to be activated for execution from ActionType
(ActionType.html)
.

---

is especially useful for a dynamic schedule that continually receives new actions to be executed at future times, but will never repeat actions that were previously scheduled. Depending on the underlying implementation of the schedule, making sure the old actions get dropped using AutoDrop can considerably improve the performance of the schedule.

When an option like AutoDrop is used, or whenever the contents of an action plan undergo a significant amount of dynamic update, the action plan would ordinarily be intended only for a single point of use. If AutoDrop is specified, a restriction against multiple active references is enforced. An error will be raised if two activities ever attempt to process a plan with AutoDrop enabled at the same time.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

option inherited from ActionPlan. This initial order may also be further constrained by partial order constraints among the actions. (.. Partial order constraints are currently unimplemented.)

Inherited behavior

```
@deftype ActionGroup <CompoundAction, ActionCreating, OrderedSet, CREATABLE>
```

ActionCreating defines the createAction messages for ActionGroup just so that this protocol may be shared with Schedule, where they provide a convenience interface for the creation of actions in the schedule at time zero. These messages will be documented here in ActionGroup as if they had been defined on ActionGroup itself.

An action group inherits its underlying representation from the OrderedSet
(../collections/OrderedSet.html)
type of the collections
(../collections.html)
library. All the members of the ordered set must consist only of actions that are created by one of the createAction messages defined on ActionGroup itself. Once the actions are created, they may be accessed or traversed using standard messages of the OrderedSet type. Partial orders on these actions are also established by inherited messages. (.. Currently, partial orders are not implemented on either collections or action plans, but the messages inherited from OrderedSet will be the basic method for maintaining them once implemented.)

ActionGroup behavior

Actions are created within the ActionGroup using one of the createAction messages. A createActionTo: message specifies that the action to be performed is defined by an Objective C message selector to be performed on a receiving object plus any required arguments. The message selector is specified by the message: argument and the receiving object is specified as the first argument, target.

Given a variable aTurtle already initialized to a valid object, following is an example of a createActionTo message:

```
[createActionTo: aTurtle message: M(print)];
```

M() is a macro defined in defobj.h which is just a convenient shorthand for the Objective C expression @selector(message-name). As in any selector, the message name must include all parts of the message including colons and subsequent argument portions. An example of a message with one additional argument is the following:

```
[createActionTo: aTurtle message: $m(move:) : (id)10];
```

The createAction messages declare all arguments of the message to be of object id type, but you are free to cast other pointers and values up to the limits defined by the global portability assumptions
(../defobj/globalport.html)
. These is not portable across all machine architectures, but is expected to be portable across the 32-bit and 64-bit architectures on which Swarm will be supported. The message you send

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

@end

Description

A schedule is compound action whose basic representation is a sorted Map of actions that have been created within it. The key value associated with each of these actions is an unsigned integer value for which the typedef timeval_t

is supplied.

Inherited behavior

@deftype Schedule <CompoundAction, ActionCreating, Map, CREATABLE>

A schedule inherits its underlying representation from the Map
(../collections/Map.html)
type of the collections
(../collections.html)
library. All the members of the ordered set must consist only of actions that are created by one of the createAction messages defined on Schedule itself. Once the actions are created, they may be accessed or traversed using standard messages of the Map type. The key values of this collection, however, must be cast to and from the id type defined for key values by the Map type. Partial orders on these actions can also established by messages inherited from Map. (.. Currently, partial orders are not implemented on either collections or action plans, but the inherited messages will be the basic method for maintaining them once implemented.)

Create-time options

RelativeTime

The RelativeTime option specifies that all the times in the schedule are relative to the time when processing of the entire schedule begins. Otherwise, the times are assumed to be absolute times, with their base in the starting time of the entire model.

(.. This option is currently supported only on schedules, not swarms.)

RepeatInterval

The RepeatInterval option specifies that as soon as all actions in the schedule have completed, it is to be rescheduled at a time computed as the time at which the schedule was last started, plus the value specified as the RepeatInterval argument. This option overrides the normal

Ordinarily, a concurrent action group is created to process actions at the same timestep only if more than one action is scheduled at that timestep. The overhead of these action groups is relatively low, because it just creates a single new object to which actions are directly linked, but it is still faster to avoid creating them if only one action is present at a timestep. If a custom subclass is being provided however, it may need to examine the actions at a timestep even if there is only one.

---

timeval˙t – type for clock values

The following typedef is provided for convenience in declaring time values:

```
typedef unsigned   timeval_t;        // type for time values
```

Values of this type are used as keys when inserting actions into a schedule at a particular time, or for querying the current time value of a swarm or schedule during its execution. (The shorter name time_t has already been taken by one the standard C libraries.)

For very long running models using finely divided units of time, it is quite possible that a 32-bit unsigned time value could overflow. Special messages are provided in the execution machinery to reset all times to a different base value if a danger of overflow exists. If this machinery is exercised, all times of all referenced action plans are reset to a different base value in unison, which makes the shift as transparent as possible. Separate support is also available to declare time values that are subunits of the discrete clock values in a containing schedule or swarm. (.. Currently, all this extended time unit support is unsupported, though there is an example of scheduling at subunit times in a GridTurtle
(../grid.html)
test program.)

Schedule behavior

The messages to create actions within a schedule are essentially the same as those for ActionGroup

(ActionGroup.html)
, except for the presence of an initial at: argument indicating the time at which an action is to be performed. Except for the time associated with each action, meaning of the createAction messages is the same as for ActionGroup.

When multiple actions are all scheduled at the same time, they are all inserted into a concurrent action group created for that time value. The ConcurrentGroupType option may be used to override the default action group for these concurrent actions by a custom user-defined subclass. (.. Details of doing this are not yet documented, but there are examples.)

Name

SwarmProcess – object that encapsulates a collection of processes

---

Synopsis

```
//
// SwarmProcess --
an object that holds a collection of concurrent subprocesses
//
@deftype SwarmProcess <ActionType, Zone, CREATABLE>CREATING
- setInternalZoneType: aZoneType;
- (void) setSynchronizationType: aScheduleType;

- (void) setInternalTimeMultiplier: (timeval_t)multiplier;
USING
- getSynchronizationType;
- (timeval_t) getInternalTimeMultiplier;

- getInternalZone;
- getActivity;
@end

//
// ActivationOrder -- default concurrent group type for swarm
//
@deftype ActivationOrder <ActionGroup, CREATABLE>@end
```

Description

A swarm is an object created to coordinate the processing performed on other action types. Just like any other action type, a swarm may be activated for running as a subactivity of another swarm. A swarm contains no actions of its own, since its only purpose is to coordinate the processing of other activities.

SwarmProcess is the type of an object that can be activated to run in some context, and which itself then defines a context in which other action plans can be activated. Subtypes of this generic type can each define a custom interface to activities running within the swarm. The generic swarm type defines only access to an underlying activity that keeps track of the activated or running subactivities, along with an option that may be set to a collection of objects currently being processed by the swarm.

InternalZoneType

The InternalZoneType option sets the type of zone which is created by the swarm to hold objects created within the swarm. If set to nil, no internal zone is created within the swarm, and all use of the swarm as if it were a zone raises an error. The default of this option is standard Zone type. (.. Since there is no other Zone type yet, there's no reason to set this option yet except to turn off the internal zone. ..)

SynchronizationType

Synchronization type sets the type of schedule which is used internally by the swarm to synchronize subschedules. Its default is a schedule with a concurrent group of ActivationOrder.

InternalTimeMultiplier

(.. This option will be used in the future to support an internal time unit within the swarm that can be more fine-grained than the context in which the swarm is included. ..)

SwarmProcess behavior

```
- getInternalZone;
- getActivity;
```

getInternalZone returns a Zone object that is used by the swarm to hold its internal objects. Even though the swarm itself inherits from Zone and can be used as a Zone for nearly all purposes, this message is also provided so that the zone itself can be obtained independent of all zone behavior.

getActivity returns the activity which is currently running of subactivities within the swarm. This activity is the same as the value returned by activateIn: when the swarm was first activated. It returns nil if the swarm has not yet been activated.

Unlike other action types, swarms and swarm activities always exist in a one-to-one relationship, provided that the swarm has been activated. This restriction to a single activity enables the swarm to do double-duty as a custom object that provides its own interface to the activities running within the swarm.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Description

Action is a common supertype of all action types which may be created within an action plan. Each action is always controlled by a single action plan to which it belongs. This action plan is referred to as its owner. Given the action object, its owner plan may be obtained using the inherited getOwner message.

> Actions are allocated in the same zone as their owner plan, and may be created only using one of the createAction messages on an ActionGroup or Schedule. Each of these messages returns the action that was created as its return value. Actions in an action plan may also be obtained by processing the plan using any of its messages inherited from its underlying collection. Actions may also be removed from an action plan using a remove message on the underlying collection.
>
> (.. Note: currently, an action cannot be removed while it is currently being executed. This means that the function or message called by an action cannot itself remove that same action from its action plan. This restriction will be removed in the future.) No matter what the type of action, the default ordering constraints established by its membership in an owning plan may be supplemented by explicit specification. These specifications are made directly between actions in the same plan, or between actions in different plans if they are suitably qualified. Given any action belonging to any plan, the set of specific explicit ordering constraints in which it participates may be obtained by one of the messages getSuccessors, getPredecessors, getCostarts, and Coends inherited from the KeyedCollection
> (../collections/KeyedCltn.html)
> type.
>
> Separate subtypes of Action are defined for each of the various forms of createAction messages that create them. The current representation of these actions will be undergoing change as support for their parameter and return types is migrated into more basic support from the defobj
> (../defobj.html)
> library. Each action type provides messages to retrieve and set the values of all argument values bound into the action. These capabilities will remain, but different messages will eventually be supported. This documentation will be completed once the messages supported on Action types are finalized.

Roger Burkhart <rmb@santafe.edu>
(mailto:rmb@santafe.edu)

Last modified: 1997-12-19 (mgd)

```
- (void) setOwnerActivity: ownerActivity;
- getOwnerActivity;

- getSubactivities;

- getControllingActivity;
- getTopLevelActivity;
- getSwarmActivity;
- getScheduleActivity;

- setSerialMode: (BOOL)serialMode;
- (BOOL) getSerialMode;

- getCurrentSubactivity;  // serial mode only
@end
```

Description

A object of type Activity implements the processing of actions within an action plan. Each subtype of action plan has a corresponding subtype of Activity that implements processing of its respective type of plan.

The Activity types are part of the virtual processing machinery of an action plan interpreter. All the important elements of this machinery, including their current state of processing, are exposed to aid in development of general-purpose tools such as debuggers. Except for applications that need to create and run their own reflective activities, direct use of activity types by a modeling application should be rare.

An new activity object is created by each activateIn: message sent to an action type. activateIn: initializes a new activity object and prepares it for processing, but does not itself execute any actions. Subsequent messages, such as run, must be used to initiate processing within an activity. If an activity is activated to run under a swarm context, the owning swarm itself controls all processing within the subactivity.

An Activity type holds a tree of currently running, or a potentially runnable, subactivities. Each activity object represents a state of processing within a single action plan. The structure is a tree because multiple activities could be running or eligible to run at the same time. This occurs whenever two activities are specified as concurrent, either because are being performed by concurrent actions in some other plan, or because they were started for autonomous execution under the same swarm. When parallel processing machinery is present, each activity could also be advancing its own state independent of those of any other activity.

The current implementation supports only a single serial processor. Under serial execution, only one activity may be active at one time. This means that the current

```
Released, Terminated, Completed;
```

The getHoldType returns a code for the particular hold constraint under which an activity is currently holding. It returns nil if the basic status of the activity is not Holding. (.. Currently no hold constraints other than merging within a swarm are supported, and this message always returns nil.)

```
extern id <Symbol>  HoldStart, HoldEnd;
```

Activity context macros

The current running activity is always available for reference within any action running within the activity. (.. Currently, the getCurrentActivity() macro raises an error, but the other macros work as long as there is a currently running activity. ..) The following macros provide access to various information available from the current running activity and its owner activities:

```
#define  getCurrentActivity()
#define  getCurrentAction()
#define  getCurrentTime()
#define  getCurrentScheduleActivity()
#define  getCurrentSchedule()
#define  getCurrentSwarmActivity()
#define  getCurrentSwarm()
#define  getCurrentSwarmActivity()
#define  getTopLevelActivity()
```

Activity subtypes

(.. Only a synopsis of Activity subtypes is provided here for now. Note, however that access to a current time is available from a schedule or swarm activity, along with the ability to run a schedule until a particular time value is first reached.)

Name

GroupActivity – state of processing within an ActionGroup

Name

SwarmActivity – a collection of started subactivities

Synopsis

```
//
// SwarmActivity -- a collection of started subactivities
//
@deftype SwarmActivity <ScheduleActivity>
@end
```

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Activity Library: Interface Design Notes

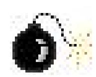
(This section of documentation is not yet available.)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Objectbase Library

Documentation and Implementation Status



Compatibility

- 1.0.4 => 1.0.5
  No changes.
- 1.0.3 => 1.0.4
  The name of this library is now **objectbase**, it is has been renamed from **swarmobject** largely to reflect the more generic nature of the library and also motivated by the impending port of Swarm to Windows NT (to avoid filename conflicts with the SwarmObject class). There should be little, or no effect on the user, the only visible change is the fact that the actual library (.a) or (.so) file will now have a different name and the header file name has changed. A symbolic link from objectbase.h to swarmobject.h has been provided in the distribution, to ensure backwards compatibility, however, users should not continue to rely on this being so. Users should port references to swarmobject.h in their code to objectbase.h, because this symlink will be removed in a future release.
  Note: this is no way affects the SwarmObject class which remains the same as in all previous releases.
- 1.0.0 => 1.0.1
  The interface has changed again! *EmptyProbeMap* is now a subclass of *CustomProbeMap*, which is subclassed from *ProbeMap*. And a shortcut create: method was added to that branch.
  Also, a new method was added to *ProbeLibrary* called isProbeMapDefinedFor that serves to non-invasively test for the existence of a *ProbeMap* for a given class.
- *Beta* => 1.0.0
  The new interface for the *swarmobject* library might cause some problems for apps that worked under the Beta release of Swarm. To get the whole scoop, read the Library Interface Conventions
  (defobj/library.html)
  . To get a few hints at explicit changes you might need to make, check out: Porting apps to use the new *swarmobject* interface
  (objectbase/PortingAppsToNewInterface.html)
  .

(probing.html)
will explain more about the reasoning and purpose behind probes.

Documentation

The *objectbase* library is defined and documented following the basic structure of the Library Interface Conventions

(defobj/library.html)
. The following standard sections of documentation are available:

- Interface Reference
(objectbase/Interface.html)
- Usage Guide
(objectbase/UsageGuide.html)
- Advanced Usage Notes
(objectbase/AdvancedUsage.html)
- Interface Design Notes
(objectbase/InterfaceDesign.html)
- Implementation Notes
(objectbase/Implementation.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Swarmobject Library: Porting Apps to the New Interface

The problems that might arise because of this new interface should have to do with the placement or existence of C preprocessor `#import`s. In the beta release, the `objectbase.h` file simply included all the header files for each class defined in the objectbase library. These files contain the `@interface` declarations for each respective class. Hence, before, if an application file included `objectbase.h`, it was including the interface for every class in the library.

The Swarm standard module interface uses Objective-C protocols to establish which messages may be sent to objects of a class. Doing this isolates the classes in the library more completely from the rest of the code. So, the problems that are likely to arise from this new interface are ones of access to objects that reside in the objectbase library. And they are of two types:

1. Messages that are no longer available on one or more classes where they were available before. I.e. you see warnings like the following when you compile your app:

```
warning: `ClassName' does not respond to `messageName'
```

What this means is that since `objectbase.h` simply imported all the class header files, somewhere in that chaotic mess of an import tree, the header for the `ClassName` was imported. Now, since `objectbase.h` only defines protocols for the classes it contains, it no longer imported that header file.

Solution #1: If `ClassName` is a member of the objectbase library, then all you need to do is import `objectbase.h` in the file where that class (or an instance of it) is used. If, however, the class in question is a member of some other library, then you should find out which library it is a member of and include the appropriate header file. (This latter case is actually part of the justification for making these libraries into "modules" like this.... to avoid incestuous include trees.)

Another case where this occurs is when using objects that have been statically typed with a class name. In using the protocols instead of direct class interfaces, we are depriving the compiler of the ability to analyze the type of an object and decide if that object can accept a given message. This is a bit counter intuitive for those of us reared with strongly typed languages and styles. But, Swarm takes advantage of the freedom this allows and it is part of Swarm-style programming
(../defobj/style.html)
.

Solution #2: If this is what's happening to generate the above warning, there are two further degradations of typing available. The next step down is to take away the specific class type and replace it with a protocol qualification on a pure `id` type. For example:

Objectbase Library: Interface Reference

The *objectbase* library uses the module definition conventions
(../defobj/module.html)
established by the *defobj* library
(../defobj.html)
. It also depends on standard supertypes and classes defined by this library.

Dependencies

Following are the other header files imported by <objectbase.h>:

```
#import <defobj.h>
#import <objectbase/SwarmObject.h>;
```

The *defobj* library interface is included to provide the basic object support. The *SwarmObject* class header file was provided for backward compatibility since the *objectbase* library has been brought into agreement with the library standards.

Object Types (with links to reference pages)

```
SwarmObject
 (SwarmObject.html)
                  /     |     \
                 /      |      \                  ProbeLibrary
 (ProbeLibrary.html)
                /       |       \
  ActivityControl
 (ActivityControl.html)
      Probe
 (Probe.html)
      ProbeMap
 (ProbeMap.html)
                     /   |        |  \   \
                   /     |        |   \    \
                 /       |        |    \     \
               /         |        |     \      \
        VarProbe
 (VarProbe.html)
       Message
 (MessageProbe.html)
   Custom
```

(CompleteProbeMap.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

  - + createBegin: aZone;
  - - createEnd;
  - + customizeBegin: aZone;
  - - customizeEnd;
  - - customizeCopy: aZone;
- Drop
  - - getZone;
  - - (void)drop;

In addition to these, SwarmObject is augmented by the messages that provide the probe support.

getInstanceName is a message used by the probe machinery to get a meaningful word or phrase to indicate the object being probed (and, in the case of a probe display, place it in a widget on the screen). Support for this has recently been added to the DefinedObject object type, thereby making it more generally available. So, this message will likely become obsolete soon.

getProbeMap returns a pointer to the ProbeMap for the object if there has been one created for that object's class. If it hasn't, then it create's a default one.

getCompleteProbeMap returns a newly created CompleteProbeMap for the object.

getProbeForVariable: (const char *)aVariable returns the VarProbe indexed in the ProbeMap by the string aVariable.

getProbeForMessage: (const char *)aMessage returns the MessageProbe indexed in the ProbeMap by the string aMessage.

(Interface.html)

(../objectbase.html)

(ProbeLibrary.html)

ProbeLibrary Class

```
//
// ProbeLibrary --
// A (singleton) Class, whose instance is used as a container for aglobal
// mapping between classnames and their 'default' ProbeMaps. These defaults
// can be changed by the user, thus allowing him/her to customize the default
// contents of the ProbeDisplays generated when probing objects.
//
@protocol ProbeLibrary <Create, Drop, ProbeConfig>
USING
- setDisplayPrecision: (int) nSigDisplay;
- (int)getDisplayPrecision;
- setSavedPrecision: (int) nSigSaved;
- (int)getSavedPrecision;
- (BOOL)isProbeMapDefinedFor: (Class)aClass;
- getProbeMapFor: (Class)aClass;
- getCompleteProbeMapFor: (Class)aClass;
- getCompleteVarMapFor: (Class)aClass;
- getProbeForVariable: (const char *)aVar inClass: (Class)aClass;
- getProbeForMessage: (const char *)aMessage inClass: (Class)aClass;
- setProbeMap: aMap For: (Class)aClass;
@end

extern id <ProbeLibrary> probeLibrary;
```

The normal Swarm simulation will probably only ever contain one instance of this class, namely the *probeLibrary* object. This object is used for Library Generation of *Probe*s and *ProbeMaps*: its role is to cache one unique "official" *ProbeMap* for every Class ever probed during a run of Swarm. These *ProbeMap*s are generated lazily as they are requested.

- setDisplayPrecision: (int)nSigDisplay

  Sets the number of significant digits for floating point and double floating point numbers displayed on GUI widgets. Currently only implemented for VarProbes not MessageProbes as yet. This allows *all* probes checked out the global ProbeLibrary instance to access this displayed precision. However, individual probes can vary from this global default, by using the setFloatFormat method on a exisiting probe. See probing precision (../probing.html#precision) for more details on how to do this.

- (int)getDisplayPrecision

In those cases where the user wants to set the standard probe map directly, she can provide the probe map using this method, and it will be cached as though it were produced by the library itself.

(SwarmObject.html)

(Interface.html)

(Probe.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

run sends a run message to the activity if the conditions are appropriate. This message causes the activity to continue executing the actions on it's schedule until either no other actions are waiting, at which point it returns Completed, or until it is stopped by a subactivity or a stop message to the activity.

stop sends a stop message to the activity if the conditions are appropriate. This message causes the control to move back up the run-stack and resume at the place in the code where the run was first executed, which doesn't mean much in a parallel execution context; but, in a serial context, it means that the next action on the super-activity will begin without finishing the rest of the current activity's actions.

next sends a next message to the activity if the conditions are appropriate. It runs an activity forward through as many actions as necessary until it hits a breakFunction, at which point it walks back up the tree of activities and returns Stopped. In most cases, this means that an entire action or action group on the activity under control will be executed, including completion of all subactivities.

step sends a step message to the activity if the conditions are appropriate. It causes the execution of a single action.

stepUntil: (timeval_t)stopTime sends a stepUntil: message to the activity if conditions are appropriate. This causes all actions on the activity's schedule, including any actions on subactivities' schedules, to be executed until the activity's relative time is equal to stopTime - 1.

(void)terminate sends the terminate message to the activity, which causes all actions and action groups to be removed from its schedule. This can cause some problems. If terminate is sent to an activity and, subsequently, a run is attempted on that activity, the program will exit with an error. A way to avoid this might be to eliminate the ActivityControl attached to the terminated activity; but, that would preclude the reuse of ActivityControllers on other activities. Ideally, a controller could be adjusted to monitor any of several activities and it could leap from one activity to another. So, this behaviour is advertised as a feature.

updateStateVar updates the ActivityControl instance variables and tests for the continued existence of the activity that's being controlled. This message is sent on each cycle of the schedule for the activity being controlled.

getStatus returns the status of the activity.

(const char *)getInstanceName This method is provided for backwards compatibility to the probe machinery. It echos the displayName resident in the object name database. When the probe machinery is upgraded to use this database, this method will become obsolete.

(CompleteVarMap.html)

Probe Class

```
//
// Probe --
//   Abstract superclass of both VarProbe and MessageProbe.
//
@protocol Probe <SwarmObject>
CREATING
- setProbedClass;
SETTING
- setSafety;     // There should be a setSafety: BOOL
- unsetSafety;   // and getSafety methods instead...
USING
- clone: aZone;
- (Class)getProbedClass;
- (const char *)getProbedType;
@end
```

A Probe is simply an object that contains pointers to an element (instance variable or message description) of another object. The Probe contains instance variables that describe the referent's class and type. It's actually an abstract class that is further subdivided into VarProbe and MessageProbe, which represent the two basic types of elements of any object. The Probes are collected into a ProbeMap and subsequently installed in the ProbeLibrary.

Create Phase Protocol

setProbedClass: (Class)aClass sets the class of the referent and must be called at create time.

Setting Phase Protocol

setSafety turns on the option of checking the compatibility of the class of the object before any actions are performed on the referent. This is only used in the VarProbes and should probably be moved.

unsetSafety turns the safety option off.

Regular Protocol

(Class)getProbedClass returns the class of the referent as a Class pointer.

VarProbe Class

```
//
// VarProbe --
// Allows the user to inspect a given variable in any candidate
// which is an instance of, or inherits from, a given class
//
@protocol VarProbe <Probe>
CREATING
- setProbedVariable: (const char *)aVariable;
SETTING
- setStringReturnType: returnType;
USING
- (const char *)getProbedVariable;
- setFloatFormat: (const char *)format;
// Currently, setNonInteractive must be used *after* create phase
// In any case these should also become setInteractive: BOOL etc.

- setNonInteractive;
- (int)isInteractive;

- (void *)probeRaw: anObject;
- (void *)probeAsPointer: anObject;
- (int)probeAsInt: anObject;
- (double)probeAsDouble: anObject;
- (const char *)probeAsString: anObject Buffer: (char *)buffer;
- (const char *)probeAsString: (id) anObject Buffer: (char *)buf withFullPre-
cision: (int)precision;
- setData: anObject To: (void *) newValue; // pass by reference.
- (int)setData: anObject ToString: (const char *)s;
@end
```

This is a specialized subclass of the abstract class *Probe*. It completes the specification of a probe whose referent is an instance variable element of an object.

Create Phase Protocol

- setProbedVariable: (const char *)aVariable sets the variable being probed (elsewhere called the referent). The *aVariable* identifier is simply a character string consisting of the identifier

- (void *)probeRaw: anObject

- (void *)probeAsPointer: anObject

- (int)probeAsInt: anObject

- (double)probeAsDouble: anObject

- (const char *)probeAsString: anObject Buffer: (char *) buffer

- (const char *)probeAsString: anObject Buffer: (char *) buffer withPrecision: (int) precision

  These are the methods which actually use the probe. *probeRaw* simply returns a pointer to the probed variable, whereas the other methods return it as a particular type. *probeAsString* simply prints the value of the variable into the buffer (in more or less the same format you would expect from a *sprintf()*). The buffer should be pre-allocated. The second form of *probeAsString* is only used internally by ObjectSaver

  (../simtools/ObjectSaver.html)

  to use the "saved as" precision form which can be different from the "displayed" precision. See the section on setting the probing precision

  (../probing.html#precision)

  for floating-point numbers. Note: The *VarProbe* will complain if you try to do weird type conversions: for example, if the probed variable is of type pointer and you try to use the *probeAsDouble* method then the return value will be undefined (and you will get an error message on *stderr*).

- setStringReturnType: returnType

  When the *probedVariable* is of type unsigned char or char, the method *probeAsString* will, by default, return a string of the format: "'%c' %d". This is meant to reflect the commonplace use of an unsigned char as a small int. However, the user can tailor the format returned by the probe to use only one of the two possible representations.

The legal values for returnType are:

- DefaultString ("'%c' %i")
- CharString ("'%c'")
- IntString ("%i").

Note: When setting the value of an unsigned char or a char using the *setData:ToString:* (method described below), the expected format of the string is always "%i" unless *CharString* was chosen (in which case the format should be "'%c'").

- setData: anObject To: (void *)newValue

- setData: anObject ToString: (const char *)s

  The programmer can set the *probedVariable* via these methods, either by providing a pointer to the new value or a string which the probe will read and convert appropriately. Note: For the time being, the kernel itself is the only software which uses probes in this fashion, so we can get away with expecting either

MessageProbe Class

```
//
// MessageProbe --
//   Allows the user to call a given message on any candidate
//   which is an instance of, or inherits from, a given class
//
@protocol MessageProbe <Probe>
CREATING
- setProbedSelector: (SEL) aSel;
- setProbedMessage: (const char *)aMessage;
- setHideResult: (int) val;
USING
-(int)isResultId;                 // I doubt that a user will
-(int)isArgumentId: (int)which;   // ever need these.

- (const char *)getProbedMessage;
- (int)getArgNum;
- (const char *)getArg: (int)which;
- (const char *)getArgName: (int)which;
- (int)getHideResult;

- setArg: (int)which To: (const char *)what;

- dynamicCallOn: target resultStorage: (const char **)result;
- dynamicCallOn: target;

- updateMethodCache: anObject;
- (int)intDynamicCallOn: target;
- (float)floatDynamicCallOn: target;
- (double)doubleDynamicCallOn: target;
@end
```

This is a specialized subclass of the abstract class Probe. It completes the specification of a probe whose referent is a message element of an object.

Create Phase Protocol

setProbedSelector: (SEL)aSelector

setProbedMessage: (const char *)aMessage

sets the message to be probed. At least one of these methods must be called.

`dynamicCallOn: target resultStorage: (const char **)result`

`(int)intDynamicCallOn: target`

`(double)doubleDynamicCallOn: target`

These are the methods used to actually generate a dynamic message call on the target object. The first method does not return a result, the second will return a dynamically malloc'ed string based representation of the result, whereas the third and fourth methods assume the user knows the exact typing of the method and would like a direct translation into the appropriate datatype.

(VarProbe.html)

(SwarmObject.html)

(ProbeMap.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Objectbase Library: Usage Guide

Overview

The objectbase library contains the most basic objects users need to design their agents and swarms. It also serves, at present, as a repository for the probe machinery, which is provided for every SwarmObject. The way the classes in this library are to be used is varied. But, basically, it is provided so that the user will have something to subclass from for her own objects and Swarms.

Example Usage of SwarmObject

The best way to explain how the library should be used is to walk through an example. So, using Heatbugs, we'll walk through the ways objectbase is used and discuss them. Since more documentation is usually better than less, I'm going to explain things at a low level so that those not familiar with Objective C will understand the discussion. If you already are familiar with Objective C, then you should skip this part.

First off, the basic elements of the Heatbugs simulation are the heatbugs, the model swarm (which bundles the heatbugs), and the observer swarm (which bundles the displays of the probes poking into the model swarm and the heatbugs). The interface files for each show what must be imported and the declaration syntax needed to subclass from SwarmObject.

We'll use `Heatbug.h` for our discussion here. The first part of the file shows the C-preprocessor imports needed:

```
#import <objectbase/SwarmObject.h>
#import <space.h>
#import "HeatSpace.h"
#import <tkobjc/Raster.h>
```

The `#import <objectbase/SwarmObject.h>;` is included in order to subclass from SwarmObject. However, to provide backwards compatibility, we've placed this import in the library interface file `objectbase.h` as well, which means one could subclass from SwarmObject by simply importing the `objectbase.h` file. This is discouraged in order to make the library interfaces as standard as possible.

The next objectbase relevant piece of code in this file is:

```
@interface Heatbug: SwarmObject
{
  double unhappiness;
  int x, y;
  HeatValue idealTemperature;
  HeatValue outputHeat;
```

The reasons for doing this are related to the object phase protocols used by defobj. If you would like more info on that, see Objective C - Swarm Style
(../../swarmObjC.html)
in the section entitled *The Create Phase In Principle*.

Finally, the `@end` keyword signifies the end of the interface definition. GNU Objective C allows one to leave this off; but, it is not good practice.

And that's it. Of course, there're a few tricky aspects to using the objectbase library that weren't mentioned here. Some of them will be mentioned in the Advanced Usage Notes
(AdvancedUsage.html)
and the Implementation Notes
(Implementation.html)
; but, the best way to learn is to examine the way the demo applications do it and try to make some changes yourself.

Subclassing from Swarm

Subclassing from the Swarm class works very similar to subclassing from SwarmObject.

ActivityControl

The ActivityControl object provides much more finely grained control over the execution of an interactive simulation. It addresses both the problems of not being able to stop the simulation at any given point in any given activity and provides an initial step towards a Swarm debugger.

An activity controller can be attached to any activity that is created in a Swarm simulation, including those that are created for use only by the Swarm kernel. The controller then provides the basic activity manipulation messages on that activity, which are: `run`, `stop`, `next`, `step`, `stepUntil`, and `terminate`.

The presence of the ActivityControl object might cause some confusion about what role the ControlPanel should play in the controlled execution of the various schedules. The ControlPanel should still be used for the top-level control of any simulation that is running in a context where random interference is expected (like under a GUISwarm where the user may click a button at any time). The reason this is true is because the ControlPanel sends pseudo-interrupts to the infinite loop we use to perpetuate execution of the top level Swarm (which can only be seen in the form of the `go` message on a GUISwarm at present). This type of control may change in the future! But, for now, it is how we monitor the control state of the simulation.

Now, having said that, the ControlPanel should no longer be used to run the simulation.

Objectbase Library: Advanced Usage Notes

1. ProbeMap design: When designing a ProbeMap for a given (subclass of) SwarmObject, inclusion of instance variables or messages defined in the super class might be desirable. The normal ProbeMap design code might look like (this code was taken from the tutorial app called "hello-world"):

```
probeMap = [CustomProbeMap createBegin: [self getZone]];
      [probeMap setProbedClass: [person class]];
      probeMap = [probeMap createEnd];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "room"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "party"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "name"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "stillhere"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "listOfFriends"
 inClass: [person class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "myColor"
 inClass: [person class]]];
      [probeLibrary setProbeMap: probeMap For: [person class]];
      [probeDisplayManager createProbeDisplayFor: person];
```

where `room`, `party`, `name`, `stillhere`, `listOfFriends`, and `myColor` are instance variables declared in the interface to the Person subclass. And Person is a subclass of Agent2d, which is a subclass of SwarmObject.

Now let's say that there are two variables declared in Agent2d that you want to put into this custom probe in addition to the ones you've picked out of Person. Call them `x` and `y`. The way to add them to the `probeMap` is to add the following two lines of code to the above.

```
[probeMap addProbe: [probeLibrary getProbeForVariable: "x"
 inClass: [Agent2d class]]];
      [probeMap addProbe: [probeLibrary getProbeForVariable: "y"
 inClass: [Agent2d class]]];
```

And that's it! The two superclass-declared variables, which are, in fact, instance variables of the instance of the subclass, are now included in the probe.

In addition, a convenience message has been added to the CustomProbeMap interface to compress the above rather cluttered mechanism into one message. This convenience message can be used in the usual case where a ProbeMap will consist of variables and messages from the same class. For example, the first part of the custom probe creation above can be shortened to:

```
[probeLibrary create: [self getZone] forClass: [person class]
      withIdentifiers: "room", "party", "name", "stillhere",
```

if one sends a `run` message to the ActivityControl that is attached to the `observerSwarm`'s activity, the entire model continues to run to completion, unless the user sends a `stop` message. However, if the sim is stopped at some point, a `run` message to the `modelSwarm`'s activity will have no effect at all. (Note: If you do this via the activity controllers, you see the `currentTime` variable get updated; but, the actual `run` message to the activity, itself, has no effect.)

So, the rule of thumb, for the present, is to attach ActivityControl objects only to "top-level" activities, like the ObserverSwarm.

(UsageGuide.html)

(../objectbase.html)

(InterfaceDesign.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Swarmobject Library: Implemenation Notes

---

1. Right now, Probe's rely on a special method, `getInstanceName`, that has to be implemented in any probe-able object in order to get anything other than the class name of that object into the object designation widget. However, a more general capability has been added to defobj to give a meaningful name to any object. Probes will be changed to take advantage of this new capability.
2. In the ActivityControl the frequency of the message `updateStateVar` is very high. It is sent at least once every cycle and every time a message is sent to the ActivityControl. This is unsatisfactory. Some of these messages can be pruned out of the object.
3. The `probeMap` designed for use with an ActivityControl was chosen fairly arbitrarily. Right now, it serves as a default for the class. A user can override it by designing a new one and inserting it into the `probeLibrary`.
4. Errors specific to objects in the objectbase library need to be gathered and initialized like those in the defobj library.

(InterfaceDesign.html)

(../objectbase.html)

(../objectbase.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

- fillWithObject: anObject

    These methods will fill the entire space with a specified value or object. The difference between the 'fill' and 'fastFill' methods is that the former use putObject and putValue internally, in other words, they are sensitive to any subclassing and alteration of these methods, whereas the latter use low level access routines
    directly and are therefore much faster (and remain unaffected by subclassing).

- copyDiscrete2d: (Discrete2d *)a toDiscrete2d: (Discrete2d *)b;

    This method copies the data in one Discrete2d object to another Discrete2d object. It assumes that both objects already exist.

- (int)setDiscrete2d: (Discrete2d *)a toFile: (const char *)filename;

    This method reads a PGM formatted file and pipes the data into a Discrete2d object.

Accessing elements

The Discrete2d is essentially a 2d array of ids. However, we also allow users to access the array as if it were an array of integers: the integers are cast into ids for storage. The Object methods retrieve values as objects, the Value methods cast things to integers before returning. We believe this is safe on all architectures.

- getObjectAtX: (int)x Y: (int)y

    Return the pointer stored at (x,y).

- getValueAtX: (int)x Y: (int)y

    Return the integer stored at (x,y).

- putObject: anObject atX: (int)x Y: (int)y

    Put the given pointer to (x,y) overwriting whatever was there.

- putValue: (int)v atX: (int)x Y: (int)y

    Put the given integer to (x,y) overwriting whatever was there.

Low level access

For speed, sometimes you want to go below the method call layer and access cells quickly. To allow this Discrete2d defines some macros that give you internal access. Use these at your own peril! In particular, subclasses have no way to redefine the macros.

- (id *)getLattice

    Returns the lattice pointer - use this for fast access.

- discrete2dSiteAt(lattice, offsets, x, y)

    Macro to return a pointer to the value at site x, y. lattice is the return value from getLattice, offsets is the precomputed offsets (stored in self->offsets).

maintained: lattice (the current state), and newLattice (the future state). All reads take place from lattice, all writes take place to newLattice. newLattice is copied to lattice when - updateLattice is called. DblBuffer2d can be used to implement one model of concurrent action, like in Ca2ds. NOTE: be very careful if you're using low-level macro access to the world, in particular be sure that you preserve the write semantics on the newLattice.

New Methods

- updateLattice
  Copy newLattice to lattice, in effect updating the lattice.
- (id *)getNewLattice
  Like - (id *)getLattice: return a pointer to the newLattice buffer.

Overridden Methods

- createEnd
  Rewrites the method from Discrete2d. Allocate two lattices, makes the offsets.
- putObject: anObject atX: (int)x Y: (int)y and - putValue: (int)value atX: (int)x Y: (int)y and
  Overridden so writes happen to newLattice.

Ca2d

Inherits from DblBuffer2d, defines abstract protocol for cellular automata.

New Methods

- setNumStates: (int)d
  Record the number of states the CA understands.
- initializeLattice
  Use this to set up your CA to a default initial state. Unimplemented in Ca2d.
- stepRule
  One iteration of the CA rule. Unimplemented in Ca2d.

Overridden Methods

- createEnd

```
newHeat = evapRate * (self + diffuseConstant*(nbdavg - self))
where nbdavg is the weighted average of the 8 neighbours.
```

Object2dDisplay

Object2dDisplay helps display 2d arrays of objects. Create a Object2dDisplay, give it a Raster widget to draw on, a Discrete2d, a message to call on each object, and (optionally) a collection of objects and it will dispatch the message to all objects with the Raster widget as an argument. In addition, Object2dDisplay can help you make probees.

Creation Phase

- setDisplayWidget: (Raster *)r

  Set the display widget to use for drawing.

- setDiscrete2dToDisplay: (Discrete2d *)c

  Set the 2d array to draw

- setDisplayMessage: (SEL)s

  Set the message to be sent to each object in the grid to make it draw itself.

- setObjectCollection: objects

  (optional) set a collection of objects to be displayed. If this is not given, then Object2dDisplay loops through the 2d grid sending draw messages to all objects it finds there. Giving an explicit collection of objects to draw is more efficient if your grid is sparsely populated.

Use

- display

  Draw all objects in the array (or optionally, the collection) on the raster widget. All that happens here is the display message is sent to each object - it is the object's responsibility to render itself.

- makeProbeAtX: (int)x Y: (int)y

  Find an object at the given (x,y) coordinate and build a probe display for it. This is a good method to make a button client for a raster widget like

```
[aRaster setButtonClient: aObject2dDisplay Message: M(makeProbeAtX:Y:)]
```

---

has been sufficient. In particular, the output of this class has been used, in conjunction with the fragstats
(ftp://ftp.fsl.orst.edu/pub/fragstats.2.0/)
package, to apply landscape metrics to the output of a Swarm simulation. Nevertheless, I (Manor Askenazi
(mailto: manor@santafe.edu)
) would be very interested in hearing from end-users about any other (presumably more complex formats), as the need arises.

- setDiscrete2dToFile: (Discrete2d *)aSpace
  Set the target space to be filled. This message can be used more than once, but often it is useful to keep one Int2dFiler per space (e.g. when the space is saved multiple times).
- setValueMessage: (SEL)aSelector
  This message is optional. It is used when the target Discrete2d contains objects. By sending each object the message specified by the selector, the Int2dFiler is able to get from the object an integer representing its state, which it then writes to the file.
- setBackground: (int)aValue
  This message is optional. It is used when the target Discrete2d contains objects. If a particular location in the space has no resident object, the argument of this message is the value which gets writtent to the file. The default background value is 0.
- fileTo: (const char *)fileName
  When the Int2dFiler receives this message, it opens a file called *fileName*, stores the state of a pre-specified space into it, and then closes the file.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1997-12-17

For best results, library users should test these generators themselves in some domain-specific way. One easy way to do this is to run an experiment twice: once with one class of generator (say, PMMLCG), and once with another (say, SWB). If the results differ radically, then you can suspect the generator. If they don't, well, the generator still might be faulty.

There is also a tarball of test programs for the random number library on the Swarm ftp site:
SwarmTests-0.7.tar.gz
(ftp://ftp2.santafe.edu/pub/swarm/RandomTests-0.7.tar.gz)

Documentation

The *random* library is defined and documented according to the Library Interface Conventions

(defobj/library.html)
. The following standard sections of documentation are available:

- Interface Reference
(random/Interface.html)
- Usage Guide
(random/UsageGuide.html)
- Advanced Usage Guide
(random/AdvancedUsage.html)
- Interface Design Notes
(random/InterfaceDesign.html)
- Implementation Notes
(random/Implementation.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Literature references for implemented objects

doc.generators.table
: Summary descriptive data for generators

doc.generators.maxvalue
: Values returned by [myGen getUnsignedSample]

doc.quality.generators
: Results of statistical and timing tests

doc.timings
: Timing results for generators and distributions

methods.SimpleGenerators.h
: Methods implemented by a typical Simple generator

methods.SplitGenerators.h
: Methods implemented by a typical Split generator

Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin. --Jon von Neuman

---

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Random Library: Interface Reference

The *random* library uses the module definition conventions
(../defobj/module.html)
established by the *defobj* library
(../defobj.html)
. It also depends on standard supertypes and classes defined by this library.

Dependencies

Following are the other header files imported by <random.h>:

```
#import <defobj.h>
#import <random/generators.h>
#import <random/distributions.h>
#import <random/RandomDefs.h>
#import <random/RandomVars.h>
```

The *defobj* library interface is included to provide the basic object support. `RandomDefs.h` contains some C preprocessor macros and typedefs used in the library.

Object Types (with links to reference pages)

Contents:

The following shows the protocol inheritance trees and object definitions for generators and distributions. The complete set of methods implemented by these objects have been subdivided into sub-protocols of methods that belong together logically. Just click on a (sub-)protocol name to see what methods it implements. (You may want to review the section on Protocols in the Objective-C book here!)

In the list below, any line that contains the word CREATABLE defines an object that you can use in your program. (This is part of the Swarm DefObj machinery.) In other words, while 'InternalState' is a normal protocol (a list of method definitions), the name 'ACGgen' refers to both a protocol and a class that implements that protocol. (You create an ACG generator by saying myGen = [ACGgen create: ....]). Similarly, 'GammaDist' defines both a protocol and a class that implements that protocol.

All generators and distributions ultimately inherit from SwarmObject.

(Prot.html#SingleLongSplitGen)
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

- CombinedShortSplitGenerator
(Prot.html#CombinedShortSplitGen)
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>
- CombinedLongSplitGenerator
(Prot.html#CombinedLongSplitGen)
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>
- ACGgen
(ACGgen.html)
<SingleLongGenerator, CREATABLE>
- C2LCGXgen
(C2LCGXgen.html)
<CombinedShortSplitGenerator, CREATABLE>
- C2MRG3gen
(C2MRG3gen.html)
<CombinedLongGenerator, CREATABLE>
- C2TAUSgen
(C2TAUS1-3gen.html)
<CombinedShortGenerator, CREATABLE>
  - C2TAUS1gen
  (C2TAUS1-3gen.html#C2TAUS1gen)
  : C2TAUSgen <C2TAUSgen, CREATABLE>
  - C2TAUS2gen
  (C2TAUS1-3gen.html#C2TAUS2gen)
  : C2TAUSgen <C2TAUSgen, CREATABLE>
  - C2TAUS3gen
  (C2TAUS1-3gen.html#C2TAUS3gen)
  : C2TAUSgen <C2TAUSgen, CREATABLE>
- C3MWCgen
(C3MWCgen.html)
<CombinedLongGenerator, CREATABLE>
- C4LCGXgen
(C4LCGXgen.html)
<CombinedShortSplitGenerator, CREATABLE>
- LCGgen
(LCG1-3gen.html)
<SingleShortGenerator>
  - LCG1gen
  (LCG1-3gen.html#LCG1gen)
  : LCGgen <LCGgen, CREATABLE>

  (PMMLCG1-9gen.html#PMMLCG4gen)
  : PMMLCGgen <PMMLCGgen, CREATABLE>

  - PMMLCG5gen
    (PMMLCG1-9gen.html#PMMLCG5gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG6gen
    (PMMLCG1-9gen.html#PMMLCG6gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG7gen
    (PMMLCG1-9gen.html#PMMLCG7gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG8gen
    (PMMLCG1-9gen.html#PMMLCG8gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG9gen
    (PMMLCG1-9gen.html#PMMLCG9gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>

- PSWBgen
  (PSWBgen.html)
  <SingleLongGenerator, CREATABLE>
- RWC2gen
  (RWC28gen.html)
  <SingleLongGenerator, CREATABLE>
- RWC8gen
  (RWC28gen.html#RWC8gen)
  <CombinedLongGenerator, CREATABLE>
- SCGgen
  (SCGgen.html)
  <SingleLongGenerator, CREATABLE>
- SWBgen
  (SWB1-3gen.html)
  <SingleLongGenerator>
  - SWB1gen
    (SWB1-3gen.html#SWB1gen)
    : SWBgen <SWBgen, CREATABLE>
  - SWB2gen
    (SWB1-3gen.html#SWB2gen)
    : SWBgen <SWBgen, CREATABLE>
  - SWB3gen
    (SWB1-3gen.html#SWB3gen)
    : SWBgen <SWBgen, CREATABLE>

- UniformIntegerDist
(UniformIntegerDist.html)
<IntegerDistribution, CREATABLE>
- UniformUnsignedDist
(UniformUnsignedDist.html)
<UnsignedDistribution, CREATABLE>
- UniformDoubleDist
(UniformDoubleDist.html)
<DoubleDistribution, CREATABLE>
- ExponentialDist
(ExponentialDist.html)
<DoubleDistribution, CREATABLE>
- GammaDist
(GammaDist.html)
<DoubleDistribution, CREATABLE>
- NormalDist
(NormalDist.html)
<Normal, CREATABLE>
- LogNormalDist
(NormalDist.html#LogNormalDist)
<Normal, CREATABLE>

(../random.html)

(../random.html)

(UsageGuide.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

(void) -getState: (void *) state writes a copy of the state of the object into the memory address pointed to by the "state" pointer passed to it. The memory should be allocated before the message is sent.

(void) -setState: (void *) state copies the values contained in the memory address pointed to by "state" into the state variables used by the object. The block of memory containing the state may be sent back to the heap after the message has been sent.

(./Interface.html)

(./Interface.html)

These descriptions are not finished. Please see the documentation in $(SWARMHOME)/src/random/docs/

Generator Protocols

SingleSeed

```
@deftype SingleSeed
CREATING
+  create: aZone setStateFromSeed: (unsigned) seed;
USING
- setStateFromSeed:  (unsigned)   seed;
- (unsigned) getMaxSeedValue;
- (unsigned)  getInitialSeed;
@end
```

MultiSeed

```
@deftype MultiSeed
CREATING
+ create: aZone setStateFromSeeds: (unsigned *) seeds;
```

```
   setw: (unsigned) w // log2(segment length)
setStateFromSeeds: (unsigned *) seeds;
USING
- setStateFromSeeds: (unsigned *) seeds;
- (unsigned)  lengthOfSeedVector;
- (unsigned *) getMaxSeedValues; // min is 1
- (unsigned *) getInitialSeeds; // = getInitialSeeds: 0
@end
```

Split

```
@deftype Split
CREATING
+ createWithDefaults: aZone;
USING
- setAntithetic: (BOOL) antiT;
- (unsigned)    getNumGenerators; // returns A
- (unsigned)    getNumSegments; // returns v
- (unsigned)    getSegmentLength; // returns w
- (BOOL) getAntithetic;
- initGenerator:    (unsigned) vGen; // to segment #0
- restartGenerator: (unsigned) vGen; // start of current segment
- advanceGenerator: (unsigned) vGen; // to next segment
- jumpGenerator:    (unsigned) vGen  toSegment: (unsigned long long int) seg;
- initAll; // to segment #0
- restartAll; // start of current segment
- advanceAll; // to next segment
- jumpAllToSegment: (unsigned long long int) seg;
- (unsigned long long int) getCurrentCount:   (unsigned) vGen;
- (unsigned long long int) getCurrentSegment: (unsigned) vGen;
@end
```

SplitOut

```
@deftype SplitOut
USING
- (unsigned)    getUnsignedMax;
- (unsigned) getUnsignedSample:   (unsigned) vGen;
- (float)       getFloatSample:      (unsigned) vGen; // using 1 unsigned
- (double)      getThinDoubleSample: (unsigned) vGen; // using 1 unsigned
- (double)      getDoubleSample:     (unsigned) vGen; // using 2 unsigneds
- (long double) getLongDoubleSample: (unsigned) vGen; // using 2 unsigneds
@end
```

SingleShortGenerator
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed>

No extra messages added.

SingleLongGenerator
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed, MultiSeed>

```
- setGenerator: (id) generator;
- setGenerator: (id) generator
setVirtualGenerator: (unsigned) vGen;
- createEnd;
- reset;
-               getGenerator;
- (unsigned) getVirtualGenerator;
- (BOOL) getOptionsInitialized;
- (unsigned long long int) getCurrentCount;
@end
```

BooleanDistribution
<ProbabilityDistribution>

```
//
// BooleanDistribution --
//   A probability distribution that returns YES/NO sample values.
//
@deftype BooleanDistribution <ProbabilityDistribution>
- (BOOL) getBooleanSample;
- (int) getIntegerSample; // for convenience
@end
```

IntegerDistribution
<ProbabilityDistribution>

```
//
// IntegerDistribution --
//   A probability distribution that returns integer sample values.
//
@deftype IntegerDistribution <ProbabilityDistribution>
- (int) getIntegerSample;
@end
```

UnsignedDistribution
<ProbabilityDistribution>

```
//
// UnsignedDistribution --
//   A probability distribution that returns non-
negative integer sample values.
//
@deftype UnsignedDistribution <ProbabilityDistribution>
- (unsigned) getUnsignedSample;
@end
```

DoubleDistribution
<ProbabilityDistribution>

```
//
// DoubleDistribution --
//   A probability distribution that returns an approximation of continuous
```

ACGgen Class

```
//
// ACGgen--
//   predefined random number generators, not recommended for serious use;
//   included for historical reasons (compatibility with earlier releases).
//
@protocol ACGgen      <SingleLongGenerator, CREATABLE> @end
```

Use of this generator is NOT Recommended

Documentation

```
Generator name:  | ACG

Description:  | Additive Congruential Generator

Reference:  | ACG is an often used algorithm, mentioned in Knuth,
| Numerical Recipes, etc.

| state vector = f(seed)
Algorithm:  | state_n = (state_(n-s) + state_(n-r)) mod 2^32; r > s
| output = state_n

| ACG is in the Lagged Fibonacci class of generators.
| These generators use a basic algorithm of the form
| X_n = f(X_(n-r),X_(n-s)) mod m; r>s
| The function f is typically xor, addition, subtraction,
| multiplication or subtraction with carry. It uses
| simpler math than a basic LCG, but keeps a larger state.

Comments:  | This generator relies on controlled overflow at 32 bits.
| This requires that unsigned be a 32bit value that follows
| ANSI C rules.

Implementation:  | The state vector contains the generator's r most recent
| outputs (X_n .. X_(n-r)). This vector is initialized from
| an LCG generator using one 32-bit seed value. After
| initialization we throw away r*5 numbers to 'prime' the
| generator (i.e. make sure we have entered a closed cycle.)
```

Reference for output quality:	Knuth, op.cit.

(./Interface.html)

(./Interface.html)

(./C2LCGXgen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
Comments:  | For a discussion of PMMLCG generators, see the source file
| PMMLCGgen.h.

Configuration | unsigned int numGenerators; // # virtual generators
parameters:  | unsigned int numSegments; // log2(#segments/vGen)
| unsigned int segmentLength; // log2(segment length)

State:  | unsigned int state [numGenerators] [2];
| (each virtual generator has a 4-integer state vector)

Output types | unsigned int in [0,unsignedMax].
(for each | float in [0.0,1.0) using 1 iteration of the generator.
virtual | double in [0.0,1.0) using 1 or 2 iterations of the generator.
generator):  | long double in [0.0,1.0) using 2 iterations of the generator.

Output range:  | The combined generator yields output in the range [0,m0-1]
| (i.e. [0,2147483562].)

Limits on | Number of generators A <= 2^25  (33,554,432)
parameters:  | (You'll likely run out of memory before then ...)
| Number of segments / generator <= 2^60 (i.e. v <= 60)
| Length of each segment <= 2^60 (i.e. w <= 60)

Defaults:  | A=32, v=20, w=30

Valid seeds:  | Using create:setStateFromSeed: in [1, 2^32-1]
| Using create:setStateFromSeeds:
| The generator must be seeded with 2 starting seeds:
|  Seed[0]: [1, 2147483562] = [1,m1-1]
|  Seed[1]: [1, 2147483398] = [1,m2-1]

Cycles:  | With properly chosen parameters, the component generators
| have a single full cycle of length (m-2).
| The combined generator has a cycle length of
| (m1-1)*(m2-1)/2 = 2147483562*2147483398/2 ~= 2^61.

Output Quality:  | All bits should be safe.
| The combined generator has much better statistical
| properties than either of its component PMMLCG's.

Reference for | L'Ecuyer and Cote, op. cit.
output quality:  |

Implementation  | The code for -getUnsignedMax uses a trick devised by
comment:        | Schrage to avoid 32-bit overflow when calculating the
| next state (i.e. using q=m/a and r=m%a as auxiliary
| values). Using 64-bit math (long long integers) is more
```

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,unsignedMax] = [0,m1-2].

Valid seeds:  | This generator requires 3 seeds for each component:
| within [1,m1-1] for component 1;
| within [1,m2-1] for component 2.
| Using create:setStateFromSeed: in [1,2^32-1]

Cycles:  | With properly chosen parameters, this generator
| has a single cycle of length (m1^3-1)(m2^3-1)/2
| or 2^184 < cycle < 2^185 (4.9e55).

Output Quality:  | Good.

Reference for | L'Ecuyer et al, op. cit.
output quality:  |

C2MRG3gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       13.459 uS
|   -getFloatSample, getDoubleSample:         15.203 uS
|   -getFatDoubleSample, getLongDoubleSample: 29.699 uS

Relative speed:
| Speed 0.275 (time 3.640) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--

Parameters:  | unsigned int k = 3 // lag order

| unsigned int m1 = 2^31 - 1 // modulus
| unsigned int a12 = 63308 // multiplier
| unsigned int a13 = -183326 // multiplier

| unsigned int m2 = 2145483479 // modulus
| unsigned int a21 = 86098 // multiplier
| unsigned int a23 = -539608 // multiplier

Reference for | L'Ecuyer et al, op. cit., Table 3.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length
| ((2^31-1)^3 - 1) * (2145483479^3 - 1) / 2 ~= 4.89e55,
| i.e. 2^184 < cycle < 2^185.
| (The paper, in apparent error, claims
| that the cycle is close to 2^205.)
```

C2TAUSgen Class

```
//
// C2TAUS1, C2TAUS2, C2TAUS3 --
//   short component based generator recommended for use.
//
@deftype C2TAUSgen    <CombinedShortGenerator> @end
@deftype C2TAUS1gen   <C2TAUSgen, CREATABLE> @end
@deftype C2TAUS2gen   <C2TAUSgen, CREATABLE> @end
@deftype C2TAUS3gen   <C2TAUSgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | C2TAUS

Description:  | Combined Tausworthe Generator
| based on 2 component generators of periods 2^31-1 and 2^29-1

Reference:  | Shu Tezuka and Pierre L'Ecuyer,
| "Efficient and Portable Combined Tausworthe
| Random Number Generators."
| ACM Transactions on Modeling and Computer Simulation,
| vol. 1 no. 2, April 1991, pp. 99-112.

| Original reference for the Tausworthe algorithm:
| R. C. Tausworthe, "Random Numbers Generated by Linear
| Recurrence modulo 2", Math. Comput. 19 (1965), 201-209.

Algorithm:  | state = seed, then:

(Each | b = ((old << Q) ^ old) & mask
component:)  | new = ((old << S) ^ (b >> (P-S)) & mask

(Combination:)  | output = ((new1 ^ (new2 << (P1-P2))

Implementation:  |

Parameters:  | int   P [2] // word length
| int   Q [2] //
| int   S [2] //
| int PmS [2] // P - S
| int P1mP2 // P[0] - P[1]
```

```
| Q =  2

Reference for | These values recommended by Tezuka and L'Ecuyer,
parameters:  | op. cit., table I p. 107.

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length ~ 2^60.

Output Quality:  | Good.

Reference for | Tezuka and L'Ecuyer, op.cit.
output quality:  |
```

C2TAUS2gen Documentation

```
Parameters:  | P = 31 so Mask = 2^31-1
(component 1) | S = 21 so PmS = 10
| Q =  3

| P = 29 so Mask = 2^29-1
(component 2) | S = 17 so PmS = 12
| Q =  2

Reference for | These values recommended by Tezuka and L'Ecuyer,
parameters:  | op. cit., table II p. 107.

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length ~ 2^60.

Output Quality:  | Good.

Reference for | Tezuka and L'Ecuyer, op.cit.
output quality:  |
```

C2TAUS3gen Documentation

```
Parameters:  | P = 31 so Mask = 2^31-1
(component 1) | S = 13 so PmS = 18
| Q = 13

| P = 29 so Mask = 2^29-1
(component 2) | S = 20 so PmS =  9
| Q =  2

Reference for | These values recommended by Tezuka and L'Ecuyer,
```

C3MWCgen Class

```
//
// C3MWC --
//   long component based generator recommended for use.
//
@deftype C3MWCgen   <CombinedLongGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | C3MWC

Description:  | Combined Multiply-With-Carry generator

Reference:  | Article posted to Usenet in 1994 by George Marsaglia,
| "The Mother of all Random Number Generators."
| Source found at <ftp://ftp.taygeta.com/pub/c/mother.c>
| Also document /papers/mwc1.ps on his Diehard CD-ROM,
| found at <http://www.hku.hk/internet/randomCD.html>.

Algorithm:  | K = a * (K & 65535) + (K>>16);
| J = b * (J & 65535) + (J>>16);
| I = c * (I & 65535) + (I>>16);
| L = d * (L & 65535) + (L>>16);
| M = e * (M & 65535) + (M>>16);
| N = f * (N & 65535) + (N>>16);
| return ((K+I+M)>>16) + (J+L+N);

Comments:  | This generator is a combination of 3 MWC generators,
| each of which is a combination of 2 16-bit
| Multiply-With-Carry generators.

Parameters:  | unsigned int a = 30903 // multiplier
| unsigned int b = 18000
| unsigned int c = 29013
| unsigned int d = 30345
| unsigned int e = 30903
| unsigned int f = 31083

State:  | unsigned int K,J,I,L,M,N // state vector

Output types:  | unsigned int (we return the current state).
```

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
Comments:  | For a discussion of PMMLCG generators, see the source file
| PMMLCGgen.h.

Configuration | unsigned int numGenerators; // # virtual generators
parameters:  | unsigned int numSegments; // log2(#segments/vGen)
| unsigned int segmentLength; // log2(segment length)

State:  | unsigned int state [numGenerators] [4];
| (each virtual generator has a 4-integer state vector)

Output types | unsigned int in [0,unsignedMax].
(for each | float in [0.0,1.0) using 1 iteration of the generator.
virtual | double in [0.0,1.0) using 1 or 2 iterations of the generator.
generator):  | long double in [0.0,1.0) using 2 iterations of the generator.

Output range:  | The combined generator yields output in the range [0,m0-2]
| (i.e. [0,2147483645].)

Limits on | Number of generators A <= 2^25  (33,554,432)
parameters:  | (You'll likely run out of memory before then ...)
| Number of segments / generator <= 2^63 (i.e. v<=63)
| Length of each segment <= 2^63 (i.e. w<=63)

Defaults:  | A=128, v=31, w=41

Valid seeds:  | Using create:setStateFromSeed: in [1, 2^32-1]
| Using create:setStateFromSeeds:
| The generator must be seeded with 4 starting seeds:
|  Seed[0]: [1, 2147483646] = [1,m1-1]
|  Seed[1]: [1, 2147483542] = [1,m2-1]
|  Seed[2]: [1, 2147483422] = [1,m3-1]
|  Seed[3]: [1, 2147483322] = [1,m4-1]

Cycles:  | With properly chosen parameters, the component generators
| have a single full cycle of length (m-1).
| The combined generator has a cycle length of
| 2147483646*2147483542*2147483422*2147483322/8 > 2^120.

Output Quality:  | All bits should be safe.
| The combined generator has much better statistical
| properties than either of its component PMMLCG's.

Reference for | L'Ecuyer and Andres, op. cit.
output quality:  |

Implementation  | The code for -getUnsignedMax uses a trick devised by
```

(./LCG1-3gen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
State:  | unsigned int state // a "state vector" of length 1

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,2^32-1] may be returned
| (i.e. [0,4294967295].)

Valid seeds:  | Using create:setStateFromSeed:  [1,2^32-1]
| Using create:setStateFromSeeds: [1,2^32-1]

Cycles:  | With properly chosen parameters, these generators
| have a single full cycle of length (2^32).

Output Quality:  | Low order bits are poor.

Reference for | Knuth, op.cit.
output quality:  |

LCGgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       2.564 uS
|   -getFloatSample, getDoubleSample:         4.295 uS
|   -getFatDoubleSample, getLongDoubleSample: 7.734 uS

Relative speed:
| Speed 1.442 (time 0.693) relative to MT19937 getUnsignedSample
```

LCG1gen Documentation

```
Parameters:  | a =     1,664,525
| c = 1,013,904,223

Reference for | The multiplier is from Knuth vol II (line 26 of table).
parameters:  | Adder is from Numerical Recipes, in turn from H.W.Lewis

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length m.

Output Quality:  | Low-order bits are poor.

Reference for | Knuth, op.cit. and Numerical Recipes.
output quality:  |
```

(./MRG5-7gen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,unsignedMax] = [0,m-2].

Valid seeds:  | Using create:setStateFromSeed:  [1, 2^32-1]
| Using create:setStateFromSeeds: [1, unsignedMax]
| (unsignedMax differs between MRG5, MRG6, MRG7)

Cycles:  | With properly chosen parameters, these generators
| have a single cycle of length m^k - 1.
| Thus, for m ~= 2^31 we have:
| k=5 => cycle ~= 2^155,
| k=6 => cycle ~= 2^186,
| k=7 => cycle ~= 2^217.

Output Quality:  | Good.

Reference for | L'Ecuyer et al, op. cit.
output quality:  |

MRG5gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        9.674 uS
|   -getFloatSample, getDoubleSample:         11.524 uS
|   -getFatDoubleSample, getLongDoubleSample: 22.135 uS

Relative speed:
| Speed 0.382 (time 2.616) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--

MRG6gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       10.449 uS
|   -getFloatSample, getDoubleSample:         12.337 uS
|   -getFatDoubleSample, getLongDoubleSample: 27.735 uS

Relative speed:
| Speed 0.354 (time 2.826) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--

MRG7gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       10.913 uS
```

MRG7gen Documentation

```
Parameters:  | unsigned int k = 7 // lag order
| unsigned int m = 2^31 - 19 // modulus
| unsigned int a1 = 1071064 // multiplier
| unsigned int a7 = 2113664 // multiplier

Reference for | L'Ecuyer et al, op. cit., Table 3.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (2^31-1)^7 - 1,
| i.e. 2^216 < cycle < 2^217.

Output Quality:  | Good. Parameters chosen on the basis of theory
| and empirical testing.

Reference for | L'Ecuyer et al, op. cit.
output quality:  |
```

(./LCG1-3gen.html)

(./Interface.html)

(./MT19937gen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
Output range:  | Any number in the range [0,2^32] may be returned

Valid seeds:  | Using create:setStateFromSeed:  in [1,2^32-1]
| Using create:setStateFromSeeds: in [1,2^32-1].

Cycles:  | This generator has a single cycle of length 2^19937-1.

Output Quality:  | Excellent equidistribution properties in all dimensions
| up to 623 are claimed.

Reference for | Matsumoto and Nishimura, op. cit.
output quality:  | See also pLab Report #07 on the pLab web site.

MT19937gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        3.698 uS
|   -getFloatSample, getDoubleSample:          5.502 uS
|   -getFatDoubleSample, getLongDoubleSample: 10.159 uS

Relative speed:
| Speed 1.0 (time 1.0) relative to itself (it's the benchmark)

--------------- . -----------------------------------------------------------
--

Parameters:  | unsigned int w=32 // word length
| unsigned int N=624 // size of state array
| unsigned int M=397 // lag
| unsigned int s=7 // 'magic vector'
| unsigned int t=15 //    "     "
| unsigned int u=11 //    "     "
| unsigned int l=18 //    "     "
| unsigned int b=0x102d1200 //    "     "
| unsigned int c=0x66e50000 //    "     "
| unsigned int a=0x6b5eccf6 //    "     "

Reference for | Matsumoto and Nishimura, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length 2^19937 -1,
| or 10^6001 !!! (The longest period yet.)

Output Quality:  | Good. Parameters chosen on the basis of theory
| and empirical testing.

Reference for | Matsumoto and Nishimura, op. cit.
output quality:  |
```

MWCA-Bgen Classes

```
//
// MWC --
//   two long generators recommended for use.
//
@deftype MWCAgen    <SingleLongGenerator, CREATABLE> @end
@deftype MWCBgen    <SingleLongGenerator, CREATABLE> @end
```

MWCAgen Documentation

```
Generator name:  | MWCA

Description:  | Multiply-With-Carry generator

Reference:  | Article posted to Usenet in 1994 by George Marsaglia,
| "The Mother of all Random Number Generators."
| Source found at <ftp://ftp.taygeta.com/pub/c/mother.c>
| Also document /papers/mwc1.ps on his Diehard CD-ROM,
| found at <http://www.hku.hk/internet/randomCD.html>.

Algorithm:  | K = a * (K & 65535) + (K>>16);
| J = b * (J & 65535) + (J>>16);
| return (K<<16) + J;

Comments:  | This generator is a combination of 2 16-bit
| Multiply-With-Carry generators.

Implementation:  |

Parameters:  | unsigned int a = 30903 // multiplier
| unsigned int b = 18000 // multiplier

State:  | unsigned int K,J // state vector

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,unsignedMax] = [0,2^32-1].
```

```
| It also passes all tests and is fast.

Parameters:  | unsigned int a = 30903 // multiplier
| unsigned int b = 18000 // multiplier

State:  | unsigned int K,J // state vector

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,unsignedMax] = [0,2^32-1].

Valid seeds:  | Using create:setStateFromSeed:  [1, 2^32-1]
| Using create:setStateFromSeeds: each component may have
| a 'state' <= 65535 and a 'carry' <= (multiplier-1). Thus
| the first component has a maximum seed of
| ( (a-1) << 16) + 65535.

Cycles:  | This generator is claimed to be strictly periodic, with a
| period > 2^59. (There's possibly two such cycles.)

Output Quality:  | Good.

Reference for | Marsaglia, op. cit.
output quality:  |

MWCgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                      3.399 uS
|   -getFloatSample, getDoubleSample:         5.131 uS
|   -getFatDoubleSample, getLongDoubleSample: 9.447 uS

Relative speed:
| Speed 1.088 (time 0.919) relative to MT19937 getUnsignedSample
```

(./MT19937gen.html)

PMMLCG 1-9 Classes

```
//
// PMMLCG1, PMMLCG2, PMMLCG3 --
//   single short random number generators recommended for use.
//
// PMMLCG4, PMMLCG5, PMMLCG6, PMMLCG7 --
//   component generators of C4LCGX
//
// PMMLCG8, PMMLCG9 --
//   component generators of C2LCGX
//
@deftype PMMLCGgen   <SingleShortGenerator> @end
@deftype PMMLCG1gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG2gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG3gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG4gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG5gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG6gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG7gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG8gen  <PMMLCGgen, CREATABLE> @end
@deftype PMMLCG9gen  <PMMLCGgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | PMMLCG

Description:  | Prime Modulus Multiplicative Linear Congruential Generator

Reference:  | Stephen K. Park and Keith W. Miller
| (CACM 31:10, October 1988, 1192-1201),
| "Random Number Generators: Good Ones Are Hard To Find."
| The authors refer to this generator as a "minimally
| acceptable" or "minimally sufficient" generator.

| state = seed
Algorithm:  | newstate = (a * oldstate) mod m
| output = state-1

Comments:  | The modulus m must be a prime for this class of generator
| (as opposed to the more common power of two modulus).
| For these generators we use m=2^31-1 (2,147,483,647).
```

--------------- | --

PMMLCG1gen Documentation

```
Parameters:  | a =        16,807
| m = 2,147,483,647 (2^31-1)

Reference for | Park and Miller, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (m-1).

Output Quality:  | All bits should be safe.
| These parameters have been extensively tested by the authors.

Reference for | Park and Miller, op. cit.
output quality:  |
```

PMMLCG2gen Documentation

```
Parameters:  | a =        48,271
| m = 2,147,483,647 (2^31-1)

Reference for | Park and Miller, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (m-1).

Output Quality:  | All bits should be safe.
| These parameters are not as well tested as PMMLCG1,
| but there's reason to believe they may be better.

Reference for | Park and Miller, op. cit.
output quality:  |
```

PMMLCG3gen Documentation

```
Parameters:  | a =        69,621
| m = 2,147,483,647 (2^31-1)

Reference for | Park and Miller, op. cit.
```

```
Output Quality:  | All bits should be safe.

Reference for | L'Ecuyer and Andres, op.cit.
output quality:  |
```

PMMLCG6gen Documentation

```
Parameters:  | a =       138,556
| m = 2,147,483,423 (2^31-225)

Reference for | Pierre L'Ecuyer and Terry H. Andres,
choice of  | "A Random Number Generator Based on the Combination
parameters:  | of Four LCGs", unpubl.

| This is one of the component generators of their CLCG4.

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (m-1).

Output Quality:  | All bits should be safe.

Reference for | L'Ecuyer and Andres, op.cit.
output quality:  |
```

PMMLCG7gen Documentation

```
Parameters:  | a =        49,689
| m = 2,147,483,323 (2^31-325)

Reference for | Pierre L'Ecuyer and Terry H. Andres,
choice of  | "A Random Number Generator Based on the Combination
parameters:  | of Four LCGs", unpubl.

| This is one of the component generators of their CLCG4.

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length (m-1).

Output Quality:  | All bits should be safe.

Reference for | L'Ecuyer and Andres, op.cit.
output quality:  |
```

(./Interface.html)

(./PSWBgen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
| unsigned int m // modulus

State:  | unsigned int state[r] // state vector
| int carry // carry bit

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,m-1] may be returned.

Valid seeds:  | Using create:setStateFromSeed:  [1,2^32-1]
| Using create:setStateFromSeeds: r seeds in [1,2^32-7]
| and one seed in [0,1] (the carry)

Cycles:  | With properly chosen parameters, this generator
| has a single cycle of length close to 10^414.

| Unfortunately, there are no 'jump-ahead' or 'splitting'
| facilities that would allow us to jump ahead in the cycle,
| other than calling the generator a number of times.

Output Quality:
| Much better than other Lagged Fibonacci generators, such as
| ACG and SCG.

Reference for | Marsaglia and Zaman, op.cit.
output quality:  |

PSWBgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       3.452 uS
|   -getFloatSample, getDoubleSample:         5.166 uS
|   -getFatDoubleSample, getLongDoubleSample: 9.544 uS

Relative speed:
| Speed 1.070 (time 0.933) relative to MT19937 getUnsignedSample
--------------- | -----------------------------------------------------------
---

Parameters:  | r = 43 // long lag
| s = 22 // short lag
| m = 2^32 - 5 // modulus (prime)

Reference for | Marsaglia, op. cit.
parameters:  |
```

RWC2,8gen Classes

```
//
// RWC2 --
// RWC8 ("Mother") --
//   single long generator recommended for use.
//
@deftype RWC2gen   <SingleLongGenerator, CREATABLE> @end
@deftype RWC8gen   <CombinedLongGenerator, CREATABLE> @end
```

RWC2gen Documentation

```
Generator name:  | RWC2

Description:  | 2-lag Recursion-With-Carry generator

Reference:  | Article posted to Usenet in 1994 by George Marsaglia,
| "The Mother of all Random Number Generators."
| Source found at <ftp://ftp.taygeta.com/pub/c/mother.c>
| Also document /papers/rwc1.ps on his Diehard CD-ROM,
| found at <http://www.hku.hk/internet/randomCD.html>.

Algorithm:  | x[n]  = a*(x[n-1]+x[n-2]) + carry % 2^32
| carry = a*(x[n-1]+x[n-2]) + carry / 2^32

Comments:  | This generator is a 2-lag MWC generator

Implementation:  | This generator is implemented using 64-bit math

Parameters:  | unsigned int a = 1,111,111,464 // multiplier

State:  | unsigned int carry,x1,x2 // state vector

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,unsignedMax] = [0,2^32-1].

Valid seeds:  | Using create:setStateFromSeed:  [1, 2^32-1]
```

```
State:  | unsigned short m1[10], m2[10] // state vector

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,unsignedMax] = [0,2^32-1].

Valid seeds:  | Using create:setStateFromSeed:  [1, 2^32-1]
| Using create:setStateFromSeeds:
|   a total of 18 16-bit seeds are needed;
|   each xi may be in [1, 2^31-1].
|   each carry may be in [1, sum(multipliers)-1].

Cycles:  | This generator is claimed to have a period > 2^250.
| It may have rho sequences up to length r = 8.
| It may have multiple cycles of the stated period length.

Output Quality:  | Claimed to be good.

Reference for | Marsaglia, op. cit.
output quality:  |

RWC8gen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       14.649 uS
|   -getFloatSample, getDoubleSample:         16.400 uS
|   -getFatDoubleSample, getLongDoubleSample: 31.915 uS

Relative speed:
| Speed 0.252 (time 3.961) relative to MT19937 getUnsignedSample
--------------- . ------------------------------------------------------------
--
```

(./PSWBgen.html)

(./Interface.html)

SCGgen Class

```
//
// SCG --
//   single long random number generator,
//   NOT recommended for serious use; included for
//   historical reasons (compatibility with earlier releases).
//
@deftype SCGgen     <SingleLongGenerator, CREATABLE> @end
```

Use of this generator is NOT Recommended

Documentation

```
Generator name:  | SCG

Description:  | Subtractive Congruential Generator

Reference:  | SCG is an often used algorithm, mentioned in Knuth,
| Numerical Recipes, etc.

| state vector = f(seed)
Algorithm:  | state_n = (state_(n-r) - state_(n-s)) mod 10^9; r > s
| output = state_n

| SCG is in the Lagged Fibonacci class of generators.
| These generators use a basic algorithm of the form
| X_n = f(X_(n-r),X_(n-s)) mod m; r>s
| The function f is typically xor, addition, subtraction,
| multiplication or subtraction with carry. It uses
| simpler math than a basic LCG, but keeps a larger state.

Implementation:  | The state vector contains the generator's r most recent
| outputs (X_n .. X_(n-r)). This vector is initialized from
| an LCG generator using one 32-bit seed value. After
| initialization we throw away r*5 numbers to 'prime' the
| generator (i.e. make sure we have entered a closed cycle.)

Parameters:  | unsigned int r // long lag
```

Output Quality:	Low-order bits are poor.
Reference for output quality:	Knuth, op.cit.

(./RWC28gen.html)

(./Interface.html)

(./SWB1-3gen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
Parameters:  | unsigned int r // long lag
| unsigned int s // short lag

State:  | unsigned int state[r] // state vector
| int carry // carry bit

Output types:  | unsigned int (we return the current state).
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,2^32-1] may be returned
| (i.e. [0,4294967295].)

Valid seeds:  | Using create:setStateFromSeed:  [1,2^32-1]
| Using create:setStateFromSeeds: r seeds in [1,2^32-2]
| and one seed in [0,1] (the carry)

Cycles:  | With properly chosen parameters, these generators
| have a number of long cycles, up to 10^354 in length.
| UNFORTUNATELY there is no way to pick different
| cycles, nor any easy way to 'jump ahead' in the
| chosen cycle (other than calling -getUnsignedSample
| a number of times).

Output Quality:
| Much better than other Lagged Fibonacci generators, such as
| ACG and SCG.

Reference for | Marsaglia and Zaman, op.cit.
output quality:  |

SWBgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       3.285 uS
|   -getFloatSample, getDoubleSample:         5.057 uS
|   -getFatDoubleSample, getLongDoubleSample: 9.209 uS

Relative speed:
| Speed 1.126 (time 0.888) relative to MT19937 getUnsignedSample
--------------- | -----------------------------------------------------------
---
```

SWB1gen Documentation

```
Parameters:  | r = 37
```

(./SCGgen.html)

(./Interface.html)

(./TGFSRgen.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
| float in [0.0,1.0) using 1 iteration of the generator.
| double in [0.0,1.0) using 1 or 2 iterations of the generator.
| long double in [0.0,1.0) using 2 iterations of the generator.
| (See the implementation file for more details.)

Output range:  | Any number in the range [0,2^w-1] may be returned

Valid seeds:  | Using create:setStateFromSeed:  in [1,2^32-1]
| Using create:setStateFromSeeds: in [1,2^w-1].

Cycles:  | With properly chosen parameters, these generators
| have a single cycle of length 2^(w*N) -1.

Output Quality:  | Good.

Reference for | Matsumoto and Kurita, op. cit.
output quality:  | Also recommended by P. L'Ecuyer (private communication).

TT403gen, | On a 486/66, the following approximate measures were obtained:
TT775gen, |   -getUnsignedSample:                        4.654 uS
TT800gen |   -getFloatSample, getDoubleSample:          6.430 uS
speed:  |   -getFatDoubleSample, getLongDoubleSample: 11.953 uS

Relative speed:
| Speed 0.795 (time 1.259) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--
```

TT403gen Documentation

```
Parameters:  | unsigned int w=31 // word length
| unsigned int N=13 // size of state array
| unsigned int M=2 // lag
| unsigned int s=8 // 'magic vector'
| unsigned int t=14 //    "     "
| unsigned int b=0x102d1200 //    "     "
| unsigned int c=0x66e50000 //    "     "
| unsigned int a=0x6b5eccf6 //    "     "

Reference for | Matsumoto and Kurita, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length 2^403 -1.

Output Quality:  | Good. Parameters chosen on the basis of theory
```

```
Output Quality:  | Good. Parameters chosen on the basis of theory
| and empirical testing.

Reference for | Matsumoto and Kurita, op. cit.
output quality:  |
```

(./SWB1-3gen.html)

(./Interface.html)

(./Prot.html#ProbabilityDist)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

(Interface.html)

(BernoulliDist.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

--------------- . ---
--

(./RandomBitDist.html)

(./Interface.html)

(./UniformIntegerDist.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
|  -getIntegerSample:                         7.534 uS
|  -average # of generator calls per variate: 1.000 (U)
|  -distribution net time:                    3.836 uS

Relative speed:
| Speed 0.491 (time 2.037) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--
```

(./BernoulliDist.html)

(./Interface.html)

(./UniformUnsignedDist.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
comment:        | Then minValue is returned "with probability 1".

UniformUnsignedDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getUnsignedSample:                        7.632 uS
|  -average # of generator calls per variate: 1.000 (U)
|  -distribution net time:                    3.934 uS

Relative speed:
| Speed 0.485 (time 2.064) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--
```

(./UniformIntegerDist.html)

(./Interface.html)

(./UniformDoubleDist.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
speed:  | using MT19937 as the underlying generator:

|  -getDoubleSample:                          13.777 uS
|  -average # of generator calls per variate:  1.000 (D)
|  -distribution net time:                     3.618 uS

Relative speed:
| Speed 0.268 (time 3.726) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--
```

(./UniformUnsignedDist.html)

(./Interface.html)

(./ExponentialDist.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

--------------- . ---
--

(./UniformDoubleDist.html)

(./Interface.html)

(./GammaDist.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-26

```
Relative speed:
| 0.00310 (time 32.294) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--
```

(./ExponentialDist.html)

(./Interface.html)

(./NormalDist.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

```
Name:  | LogNormalDistribution

Description:  | Log-Normal distribution returning double values

Reference:  | W.H. Press et al, "Numerical Recipes in C" (2nd. ed.).
| Cambridge University Press, 1992.

Algorithm:  | The Box-Muller method

Parameters:  | double Mean
| double Variance

State:  | double stored_value
| BOOL stored
| unsigned long long int currentCount

Output types:  | double

Output Quality:  | All bits should be safe.

LogNormalDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getDoubleSample:                          33.430 uS
|  -average # of generator calls per variate:  1.273 (D)
|  -distribution net time:                    20.498 uS

Relative speed:
| Speed 0.111 (time 9.040) relative to MT19937 getUnsignedSample
--------------- . ------------------------------------------------------------
--
```

(GammaDist.html)

(Interface.html)

Random Library: Usage Guide

Table Of Contents:

Overview

The random library basically consists of two parts, the actual pseudorandom number algorithms and the transformations that change the output of the pseudorandom number algorithms into the simulated distributions. The pseudorandom number algorithms are called generators throughout these documents and many times the "pseudo" is dropped from pseudorandom, even though there is no true random number generator implemented in Swarm.

Generator Usage Notes

Primary Author: Sven Thommesen
(mailto:sthomme@humsci.auburn.edu)
Swarm rel. 1.0.3 (random rel. 0.8)
Notes on the use of the random number generators
Sven Thommesen <sthomme@humsci.auburn.edu>
1997-09-01

IMPROVEMENTS over v. 0.6:

A host of new generators, located on the web or in the literature, have been added since the last version of Random. There is now a total of 36 different generators defined! Some of these have immense periods, some are very fast, and some have much better statistical properties than the old generators.

A new *type* of generator, the 'split' generator, has been introduced in the form of L'Ecuyer's C2LCGXgen and C4LCGXgen generators.

A 'split' generator is a long-period generator for which we are able to split the period into arbitrary sub-periods, which we can access quickly.

A bug in SWBgen was corrected. Code for ACG and SCG was also changed.

The -verifySelf method is gone. Instead see the test program located in /random/testR0.

The 'getState:' method has been named 'putStateInto: (void *) buffer', and the 'setState:' method is now 'setStateFrom: (void *) buffer'. A quick search-and-replace fixes things in your code.

Note: these methods have also changed somewhat, as has the size of the data being saved. As a result, v. 0.7 generators will refuse to 'setStateFrom' data saved by v. 0.6 objects.

There should be fewer changes like this in the next release.

USAGE notes:

I shall first discuss 'simple' generators, and then discuss how 'split' generators differ from 'simple' ones.

In the following text, wherever I use PSWBgen as an example you may substitute any other generator.

Note: any name that starts with 'my' is meant to designate a variable which you have defined in your own program.

SIMPLE generators:

You create a generator in one of 3 ways:

(a) the lazy way:

```
   myGenerator = [ PSWBgen createWithDefaults: [self getZone] ];
```

This allocates the object and initializes it with STARTSEED.

(b) using a single seed value:

```
   myGenerator = [ PSWBgen create: [self getZone]
setStateFromSeed: mySeed ];
```

```
myVector = [ myGenerator getInitialSeeds ];
```

And you can find out the largest seed values that are allowed for the particular generator:

```
myVector = [ myGenerator getMaxSeedValues ];
```

(These values vary from generator to generator, and they may not be the same for all elements of the vector for a given generator. Seeds may not be 0.)

NOTE: in the above two calls, the variable myVector is set to point to an array internal to the generator. If you want to preserve the values, you need to allocate space in your program either statically or dynamically, and use a for-loop to copy data from myVector[i] to myAllocatedVector[i].

You may reset the generator's state at any time by using the method

```
[ myGenerator setStateFromSeeds: (unsigned *) mySeedVector ];
```

This will also reset to 0 the currentCount variable.

NOTE: if you set a generator's state from a vector of seeds, the call

```
myUnsignedValue = [ myGenerator getInitialSeed ];
```

will return a value of 0 (an invalid seed). On the other hand, if you initialize the generator with a single seed value, the call

```
mySeedVector = [ myGenerator getInitialSeeds ];
```

will return the seed vector that would produce identical output to what you obtained using the single seed.

(d) antithetic values:

You can make the generator serve up antithetic values by setting:

```
[ myGenerator setAntithetic: myBooleanValue ];
```

If set, this makes -getUnsignedSample return (unsignedMax-x) instead of x, and the floating point methods return (1.0 - y) instead of y.

You can ascertain if this flag is set by calling

```
myBooleanValue = [ myGenerator getAntithetic ];
```

```
   myGenerator = [ C4LCGXgen create: [self getZone]
setA: 32 setv: 25 setw: 60
setStateFromSeeds: (unsigned *) mySeedVector ];
```

(In both cases, the only limitation is that A * 2^v * 2^w must be less than the generator's period, 2^60 for C2LCGX and 2^120 for C4LCGX.)

For obtaining output, we need to specify which of the A 'virtual' generators we want to draw from:

```
   myUnsignedValue = [ myGenerator getUnsignedSample: 12 ];
   myFloatValue    = [ myGenerator getFloatSample: myVirtualGenerator ];
   myDoubleValue   = [ myGenerator getThinDoubleSample: someUnsignedValue ];
   myDoubleValue   = [ myGenerator getDoubleSample: 32 ];
   myLongDoubleValue = [ myGenerator getLongDoubleSample: 0 ];
```

Note: virtual generators are numbered from 0 to A-1.

Obtaining the current count of variates generated likewise:

```
   myLongLongInt = [ myGenerator getCurrentCount: myVirtualGenerator ];
   myLongLongInt = [ myGenerator getCurrentSegment: myVirtualGenerator ];
```

The latter call indicates what segment number the virtual generator is currently drawing numbers from.

Other than these methods, the other methods discussed above under 'simple' generators are the same for 'split' generators.

In *addition* to this, 'split' generators have the following methods to manage the virtual generators:

```
   // Place all virtual generators at the start of the first segment:
   [ myGenerator initAll ]; // done automatically at creation

   // Place all virtual generators back to the start of the current segment:
   [ myGenerator restartAll ];

   // Place all virtual generators at the start of the next segment:
   [ myGenerator advanceAll ];

   // Place all virtual generators at the start of the indicated segment:
   [ myGenerator jumpAllToSegment: myLongLongIntValue ];
```

You may also address individual virtual generators:

```
   [ myGenerator initGenerator:    myVgen ];
```

The following code shows how to save an object’s state to disk:
(You should add your own code to deal with disk file errors,
either aborting or printing out error messages.)

```
// Ask how big a buffer we need:
stateSizeG = [ myGenerator getStateSize ];

// Allocate memory for the buffer:
stateBufG  = [[self getZone] alloc: stateSizeG];

// Ask the generator to put state data into the buffer:
[ myGenerator putStateInto: (void *) stateBufG ];

// Open a disk file for output:
myFile = fopen(myFileName, "w");
if (myFile == NULL) { }; // error on open: disk full, or no permissions

// Write the state buffer to disk in binary form:
status = fwrite(stateBufG, stateSizeG, 1, myFile);
if (status < 1) { }; // error on write: disk full?

// Close the file
status = fclose(myFile);
if (status) { }; // error on close ?

// Free the memory allocated to the buffer:
[[self getZone] free: stateBufG];

// Or, for test purposes, just zero the buffer data instead:
// memset(stateBufG, 0, stateSizeG);
```

This code shows how to set an object’s state from a disk file:

```
// Ask how big a buffer we need:
stateSizeG = [ myGenerator getStateSize ];

// Allocate memory for the buffer:
stateBufG  = [[self getZone] alloc: stateSizeG];

// Open a disk file for input:
myFile = fopen(myFileName, "r");
if (myFile == NULL) { }; // error on open: file not found

// Read state data into the memory buffer:
status = fread(stateBufG, stateSizeG, 1, myFile);
if (status < 1) { }; // error on read
```

For what it’s worth, Professor L’Ecuyer recommends his own C4LCGX and C2MRG3 generators as well as Matsumoto’s TT800 (the monster MT19937 hadn’t been released yet), and Prof. Marsaglia recommends his own Multiply-With-Carry generators (MWCA, MWCB, C3MWC, RWC2, RWC8="Mother").

UTILITY OBJECTS PROVIDED:

The following objects have been defined in <random/random.m>, and are immediately accessible from anywhere in your program:

```
id <MT19937>             randomGenerator;
id <UniformIntegerDist>   uniformIntRand;
id <UniformUnsignedDist>  uniformUnsRand;
id <UniformDoubleDist>    uniformDblRand;
```

The 3 distribution objects all draw their random numbers from the MT19937 generator, which has a period of 2^19937 (10^6001) and is quite fast.

Distribution Usage Notes

Primary Author: Sven Thommesen
(mailto:sthomme@humsci.auburn.edu)
Swarm rel. 1.0 (random rel. 0.6)
Notes on the use of the distribution objects
Sven Thommesen <sthomme@humsci.auburn.edu>
1997-09-01

IMPROVEMENTS over v. 0.6:

One new distribution class, BernoulliDist, has been added. It returns binary values (yes/true/1) with a given probability (while the old RandomBitDist has a fixed 50% probability, a fair coin toss.)

Distributions now have a new create method, ’+createWithDefaults: aZone’. This method creates the distribution object, and also a new generator object for its exclusive use. Each distribution class has a different default generator class assigned. These generators are initialized with STARTSEED, which by default equals the fixed value DEFAULTSEED, but will be equal to the varying RANDOMSEED if you start your program with the command line parameter ’-varySeed’.

All distributions have code to interact with the new ’split’ generators.

Each distribution object must be assigned a random generator on creation. You may not re-assign generators to a distribution after it has been created. You may, however, connect more than one distribution object to one generator, so that they end up drawing output from the generator in an interleaved fashion. Or you may create a new generator for each distribution object.

Each distribution has its own set of (1-2) parameters. You may deal with these parameters in two different ways: (1) you assign a set of default parameter values to the object on creation, and draw from the distribution using those parameters. Or (2), you may refrain from assigning default parameters, in which case you must specify the (possibly different) desired parameters on each call. You can (re-)set the default parameters any time, and you may call for a variate with specified parameter values even if different default parameters have been set.

Just like the generator objects, the distribution objects allow you to save and later restore their internal state.

Where I use NormalDistribution in examples below, substitute any other distribution and its parameters as needed.

NOTE: any name that starts with 'my' is meant to designate a variable which you have defined in your own program.

You create a distribution in one of several ways:

(a) the lazy way:

```
   myNormalDist = [ NormalDist createWithDefaults: [self getZone]];
```

This method will create a distribution object with no default parameters set, as well as a fresh generator object connected to it. The generator object is initialized with STARTSEED (see the discussion above). Different distribution classes use different generators for this purpose.

(b) Without default parameters, using a simple generator:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: mySimpleGenerator ];
```

'myGenerator' must of course first have been set to point to a random generator of the 'simple' type. Note that you cannot assign a different generator to a distribution after it has been created.

```
   myDouble = [ myNormalDist getDoubleSample ];
```

However, you *are* allowed to specify parameters even if default parameters have been set.

(Of course, different distributions have different parameters: RandomBitDist has none, the Uniform objects have minimum and maximum limit val-
ues, NormalDist
and LogNormalDist use Mean and Variance, ExponentialDist only Mean, and GammaDist used alpha and beta. See the file random/distributions.h for the specific methods available.)

(f) You may reset the default parameters this way, as often as you like:

```
   [ myNormalDist setMean: 3.3 setVariance: 2.2 ];
```

(g) You can obtain the current values of parameters:

```
   // Default parameters:
   myDouble1 = [ myNormalDist getMean ];
   myDouble2 = [ myNormalDist getVariance ];
   myDouble3 = [ myNormalDist getStdDev ];

   // Get a pointer to the generator object:
   myOtherGenerator = [ myNormalDist getGenerator ];

   // Get the number of the virtual generator (if a split generator is used):
   myUnsignedValue  = [ myNormalDist getVirtualGenerator];

   // Find out if default parameters have been set:
   myBoolean        = [ myNormalDist getOptionsInitialized ];

   // Find out how many variates the object has delivered so far:
   // (The counter is an unsigned long long int, which goes up to 2^64.)
   myLongLongInt    = [ myNormalDist getCurrentCount ];
```

(h) You can reset the variate counter and other state variables this way:

```
   [ myNormalDist reset ];
```

   This is most likely done in conjunction with resetting the connected generator, using [myGenerator setStateFromSeed: mySeedValue];

(i) Finally, we have the InternalState protocol methods:

```
[ myNormalDist putStateInto: (void *) stateBufD ];

// Open a disk file for output:
myFile = fopen(myFileName, "w");
if (myFile == NULL) { }; // error on open: disk full, or no permissions

// Write the state buffer to disk in binary form:
status = fwrite(stateBufD, stateSizeD, 1, myFile);
if (status < 1) { }; // error on write: disk full?

// Close the file
status = fclose(myFile);
if (status) { }; // error on close ?

// Free the memory allocated to the buffer:
[[self getZone] free: stateBufD];

// Or, for test purposes, just zero the buffer data instead:
// memset(stateBufD, 0, stateSizeD);
```

This code shows how to set an object's state from a disk file:

```
// Ask how big a buffer we need:
stateSizeD = [ myNormalDist getStateSize ];

// Allocate memory for the buffer:
stateBufD  = [[self getZone] alloc: stateSizeD];

// Open a disk file for input:
myFile = fopen(myFileName, "r");
if (myFile == NULL) { }; // error on open: file not found

// Read state data into the memory buffer:
status = fread(stateBufD, stateSizeD, 1, myFile);
if (status < 1) { }; // error on read

// Close the file:
status = fclose(myFile);
if (status) { }; // error on close

// Ask the distribution object to set its state from the buffer data:
[ myNormalDist setStateFrom: (void *) stateBufD ];

// Free the memory allocated to the buffer:
[[self getZone] free: stateBufD];
```

Random Library: Advanced Usage Guide

```
There should also be a general expository on the use of pseudo random number generation
in simulation, especially as applied to Swarm.
```

(./UsageGuide.html)

(../random.html)

(./InterfaceDesign.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

(AdvancedUsage.html)

(../random.html)

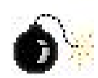
(Implementation.html)

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-10

Simtools Library

Documentation and Implementation Status



Compatibility

- 1.0.4 -> 1.0.5: GUISwarm now inherits from GUIComposite. Because GUIComposite handles the passthru of archiving keys to tkobjc primitives, the method setControlPanelGeometryRecordName is no longer needed; instead, the macro SET_WINDOW_GEOMETRY_RECORD_NAME is provided.

- 1.0.3 -> 1.0.4: All functions maintain backward compatibility. There are additional features, however, and features previously undocumented.

Overview

Simtools is the stdlib of Swarm... In other words, it is the library where we have parked many miscellaneous classes which, while very useful, are not specific enough to be placed in any other library. So, for example, simtools contains InFile, ObjectLoader and other I/O classes, the ControlPanel and GUISwarm which support GUI interactions, but also UName which simply generates unique names for objects.

Documentation

The *objectbase* library is defined and documented following the basic structure of the Library Interface Conventions

(defobj/library.html)
. The following standard sections of documentation are available:

- Interface Reference

Simtools Library: Documentation and Implementation Status

The simtools library has undergone an upgrade to the status of a library as of 1.0.4. This means the interface now conforms to the library interfaces specifications, the format of the documentation now reflects this. Other than new features, there should be no affect on the user.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

globalTkInterp (from libtkobjc) is set to be a normal Tcl interpreter instead of a Tcl/Tk/BLT interpreter.

Object Types (with links to reference pages)

As simtools is a collection of tools that are only loosely related to each other - the class hierarchy is virtually flat. Most of the classes are standalone.

Contents:

Name

GUISwarm

Synopsis

```
//
// GUISwarm -- a version of the Swarm class which is graphics aware
//

@protocol GUISwarm <SwarmProcess>
@end
```

Description

GUISwarm is a subclass of Swarm that is used as a toplevel Swarm for simulations running with a graphical user interface. The GUISwarm creates a ControlPanel automatically for you and defines a "go" method that interprets the state of the ControlPanel to keep things running in response to user input. Users subclass GUISwarm much like they subclass a normal Swarm, implementing the same kind of buildObjects, buildActions, and activateIn methods. When you are done building your Observer Swarm, start it as a toplevel via `[myGUISwarm go]`.

The control panel places a few responsibilities on the GUISwarm subclass author. In particular, a message to `[controlPanel doTkEvents]` should be scheduled fairly frequently - only when that method is executed will the user interface update (and the stop button be checked). Also, it is often useful to use `[controlPanel setStateStopped]` to wait for the user to indicate they're ready for execution to proceed.

- go
  Start the activity running, and also handle user requests via the control panel. Returns either `Completed` (the model ran until requested to terminate) or `ControlStateQuit` (the user pressed the quit button).

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Get the current button state of the controlpanel. Is one of ControlStateRunning, ControlStateStopped, ControlStateStepping, ControlStateNextTime, or ControlStateQuit.

- setStateRunning, - setStateStopped, - setStateStepping, - setStateQuit, - setStateNextTime

  Sets the state to the indicated value, and also sends the appropriate stop or terminate message to the running activity tree. The - setStateStopped message is particularly useful since it will cause the simulation to stop until the user interactively sets it back in motion (in other words, this method is useful in generating a software-triggered pause).

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

- setWindowGeometryRecordName: (const char *)name
  Set the name to be used for archiving the geometry of the control panel widget. There is no default; the position of the control panel will not be preserved unless you use this method.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

In order to record the window position of a ProbeDisplay, it is necessary to give it a name. This method is used to give an instance ProbeDisplay a name, which will used by the Archiver when recording its geometry information.

Regular Protocol

- getProbedObject
- getProbeMap
- update

    This method maintains consistency between the values of the probedObject's variables and the values which are displayed in the ProbeDisplay. Ideally, this method should be called every time the object is modified by the simulation. In practice, the user schedules an update on the probeDisplayManager which in turn communicates to all the active ProbeDisplays in the system.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Name

ProbeDisplayManager

Synopsis

```
//
// ProbeDisplayManager --
// A singleton class whose instance is used to manage all the
// ProbeDisplays created by the user during a GUI run of the
// simulation.
//

@protocol ProbeDisplayManager <SwarmObject>
USING
- createProbeDisplayFor: anObject;
- createProbeDisplayFor: anObject setWindowGeometryRecord-
Name: (const char *)name;

- createCompleteProbeDisplayFor: anObject;
- createCompleteProbeDisplayFor: anObject setWindowGeometryRecord-
Name: (const char *)name;

- addProbeDisplay: probeDisplay;
- removeProbeDisplayFor: anObject;
- removeProbeDisplay: probeDisplay;
- setDropImmediatelyFlag: (BOOL)dropImmediateFlag;
- update;
@end
```

Description

`- addProbeDisplay: pd`

`- removeProbeDisplay: pd`

These methods are used to specify which Probe Displays should be managed by the ProbeDisplayManager.

`- update`

This method will recursively send an update message to all the Probe Displays managed by the ProbeDisplayManager.

Name

InFile - class to perform file input

Synopsis

```
//
// InFile --
// A class which was intended to support file input. There have been
// justified requests from our userbase to re-design this interface.
//

@protocol InFile <SwarmObject>
CREATING
+ create: aZone withName: (const char *)theName;
USING
- (int)getWord: (char *)aWord;
- (int)getLine: (char *)aLine;
- (int)getInt: (int *)anInt;
- (int)getUnsigned: (unsigned *)anUnsigned;
- (int)getLong: (long *)aLong;
- (int)getUnsignedLong: (unsigned long *)anUnsLong;
- (int)getDouble: (double *)aDouble;
- (int)getFloat: (float *)aFloat;
- (int)getChar: (char *)aChar;
- (int)unGetChar: (char)aChar;
- (int)skipLine;
@end
```

Description

This class is intended to simplify the input file-I/O in Swarm. It essentially deals with the detailed file opening and closing routines thus alleviating the need for C file I/O procedure calls.

+ create: aZone withName: (const char *)theName
  This is the create method for InFiles, where theName is, of course the name of the file to open.

- (int)getWord: (char *)aWord

Name

OutFile - class to perform file output

Synopsis

```
//
// OutFile --
// A class which was intended to support file output.
//

@protocol OutFile <SwarmObject>
CREATING
+ create: aZone withName: (const char *)theName;
USING
- putString: (const char *)aString;
- putInt: (int) anInt ;
- putUnsigned: (unsigned) anUnsigned;
- putLong: (long) aLong ;
- putUnsignedLong: (unsigned long) anUnsLong ;
- putDouble: (double) aDouble ;
- putFloat: (float) aFloat ;
- putChar: (char) aChar ;
- putTab ;
- putNewLine ;
@end
```

Description

This class is intended to simplify output file-I/O in Swarm. It essentially deals with the detailed file opening and closing routines thus alleviating the need for C file I/O procedure calls.

`+ create: aZone withName: (const char *)theName`
: This is the create method for OutFiles, where theName is, of course the name of the file to open.

`- putString: (const char *)aString`

`- putInt: (int) anInt`

Name

AppendFile - class for appended file output

Synopsis

```
//
// AppendFile --
// A class to support appended file output.
//
@protocol AppendFile <SwarmObject>CREATING
+ create: aZone withName: (const char *)theName;
USING
- putString: (const char *)aString;
- putInt: (int)anInt;
- putUnsigned: (unsigned)anUnsigned;
- putLong: (long)aLong;
- putUnsignedLong: (unsigned long)anUnsLong;
- putDouble: (double)aDouble;
- putFloat: (float)aFloat;
- putChar: (char)aChar;
- putTab;
- putNewLine;
@end
```

Description

This class subclasses from OutFile, the only difference in functionality, being that it opens a given file in *Append*

mode, rather than *Overwrite*mode, which OutFile does.

+ create: aZone withName: (const char *)theName
  This is the create method for AppendFiles, where theName is, of course the name of the file to open.

- putString: (const char *)aString

- putInt: (int)anInt

- putUnsigned: (unsigned)anUnsigned;

Name

ObjectLoader - class to load objects IVars from a file

Synopsis

```
//
// ObjectLoader --
// An attempt to design some form of standard for object loading.
//

@protocol ObjectLoader <SwarmObject>
USING
+ load: anObject from: aFileObject;
+ load: anObject fromFileNamed: (const char *)aFileName;

- setFileObject: aFileObject;
- loadObject: anObject;
// In case the same class is being loaded multiple times...
- updateCache: exampleTarget;
@end
```

Description

This class is used to initialize the variables of a target object from a data file. The data file is required to have a

very simple format:

```
# A Comment

@begin

aVarName aValue
aVarName aValue # A comment
aVarName aValue
#
# A Comment
#
aVarName aValue
```

Name

ObjectSaver - class to save object's IVars to a file

Synopsis

```
//
// ObjectSaver --
// An attempt to design some form of standard for object saving.
// The ProbeMap argument is used to specify what variables get saved.
//

@protocol ObjectSaver <SwarmObject>
USING
+ save: anObject to: aFileObject;
+ save: anObject to: aFileObject withTemplate: aProbeMap;
+ save: anObject toFileNamed: (const char *)aFileName;
+ save: anObject toFileNamed: (const char *)aFileName withTemplate: (Probe-
Map *)aProbeMap;

- setFileObject: aFileObject;
- setTemplateProbeMap: aProbeMap;
- saveObject: anObject;
@end
```

Description

This class is used to write an object's variables to a specified file. If only a subset of the variables should be

written out, the set is specified by a template ProbeMap (where the ProbeMap will contain Probes for those variables which should be saved).

+ save: anObject to: aFileObject

+ save: anObject to: aFileObject withTemplate: aProbeMap

+ save: anObject toFileNamed: (const char *)aFileName

+ save: anObject toFileNamed: (const char *)aFileName withTemplate: aProbeMap

Name

QSort - class to sort a collection

Synopsis

```
//
// QSort --
// A class (not to be instantiated) wrapper around the C sort routine.
//

@protocol QSort <SwarmObject>
USING
+ sortObjectsIn: aCollection;
+ (void)sortObjectsIn: aCollection using: (SEL)aSelector;
+ (void)sortNumbersIn: aCollection;
+ (void)sortNumbersIn: aCollection us-
ing: (int(*)(const void*,const void*)) comp_fun;
+ (void)reverseOrderOf: aCollection;
@end
```

Description

QSort is simply a "wrapper" for the C native "qsort" function, as applied to a Swarm collection. The values will appear in ascending order by default. Reversing the order of a collection can be made by calling reverseOrderOf. All these methods modify the underlying collection, so any indexes should always be regenerated.

`+(void) sortObjectsIn: aCollection`
: Given a collection, this method will sort the objects with the "compare" function for the object. If the objects don't provide a compare function, it uses the default from the defobj library.

`+(void)sortObjectsIn: aCollection using: (SEL)aSelector`
: This method allows the user to specify a given message selector to use to "compare" the objects.

`+(void) sortNumbersIn: aCollection;`
: This method is intended specifically for integer comparisons, again using a default "compare" function. The default assumes numbers should be monotonically increasing.

Name

NSelect - class to select exactly N elements at random from a collection

---

Synopsis

```
//
// NSelect --
// A class (not to be instantiated) wrapper around a Knuth algorithm
// for the selection of exactly N elements form a collection without
// repetition.
//

@protocol NSelect <SwarmObject>
USING
+ (void)select: (int)n from: aCollection into: bCollection;
@end
```

Description

NSelect selects exactly N elements from a collection without repetition. A target collection must be provided.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

+ create: aZone setBaseNameObject: aStringObject

- setBaseName: (const char *)aString

- setBaseNameObject: aStringObject

    These messages should be used to set or reset the base name.
    Note: these methods will automatically reset the counter.

- (const char *)getNewName

- getNewNameObject

    Generates a new name in the requested format.

- resetCounter

    Resets the counter used as a suffix in the unique names generated.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1997-12-19

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

TkObjc Documentation

Tcl/Tk

(ftp://ftp.smli.com/pub/tcl/)
is a scripting language and graphical widget set. TkObjc is a library of wrapper classes around Tk and BLT widgets. It's purpose is to provide a simple graphical interface while hiding most Tk-specific code from library users. To create and use graphical widgets, the user merely needs to create and use objects. Many of the objects here are straightforward wrappings of Tk widgets, but some (ButtonPanel, for instance) are combinations of other widgets, and others (Raster) are novel code.

TkObjc is currently based on Tcl 7.5, Tk 4.1
(ftp://ftp.smli.com/pub/tcl/)
, BLT 2.1
(ftp://ftp.neosoft.com/pub/tcl/NEW/)
or BLT 2.3
(http://www.tcltk.com/blt/index.html)
,and tclobjc-1.2
(ftp://ftp.santafe.edu/pub/swarm/needed-software/tclobjc-1.2.tar.gz)
(available from Swarm authors). Very little of this code is library-version dependent, however, as most of it works by directly calling the Tk interpreter.

The basic purpose of tkobjc is to package Tk functionality. Therefore, tkobjc's behaviour is similar to the Tk toolkit. For simple usage one should be able to get fairly far just by looking at this document, the header files, and the Swarm examples: more complicated graphical output will require the programmer have some familiarity with Tk.

---

globalTkInterp

TkObjc gets most of its work done by talking to the Tcl interpreter object `globalTkInterp`, an instance of class TkExtra. Strings handed to the interpreter's eval: method are evaluated by the Tcl interpreter and any side effects are performed. The libtclobjc library provides the functionality that glues Tk to Objective C. Users with simple needs never need to talk to the Tk interpreter directly.

Widget Base Class

All graphical widgets inherit from the Widget base class. Widget defines most of the behaviour needed: Widgets are created by the user with a particular parent, and then "pack"ed in order

Button

Buttons a user can click. Each button has a label and a command to be executed when the button is pressed.

- setText: (const char *)t
  set the button's text string and resize to fit.
- setCommand: (const char *)c
  set the button's command, what to do. This is a Tcl string passed off to the interpreter. See the ControlPanel code in simtools for an example.

Input Widgets

All widgets which take input from the user inherit from a base class, InputWidget. InputWidgets get their input in one of two ways: by being readable, or by being linked to a C variable.

InputWidget (base class)

- (const char *)getValue
  read the value as a string
- setValue: (const char *)v
  set the value as a string
- linkVariable: (void *)p Type: (int)type
  Link the widget directly to a C variable - any changes from the user will be automatically stored in the variable. Types are TCL_LINK_INT, TCL_LINK_BOOLEAN, TCL_LINK_DOUBLE, and TCL_LINK_STRING.

Entry

Handles data of any input type - user types in a string.

CheckButton

Boolean data only, click to set.

- setColor: (const char *)color, - setWidth: (int)w, - setDashes: (int)i, - setSymbol: (const char *)s

    change the apperance of the graph. The argument to dashes specifies the dash width. Valid symbols are line (default), square, circle, diamond, plus, cross, splus, scross.

- addX: (double)x Y: (double)y

    add the datapoint (x,y) to the graph

- resetData

    remove all data from this element, start over

-(BLTVector *)getXData, -(BLTVector *)getYData

    Get the data in the graph as a BLTVector (undocumented)

Histo

Histogram display tool, based on BLT's barchart. The number of bins is fixed at creation time, then the user hands the Histo an array of datapoints (double or int) to display (or optionally an array of datapoints and *locations* where the bars should be drawn (specified as doubles).

- title: (const char *)t, - axisLabelsX: (const char *)xl Y: (const char *)yl

    title the histogram, give it axis labels.

- setNumPoints: (int)n Labels: (const char * const*) l Colors: (const char * const *)c

    initialize the histogram, tell it how big its dataset is. Labels and Colors are arrays of strings (one string per point) for text labels and the colours of the bars (the last two arguments are optional).

- drawHistoWithDouble: (double *)points, - drawHistoWithInt: (int *)points

    an array of points (two possible types) to draw in the histogram.

- drawHistoWithDouble: (double *)points atLocation: (double *)locations, - drawHistoWithInt: (int *)points atLocation: (double *)locations

    An array of points (two possible types) to draw in the histogram, and an array of specific locations in which to draw them (the bar width at every location will be of width 1.0).

Graphical Output

TkObjc has one basic class (and a subclass) for displaying graphical data. The model used here is a 2d raster display: an array of display elements. Display elements are typically simply pixels, blocks of a solid colour (specified via an XColormap). Other options are a pixmap (XPixmap) or arbitrary Xlib display code (implementation of XDrawer).

ZoomRaster

ZoomRaster is a subclass of Raster that implements a zoomable image. It handles translation between logical coordinates and screen coordinates.

- increaseZoom, - decreaseZoom, - setZoomFactor: (int)z
  change the zoom size
- (unsigned)getZoomFactor
  returns an integer representing the zoom size
- handleConfigureWidth: (unsigned)newWidth Height: (unsigned)newHeight
  reconfigures zoom when the window is resized

XDrawer, XPixmap

Using XDrawer and XPixmap, you can write arbitrary Xlib code to draw in the window. XDrawer is just a protocol for the drawOn:X:Y:GC:Caller: message.

XPixmap provides packaging for drawing pixmaps onto displays.

- setFile: (const char *)filename
  designates the file from which to read the pixmap image
- createEnd
  loads the pixmap and sets the display and window
- (Pixmap) getPixmap
  returns the id of the pixmap data structure
- (BOOL) getMasked
  returns YES if a mask has been set, NO if it hasn't
- (Pixmap) getMask
  returns the id of the pixmap mask
- (unsigned) getWidth, -(unsigned) getHeight
  returns the width or height of the pixmap

Canvas

There are the beginnings of a packaging of the Tk canvas widget that allows arbitrary drawing of complicated shapes. This code is still under development and is not thoroughly documented. Canvas

is a subclass of Widget and so inherits those methods but overrides createEnd.

set the two diagonal corner coordinates of the rectangle

TextItem : CanvasItem

- setX: (int)x Y: (int)y
  sets the coordinate for the center of the text
- setText: (const char *)text
  sets the content of the text

CompositeItem

A CompositItem is simply a CanvasItem that consists of several pieces. (Why doesn't this inherit from CanvasItem?) The interface is identical to CanvasItem except for the additional method moveX:Y. Again, this is an abstract class.

- moveX: (long)delta_x Y: (long)delta_y

  passthrough method to be implemented by subclass

NodeItem : CompositeItem

- setX: (int)x Y: (int)y
  position the item on the canvas
- (int)getX, - (int)getY;
  return the x or y coordinate for the item
- setString: (const char *)string
  designate the string that will label this node
- setColor: (const char *)aColor
  designate the fill color of the node
- setBorderColor: (const char *)aColor
  designate the border color of the node
- setBorderWidth: (int)aVal
  set the width of the border in pixels

LinkItem : CompositeItem

- setFrom: from
  designate the object (node) that will be the source of the link
- setTo: to
  designate the node that will be the destination of the link
- setColor: (const char *)aColor
  set the color of the link

Analysis

This is the library where tools primarily related to analysis tasks, reside. This includes tools which simplify the task of graphing values or displaying distributions as well as more specific measurement tools (such as Average, Entropy).

Averager

Averager objects read a value (via a MessageProbe) from a collection (typically a list) of objects and collect statistics over them.

- setCollection: aCollection
  sets the collection of objects that will be probed.
- setProbedSelector: (SEL)s
  The message to be sent to each element in the collection (inherited from MessageProbe)
- (double)getAverage, - (double)getTotal, - (double)getMax, - (double)getMin - (int)getCount
  various statistics that the averager collects. These are read out of the object, not computed everytime they are asked for.
- update
  Update the average - send the message throughout the collection and update the average, total, etc. Schedule this message when you need to collect data.

Entropy

Entropy objects read probabilities (via a MessageProbe) from a collection of objects and calculate the entropy of the underlying distribution.

NOTE: This implies that the objects which are being polled must somehow calculate a valid probability, in other words, they are probably aware of a global property such as N (the number of agents in the system).

- setCollection: aCollection
  sets the collection of objects that will be probed.
- setProbedSelector: (SEL)s
  The message to be sent to each element in the collection (inherited from MessageProbe)
- (double)getEntropy
  Returns the calculated Entropy (i.e. the entropy is read out of the object, not computed everytime it is requested).

The title is used in the graphical version of the EZGraph and appears at the top of the graph window.

- setAxisLabelsX: (const char *)xl Y: (const char *)yl
  Sets the labels on the graph. Only relevent for the graphical version.

- createSequence: (const char *)aName withFeedFrom: anObj andSelector: (SEL)aSel
  This creates a sequence in the EZGraph based on the return value provided by the object *anObj* when sent the selector *aSel*. If file I/O is enabled then the data will be sent to a file with the name *aName*, otherwise the *aName* argument is simply used as the legend for the graph element generated by EZGraph.

- createAverageSequence: (const char *)aName withFeedFrom: aCollection andSelector: (SEL)aSel

- createTotalSequence: (const char *)aName withFeedFrom: aCollection andSelector: (SEL)aSel

- createMinSequence: (const char *)aName withFeedFrom: aCollection andSelector: (SEL)aSel

- createMaxSequence: (const char *)aName withFeedFrom: aCollection andSelector: (SEL)aSel

- createCountSequence: (const char *)aName withFeedFrom: aCollection andSelector: (SEL)aSel
  These methods take a collection of objects and generate a sequence based on a statistic over the responses of the *entire* object-set (as opposed to a single object, as in - createSequence:withFeedFrom:andSelector:).

- step
  This method will cause the graphical display and/or the output files to be updated, i.e. the selector is applied to the target object(s) and the results are then displayed graphically and/or written to the appropriate output files.

- getGraph
  This method lets the user access the BltGraph generated internally by the EZGraph (assuming the creation of a graphical display was in fact required).

EZBin

This class allows the user to easily histogram data generated by a collection of objects. In addition the class will generate some standard statistics over the resulting dataset.

- setGraphics: (int)state
  If the user does not require a graphical display of the histogram, s/he should call this method with state set to 0. The default state is 1 meaning that by default the data appears graphically in a window.

- (int)getOutliers
  Get the number of entries which landed out of the bounds of the histogram. Pressing the "o" key on the graphical representation of the histogram will display this value (both as an integer and as a percentage of the total number of attempted entries).
- (double)getMin
- (double)getMax
- (double)getAverage
- (double)getStd
  These messages are used to get some standard statistics over the dataset.

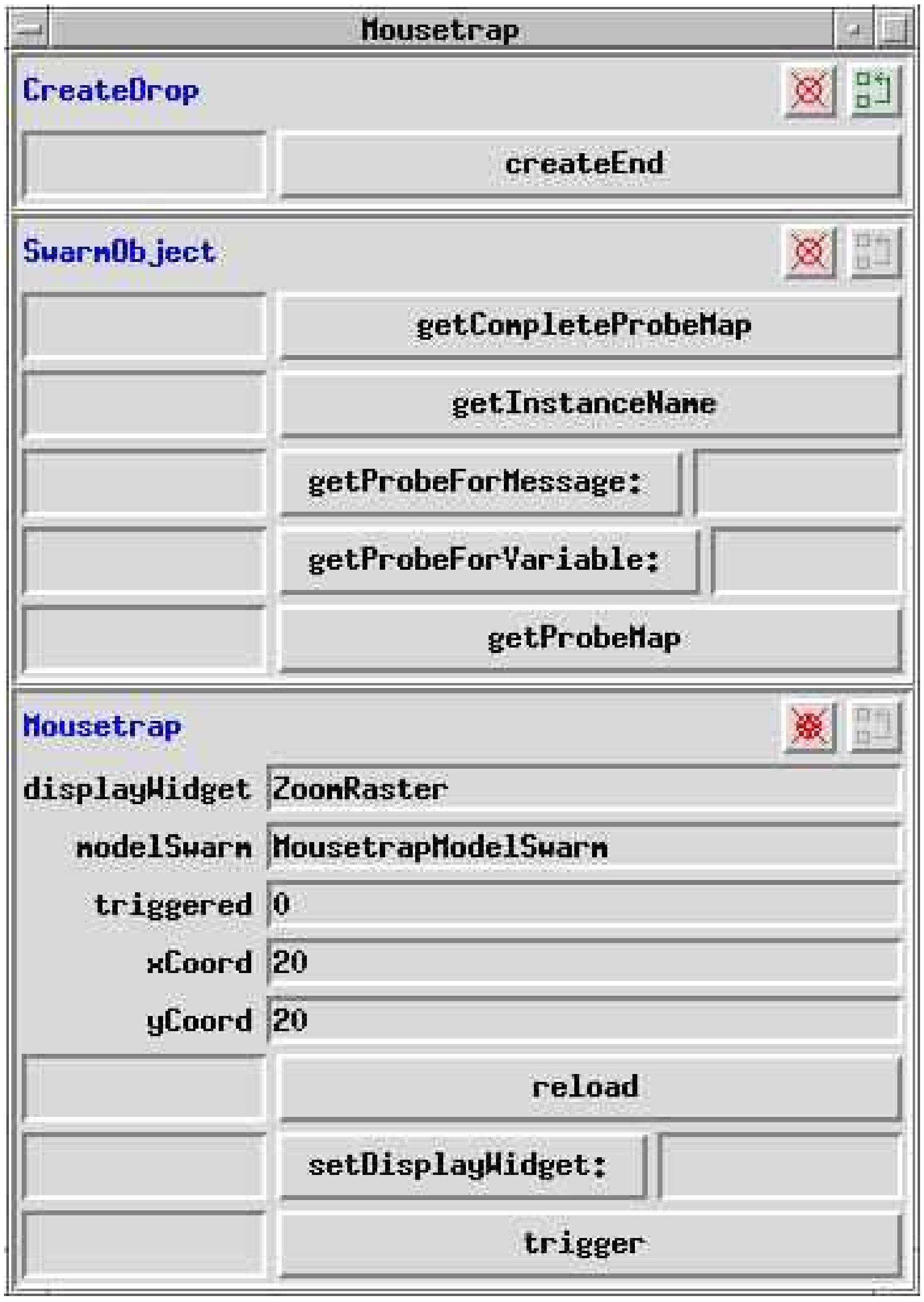
EZDistribution

This is a subclass of EZBin which normalizes the data and treats it as a distribution. This means that in addition to the statistics it can calculate by virtue of being a subclass of EZBin, it can also calculate the entropy of the distribution as well as return the probabilities associated with the individual bins.

- (double *)getProbabilities
  Returns an array of doubles representing the probability of every bin in the distribution.
- (double)getEntropy
  Returns the entropy of the distribution.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08


Mousetrap
CreateDrop
createEnd
SwarmObject
getCompleteProbeMap
getInstanceName
getProbeForMessage:
getProbeForVariable:
getProbeMap
Mousetrap
displayWidget
ZoomRaster
modelSwarm
MousetrapModelSwarm
triggered
0
xCoord
20
yCoord
20
reload
setDisplayWidget:
trigger

```
[probeDisplayManager createProbeDisplayFor: anObject] ;
```

And that the contents of the returned ProbeDisplay will be based on a ProbeMap obtained from the probed object itself.

Probe support resident in the objectbase
(objectbase.html)
library is documented with that library. Extra description, above and beyond the bare interface description for precision probing and SwarmObject is below:

Precision setting for floating-point numbers with probes

Global setting of precision

There are two types of global precision setting via ProbeLibrary
(objectbase/ProbeLibrary.html)
:

-setDisplayPrecision: (int) nSigFigsSaved
: Sets the number of significant figures for floating point (and double-floating) numbers diplayed on a GUI widget. Currently this is only implemented for VarProbes. The display uses the %*g sprintf-style formatting, which can vary slightly from implementation to implementation. If you set the number of significant figures to 3, then a float of value of 0.6344346 is displayed as 0.634 on the GUI widget. Note that this in no way affects the underlying stored value of the floating point number.

-setSavedPrecision: (int) nSigFigsSaved
: Sets the global default for the saving of floats through ObjectSaver. All objects with floats and doubles as instance variables are saved with the precision specified by this method. This is independent of the displayed precision of the same instance variable on a GUI widget.

To actually initialise these defaults: in the top level swarm, you should add the calls to the global ProbeLibrary instance (which is actually created by the initSwarm call in main) during the - createBegin method (this sets the precision in the global instance, before any probes are checked out of the instance. If neither method is called on probeLibrary, then the precision defaults to six significant figures in both cases.

Example: Here's how it's done in HeatbugObserverSwarm.m:

```
+ createBegin: aZone
{
  HeatbugObserverSwarm * obj;
```

```
  HeatbugModelSwarm * obj;
  id <ProbeMap> probeMap;
  id floatProbe;

  [...]
  // the -setFloatFormat is applied to the probe which is
  // "returned" from the call to probeLibrary
  floatProbe = [[probeLibrary getProbeForVariable: "randomMoveProbability"
    inClass: [self class]]
                                  setFloatFormat: ".3f"];

  // now we have the probe - put it back into the customMap
  [probeMap addProbe: floatProbe];

  [...]
  return obj;
}
```

Or, more compactly:

```
+ createBegin: aZone
{
  HeatbugModelSwarm *obj;
  id <ProbeMap> probeMap;

  [...]

  [probeMap addProbe: [[probeLibrary getProbeForVari-
able: "randomMoveProbability"
    inClass: [self class]]
                           setFloatFormat: "%.3f"]];
  [...]
  return obj;
}
```

SwarmObject Support for Probing

As mentioned earlier, all SwarmObjects can generate Probes/ProbeMaps. By default, they achieve this by "checking out" the appropriate references from the probeLibrary.

> Interface definition
> (objectbase/SwarmObject.html)
> from objectbase
> (objectbase.html)
> library.

On the one hand if an object to be probed is specified *without* any particular ProbeMap being specified, then the ProbeDisplay generated will provide a window of class DefaultProbeDisplay, which displays all the variables resident in that class structure.

On the other hand, if a ProbeMap *is* specified then the ProbeDisplay follows exactly the specification as represented by the contents of the ProbeMap (above). When used in this manner, ProbeDisplays can generate tailored interfaces to objects (so for example, we have purposefully hidden certain instance variables in the HeatbugModelSwarm class, and have shown only one of the methods which the class understands).

Graphical Interface Usage

Common to both the standard ProbeDisplay and the CompleteProbeDisplay:

- The different fields in the ProbeDisplay can be updated by typing in new values and pressing Return. However, certain fields (containing pointers or ids, for example) cannot be modified and will generate a beep if such a modification is attempted.
- If an instance variable/argument slot is defined to hold an object, then that object can be drag&dropped into another variable/argument slot by clicking on it with the first mouse button (a small rectangle with the name of the object will appear - simply drag it to another object-typed variable/argument

library.

Marcus G. Daniels <mgd@santafe.edu>
(mailto:mgd@santafe.edu)

Last modified: 1998-01-08