http://www.santafe.edu/projects/swarm/
(http://www.santafe.edu/projects/swarm/)
    Swarm's Internet presence.

intro.html
(intro.html)
    What a Swarm application looks like, and how it is developed.

Installing Swarm

Swarm uses quite a few libraries and is intended to run under all major flavors of Unix. This presents the unpleasant but largely unavoidable side-effect of making Swarm hard to install. Ideally, you should get the sys-admin for your site to do the job. In any event, the following resources should prove useful:

http://www.santafe.edu/projects/swarm/release.html
(http://www.santafe.edu/projects/swarm/release.html)
    Swarm release web pages.

install.html
(install.html)
    Installation instructions

overview.html
(overview.html)
    Overview of the Swarm system.

To Do list
(todo.html)
    Additions we want to have to Swarm.

FAQ.html
(faq/FAQ.html)
    FAQ for the Swarm project.

LICENSE-libs
(LICENSE-libs)
    License terms for the distribution of the Swarm libraries.

LICENSE-apps
(LICENSE-apps)
    License terms for this distribution of the Swarm example applications.

Compatibility

static seed for 1.0.2. This is acceptable because the use of the default generators and distributions is DEPRECATED.

4. 1.0.0 => 1.0.1

There should not be any imcompatibilities between 1.0.1 and 1.0.0. There are a couple of changes that affect the behavior of Swarm, however. The big ones are:

- The automatic dropping of probe displays upon the dropping of an object to which those probe displays were attached. This could break your application code if you leave in the `dropProbeDisplaysFor:` message where you drop such an object.
- The default probeMap has changed. This is only an issue when `createProbeDisplayFor:` is called without having previously created and installed a probeMap for that class. The new behavior is to create a probeMap on-the-fly that contains only the instance variables for that class.

See the Release Page

(http://www.santafe.edu/projects/swarm/release.html#changes)
for other differences.

5. *Beta* => 1.0.0

As can be expected with any software package, it is sometimes unavoidable that changes in the functionality of the package will cause incompatibilities with earlier versions. This is especially true when a package is a "proof-of-principle" package like Swarm. We've made an attempt to compile all the problems a user might have moving to the new release and put them here. Please read this thoroughly to decide what you might need to do to your app to get it to work with the new release.

a. The biggest and most pervasive problems will be due to the new *random* library. Notes on how to deal with this problem are provided in the documentation for the *random* library
(src/random.html)
.

b. A rather benign problem results from the repackaging of the *swarmobject* library. The interface to this library was brought into sync with the *defobj* and *collections* libraries. (The rest of the errant libraries will follow in a later release.) The solutions to the problems associated with this interface change are detailed in the documentation for the *objectbase* library
(src/objectbase.html)
.

c. There are a couple rather benign incompatibility introduced with 1.0 in the new *activity* library. Many parts of *activity* have changed. But, for the most part, everything works exactly the same. For details on the incompatibilities please see the documentation for the *activity* library
(src/activity.html)
. Briefly, the incompatibilities are:

When adding printf() just isn't enough...

emacs tips
(emacs-objc.html)
Some hints for editing Objective C code under emacs.

Core Libraries

Swarm is essentially a collection of libraries against which you link your simulation code. The following is a list of the those libraries which are crucial to the proper execution of Swarm, and are therefore shipped with the Swarm distribution.

- Defobj Library (src/defobj.html)
- Collections Library (src/collections.html)
- Activity Library (src/activity.html)
- Objectbase Library (src/objectbase.html)
- Space Library (src/space.html)
- Random Library (src/random.html)
- Simtools Library (src/simtools.html)
- Graphical Interface Library (tkobjc) (src/tkobjc.html)
- Analysis Tools (src/analysis.html)
- Probe Interface (src/probing.html)

Sample Application

The following sample applications represent the current best source of guidelines/advice about the structure of Swarm applications and the way in which they should be written (along with `intro.html`

(intro.html)
). In particular, the Heatbugs code has been thoroughly commented and demonstrates some of the main features in Swarm. The template application is a good place from which programmers can copy code as a starting point for their

A Top-Down Introduction To Implementing an Experiment Using Swarm

This document attempts to explain the logical structure of a Swarm experiment application. Starting with a very general outline of an idealized experimental procedure, we successively increase the level of specification of each stage of this idealized structure until we arrive at details of an actual running Swarm application.

Along the way, we introduce and describe (very briefly) some of the tools currently available in Swarm to help users build experiments. The tools presented here are only suggestive: the Swarm library documentation and example applications will give a more detailed view of using specific components of Swarm.

These tools are more fully documented in the appropriate library directories. This high-level overview skips much of the detail in an effort to illustrate the logical structure of a Swarm application and to provide some motivation for why things are done the way they are in the example applications.

How we'll proceed

In the following, we will start out with a very high-level outline/abstraction of the "generic form" of an experimental procedure. Then we will increase the magnification on each piece of the outline until we reach specific details of Swarm code. If you compare the gross structure of the example applications you will see that despite the differences in detail, the overall architecture of Swarm applications adheres to the general abstraction of an experimental procedure.

The application we will use as an example is the Heatbugs application. The source code for that example is a thoroughly commented application and intended to be read by new Swarm programmers. It can be a useful template to start from: so can the very-stripped-down "template" application. We recommend that you gradually ease yourself into the Swarm perspective by experimenting with the example applications and branching out by modifying them in various ways until you catch on to the conceptual style of a Swarm Experiment.

> Therefore, a very important aspect of setting up an experiment in a computer is how one weaves the multiple threads of time that must be woven together coherently in order to produce reliable, repeatable results. Much of our work on Swarm has been devoted to not only making the task of managing concurrency manageable, but towards mechanisms to make people aware that they are *always* making implicit assumptions about how multiple threads of time are interacting with one another in their experimental setups. Swarm forces experimenters to make their concurrency assumptions explicit, so that others can reproduce their results by implementing the same assumptions about the flow of time.

Mag 2x: Basis of Swarm Computation

Here is a first approximation to embedding the above outline of an experimental procedure in Swarm code:

> Swarm is implemented in the Object-Oriented Programming language Objective-C. Computation in a Swarm application takes place by having objects send messages to each other. The basic message syntax is

```
[targetObject message Arg1: var1 Arg2: var2]
```

Where "targetObject" is the recipient of the message, "messageArg1:Arg2:" is the message to send to that object, and "var1", "var2", etc, are arguments to pass along with the message. Objective C's messages are keyword/value oriented, which is why the message name "messageArg1:Arg2:" is interspersed with the arguments.

> The whole idea of Swarm is to provide an execution context within which a large number of objects can "live their lives" and interact with one another in a distributed, concurrent manner. Furthermore, we wish to insulate the user from having to master all of the highly baroque computer-science wizardry usually required to implement such massively distributed systems of autonomous agents reliably and robustly.
>
> In the context of the Swarm simulation system, the generic outline of an experimental procedure takes the following form:
>
> 1. Create an artificial universe replete with space, time, and objects that can be located, within reason, to certain "points" in the overall structure of space and time within the universe., and allow these objects to determine their own behavior according to their own rules and internal state in concert with sampling the state of the world, usually only sparsely.
> 2. Create a number of objects which will serve to observe, record, and analyze data produced by the behavior of the objects in the artificial universe implemented in step 1)
> 3. Run the universe, moving both the simulation and observation objects forward in time under some explicit model of concurrency.

petri dish on the lab bench. Other observer objects can then input data into the model swarm (setting simulation parameters, for instance) and read data out of the model swarm (collecting statistics of the behavior of agents).

Just as in setting up a model swarm, an observer swarm has a collection of objects (the instrumentation), a schedule of activity, and a set of inputs and outputs. The activity of the observer schedule is to drive data collection - read this number out of the model, draw it on a graph. The inputs to the observer swarm are configurations of the observer tools: what sorts of graphs to generate, for instance. The outputs are the observations.

When running in graphics mode, the observer swarm objects are largely used to mediate user interface. For instance, in Heatbugs the HeatbugObserverSwarm creates Raster widgets, EZGraphs, and Probes. All of these objects are connected into the HeatbugModel swarm to read data, and to graphical interface objects so the human sitting in front of the computer can observe the world.

Interactive, graphical experimentation with models is useful for coming up with intuitions. But for serious experimentation it is necessary to collect statistics, which means doing many runs and storing data for analysis. As an alternative to a graphical observer swarm, you can also create *batch swarms*, observer swarms that are intended to be run without any interaction at all. Instead of putting up fancy graphics, batch swarms read data from files to control the model and write the data out to other files for analysis. The key here is that the model swarm used in the batch swarm is the exact same model as that used in a graphical observer swarm: the only difference is what tools the observer (batch or graphical) connects to the model.

Summary

Multiple Swarms are used to create an experimental apparatus and control it. The use of multiple Swarms is not restricted only to this use, though: in particular, a model Swarm could itself contain its own subswarms, building a hierarchical simulation. In future Swarm development, we intend to use the power of the multiple Swarm modeling approach to build complicated and flexible models.

Mag 4x: Sketch of Code

Now that we have multiple Swarms and an experimental apparatus, it's time to learn how to use the objects themselves inside an application. Some examples are provided here: to understand this better, it will be necessary to read through example applications and the library documentation. These examples come from the Heatbugs application.

Defining an Agent

The agents are typically the real focus of a modeling effort. Most of the work in a simulation comes in defining the agent behavior so that the computer agents resemble the real world phenomena you are trying to create.

In the case of Heatbugs, agents are pretty simple. Here is their definition, from Heatbug.h:

```
@interface Heatbug: SwarmObject {
 double unhappiness; // my current unhappiness
 int x, y; // my spatial coordinates
 HeatValue idealTemperature; // my ideal temperature
 HeatValue outputHeat; // how much heat I put out
 float randomMoveProbability; // chance of moving randomly

 Grid2d * world; // the world I live in
 int worldXSize, worldYSize; // how big that world is
 HeatSpace * heat; // the heat for the world
 Color bugColor; // my colour (display)
}

-setWorld: (Grid2d *) w Heat: (HeatSpace *) h; // which world are we in?
-createEnd;

-(double) getUnhappiness;

-setIdealTemperature: (HeatValue) i;
-setOutputHeat: (HeatValue) o;
-setRandomMoveProbability: (float) p;
-setX: (int) x Y: (int) y; // bug's position
-setBugColor: (Color) c; // bug's colour (display)

-step;

-drawSelfOn: (Raster *) r;
```

Heatbug is a subclass of SwarmObject. SwarmObjects have very little behavior of their own - they are defined as the root class of most objects and control computer science aspects like memory allocation and probability.

Heatbug carry with them a variety of state variables. For instance, each Heatbug has a notion of its ideal temperature, which will affect it's behavior. In addition, Heatbugs have variables that let them know about the world: these agents are storing references to the HeatSpace object, for example.

Most of the Heatbug methods have to do with setting up the agents state - the inputs to a Heatbug. Every heatbug must set up its world and heat objects, via the `setWorld:Heat:` method. In addition when Heatbugs are created they have their ideal temperature set, their output heat, etc. Heatbugs are also observable. Heatbugs

Building Space objects

In Swarm, spaces are really just another kind of agent. In the heatbugs model we create a HeatSpace, a subclass of a diffusion object from the Swarm space libaries (specified in HeatSpace.m). Here is the code from buildObjects in the HeatbugModelSwarm

```
heat = [HeatSpace createBegin: [self getZone]];
[heat setSizeX: worldXSize Y: worldYSize];
[heat setDiffusionConstant: diffuseConstant];
[heat setEvaporationRate: evaporationRate];
heat = [heat createEnd];
```

the object is created, a few parameters are set, and then the creation is finalized.

Scheduling a Model Swarm

Once all of the simulated objects are created in buildObjects, the next task is to schedule them in the method buildActions.

```
modelActions = [ActionGroup create: [self getZone]];
[modelActions createActionTo: heat message: M(stepRule)];
[modelActions createActionForEach: heatbugList message: M(step)];
[modelActions createActionTo: heat message: M(updateLattice)];

modelSchedule = [Schedule createBegin: [self getZone]];
[modelSchedule setRepeatInterval: 1];
modelSchedule = [modelSchedule createEnd];
[modelSchedule at: 0 createAction: modelActions];
```

The heatbug model schedule actually consists of two components: an ActionGroup called modelActions and a Schedule called modelSchedule. The ActionGroup is a tightly coupled list of three messages: every time the action group is executed, it will send three messages in a row:

```
[heat stepRule];
[heatbugList forEach: step];
[heat updateLattice];
```

The ActionGroup alone specifies three messages to send - in order to put it in the simulation, that ActionGroup is then dropped into a Schedule. The Schedule itself only has one action - to execute modelActions itself. That action takes place at time 0. But because we've set a repeat interval on the schedule of 1, the schedule itself loops, executing every 1 time step. The final result is that modelActions is executed at time 0, time 1, etc.

Building a Graphical Observer Swarm

With the model swarm defined, arranging for a graphical observer Swarm is the next step. For Heatbugs, the code is in HeatbugObserverSwarm. The structure of an observer swarm is almost (in this case a List) using the provided message (in this case getUnhappiness), and generate a datapoint from these values by averaging them.

main()

The last main type of code needed for an application is the function main(), the first function called in your program. All the real work has been done already - all that's left is to create the objects at the right time.

```
int
main(int argc, char ** argv) {
 id theTopLevelSwarm ;

 // Swarm initialization: all Swarm apps must call this first.
 initSwarm(argc, argv);

 // swarmGUIMode is set in initSwarm(). It's set to be 1 if you
 // typed heatbugs -batchmode. Otherwise, it's set to 0.

 if (swarmGUIMode == 1)
 // We've got graphics, so make a full ObserverSwarm to get GUI objects
 theTopLevelSwarm = [HeatbugObserverSwarm create: globalZone];
 else
 // No graphics - make a batchmode swarm and run it.
 theTopLevelSwarm = [HeatbugBatchSwarm create: globalZone];

 [theTopLevelSwarm buildObjects];
 [theTopLevelSwarm buildActions];
 [theTopLevelSwarm activateIn: nil];
 [theTopLevelSwarm go];

 // theTopLevelSwarm has finished processing, so it's time to quit.
 return 0;
}
```

main() calls initSwarm (required in all Swarm applications). It then detects if it should do graphics or not, creates the appropriate top level Swarm to contain the model, and sets it to running. Simple as that!

Conclusion

Swarm tries to help computer simulation authors by making it easier to write simulations by making it more formal. The text above gives a narrative introduction into using Swarm for your own models, but a real understanding of Swarm will only come when you start to read through

Installing Swarm

Installing Swarm will take some time: various external libraries need to be acquired, compiled, and installed, and then Swarm itself needs to be compiled. Please report any problems during installation back so that we can try to fix them in the future. Also we've tried to compile some of the common obstacles to installation in the Swarm FAQ

(faq/FAQ.html)
.

- Supported Systems
- Prerequisite Programs
- Prerequisite Libraries
- Configuring Swarm Libraries
- Compiling Swarm Libraries
- Compiling and Running Swarm Applications
- Post-compile cleanup

Supported Systems

Our intention is that Swarm will run on any modern Unix system. Ideally, Swarm itself should be 100% portable. Swarm is currently known to be running on SunOS 4.1.3, Linux, HP/UX, some SGI Irix boxes, MachTen/68k, and Solaris. Some of the software Swarm uses might not be entirely portable; these problems are largely out of our hands. The major difficulty so far is the tricks the GNU Objective C runtime uses may not work properly on AIX and MachTen/PPC, so Swarm may not work on those systems. Fixing this is a longterm project; if you absolutely must use Swarm on AIX or MachTen/PPC, contact us.

> See the Platform News
> (http://www.santafe.edu/projects/swarm/platforms.html)
> web page for more dynamic information.

Prerequisite Programs

We assume you already have Unix and X Windows on your system: the rest of this software is freely available. Expect to spend some time compiling and installing these packages. URLs on this page are to the home distribution site: as a convenience, the Swarm ftp site ftp://ftp.santafe.edu/pub/swarm/

(ftp://ftp.santafe.edu/pub/swarm/needed-software/)
contains copies of the necessary packages in needed-software/
(ftp://ftp.santafe.edu/pub/swarm/needed-software/)
.

. Swarm works with Tcl/Tk versions 7.4/4.0, 7.5/4.1, and 7.6/4.2. It may work with later versions, but it hasn't been tested.

BLT-2.3
: BLT is an add-on to Tk that provides more widgets. BLT is available from http://www.tcltk.com/blt/index.html (http://www.tcltk.com/blt/index.html) or ftp://ftp.neosoft.com/pub/tcl/sorted/devel/ (ftp://ftp.neosoft.com/pub/tcl/sorted/devel/) . Note that BLT 2.3 works with either Tcl/Tk 7.5/4.1 or 7.6/4.2, whereas BLT 2.1 only works with versions up to and including Tcl/Tk 7.5/4.1. There are problems with using BLT 2.3 in that some of the functionality has changed and non-fatal errors may spit out on the terminal window you start Swarm in. But, we haven't taken the time to debug these things.

tclobjc-1.2
: Glue between TCL and Objective C. This version of the library is currently only available from the Swarm authors, and is distributed (http://www.santafe.edu/projects/swarm/release.html) in the same place as Swarm.

Configuring Swarm Libraries

Once you have all of the above software installed, the next step is to configure Swarm. First unpack the Swarm .tar.gz file into a convenient place (via `gzip -dc swarm-xx.yy.zz.tar.gz | tar xf -`).

> All configuration of Swarm compilation is localized to the file `Makefile.conf`. You should change the lines OTHERINCDIRS and OTHERLIBDIRS to point to the directories where the libraries you installed are, and set OTHERLIBS to link in other OS-dependent libraries that might be required on your system. In addition, if your OS doesn't have `ranlib`, then configure the `RANLIB=` line to just execute `/bin/true`.
>
> All of the procedures run by make are defined in the three top level makefiles `Makefile.conf`, `Makefile.lib`, and `Makefile.appl`. If you run into problems, you might need to look at the Makefile code in there.

Compiling Swarm Libraries – (not required for binary distributions)

You've done the hard part, now type "make" and watch the fun! Make will recursively run make in subdirectories, building libraries, installing header files in `include` and libraries in `lib`. If running "make" in the root Swarm directory does nothing, you probably aren't running GNU Make.

Overview of the Swarm Distribution

Swarm is currently in beta test release. Installation directions can be found in install.html

(install.html)
. Documentation is incomplete: learning to program with Swarm will require reading example applications and looking at the header files in libraries. All existing documentation is indexed in index.html
(index.html)
.

Libraries

The main part of Swarm is a set of libraries, one per source directory. Briefly, the directory organization is as follows:

lib, include, bin

Installed libraries, include files, and helper binaries.

src

Swarm library sources.

defobj
(src/defobj.html)
: Support of Swarm style Objective C programming

collections
(src/collections.html)
: Collection library -- various objects that "collect" other objects

activity
(src/activity.html)
: Schedule execution mechanism -- the "process"

objectbase
(src/objectbase.html)
: Base class for simulated objects -- the "agents"

space
(src/space.html)
: 2d spatial environment libary

random
(src/random.html)

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

Last modified: 1998-01-08

Debugging Tips for Swarm

Debugging software is an art, perhaps even more so than writing the software itself. All software we write will have bugs: it is important to know how to diagnose a bug when it happens and how to write code defensively so you have less bugs in the first place.

Finding bugs

Bugs come in two categories: those that crash your program and those that don't. Bugs that crash your program are friendlier because they're obvious. Bugs that you don't notice are much more dangerous: one nagging question in every programmer's mind should be "is the program doing what I think it is doing?"

gdb

By far the most useful tool for finding bugs is a good debugger, a shell you can run a program under and set breakpoints, inspect the values of variables, etc. The best free debugger for Unix is probably gdb, available from the GNU ftp site

(ftp://prep.ai.mit.edu/pub/gnu/)
. gdb seems unfriendly and confusing at first, but it is definitely worth your time to learn it.

> The most important gdb commands are `help`: browse online help, `where`: show me a stack trace, where did we crash?, `list`: show me the source code where we are, `break`, set a breakpoint, and `print`: show me the value of of some expression. If your program is crashing on you, run it under gdb and look at the stack trace. If it looks to be buggy but you don't know where, start setting breakpoints and see when things go awry.

gdb and Objective C

Unfortunately, at this time gdb does not directly support Objective C. There are some workarounds

(http://www.cons.org/cracauer/objc-hint-gdb.html)
that make debugging Objective C programs possible. They are based on the knowledge that Objective C is little more than a glorified syntax for structs (objects) and strange function names (methods).

defobj: xprint(), xprintid(), xfprint(), xfprintid(), xexec(), and xfexec()

Swarm also has a few functions defined that can be used to make debugging easier. In particular, the function xprint(object) prints out the class of an object, and xexec(object, "message") calls the message specified on the object. These can be invoked under gdb as `call xprint(aHeatbug)`.

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

Last modified: 1998-01-08

```
      '(
;; function decls are expected to have types on the previous line
("^\\(\\(\\w\\|[$_]\\)+::\\)?\\(\\w\\|[$_]\\)+\\s *\\(\\(\\w\\|[$_]\\)+\\s *((\\|(\\)[^)]*)+" nil defun)
("^\\(\\(\\w\\|[$_]\\)+[ \t]*::[ \t]*\\)?\\(\\(\\w\\|[$_]\\)+\\|operator.*\\)\\s *\\(\\(\\w\\|[$_]\\)+\\s *((\\|(\\)[^)]*)+" nil defun)
;; nelson - objective c style defuns
("^[-+].*{" nil defun)

("^\\(template\\|typedef\\|struct\\|union\\|class\\|enum\\|public\\|private\\|protected\\).*$" nil decl)
;; datatype -- black magic regular expression (nelson - add id)
("[ \n\t({]\\(\\(const\\|register\\|volatile\\|unsigned\\|extern\\|static\\)\\s +\\)*\\(\\(\\w\\|[$_]\\)+_t\\|float\\|double\\|void\\|char\\|short\\|int\\|unsigned\\|long\\|id\\|FILE\\|\\(\\(struct\\|union\\|enum\\|class\\)\\([ \t]+\\(\\w\\|[$_]\\)*\\)\\)\\)\\(\\s +\\*+)?\\|[ \n\t;()]\\)" nil type)

;; Objective C stuff: ^@[word] gets keyword colouring
("^@\\sw*" nil keyword)
;; key words
("[^_]\\<\\(return\\|goto\\|if\\|else\\|case\\|default\\|switch\\|break\\|continue\\|while\\|do\\|for\\|super\\|self\\)\\>[^_]"
 1 keyword)))))
```

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

Last modified: 1998-01-08

(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

Library Interface Conventions

Overview

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

(../defobj.html)
. 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 "package." (Note: some of the newer Swarm libraries, such as objectbase and random, don't undergo this special processing and yet still follow all the library interface conventions described in this document.)

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

syntax. A key difference from protocols is that object types are published as real external objects that may be used at runtime to create instances of a type. A futher 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 library 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
```

automatically generated.

```
#import "grid.xt"
```

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

also discuss tradeoffs considered along with references to related or supporting work.

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

Last modified: 1998-01-08

The key requirement for Swarm is to be able to make all these operations happen to any object at any time, whenever they're supposed to occur within some simulated world. It stores these operations inside its own data structures, and when you run a simulation in Swarm, the Swarm system itself traverses these data structures to make the necessary operations happen at the proper time.

Swarm needs to be able to make any kind of action happen to any kind of object at any time, and to do this it needs very general-purpose structures that hold some kind of representation of the actions themselves. That's a special requirement that not every object language provides. You already have enough work to define the kinds of objects that can exist inside your model, and Swarm doesn't want you to have to define an even larger number of objects for every kind of event that might occur to your objects. Instead, it wants the object language to provide it with a representation of the operations on objects that you've already defined. Swarm can then store these representations in its own data structures, and make the operations happen whenever it needs to.

In Swarm, the representation of events that happen on objects are just as fundamental to the model as the objects themselves. That's why Swarm is a "discrete-event" simulation system. But if the object system doesn't provide a general-purpose enough representation of events as well as objects, there would be a lot more work to do. The Objective C system, through its special data type called a "message selector," provides a representation of operations that is flexible enough to do this, but the C++ language does not. (In C++, the compiler requires more information about an operation than the event structures in Swarm would be able to provide.)

There's a host of additional reasons why Objective C has also been a good match for the requirements of the Swarm system. Like C++, the operations on objects can be compiled to a very efficient form (though Objective C does require a little more overhead than C++ to get the operation started). This efficiency can be very important for a simulation, since simulations can run for very long periods of time to explore all the behavior that might occur within their simulated worlds. Other languages, such as Lisp, Smalltalk, and Java, also have the "dynamic message dispatch" feature that would make general-purpose event structures possible, but they still carry significant added overhead compared to C.

Objective C is also a very simple extension to the C language. Basic knowledge of C is already widespread, and a few days are typically all that is needed to learn the few additions of Objective C. (Really learning the concepts of objects, however, can take much longer, no matter what object language you try to use.)

Objective C was also a good choice for Swarm because it has a high quality, freely distributable implementation in the GNU C compiler. One of the main concerns about Objective C is that it isn't nearly as widely known or used as other languages like C++, but at least the GNU C compiler assures it will be available on machines where Swarm needs to run. The OpenStep system of the NeXT corporation, and the parallel GNUStep project, also help assure that Objective C is a living language.

There are even more powerful aspects of Objective C that Swarm takes advantage of. Some of these are described in the Advanced Usage Guide
(adv.html)

Some of these rules and guidelines don't even require any specific software to implement them, but are just conventions and recommended style that anyone programming in Objective C can follow when their goals are similar to those of Swarm, and to a large extent often do. These kinds of conventions are the focus of this section; following sections discuss aspects of Swarm Objective C style that depend on specific software support.

Lots more to come... For now, see Library Interface Conventions (library.html) for some of this information, and Swarm tutorials for much of the rest.

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

Last modified: 1998-01-08

Defobj Library: Interface Reference

Dependencies

The defobj library is defined and documented using the library interface conventions

(library.html)
established by the defobj library. It imports the Object superclass and other standard type definitions of the GNU Objective C runtime system.

The collections library must always be linked along with the defobj library. Even though the interface definitions of defobj do not depend directly on collections, the implementations of these libraries both require the presence of the other. Only the collections library should be initialized directly; initialization of the collections library automatically initializes the defobj library as well.

Contents

Name

DefinedObject – standard support for all objects

Synopsis

```
//
// DefinedObject -- object with defined type and implementation
//
@deftype DefinedObject
- getZone;
- (BOOL) respondsTo: (SEL)aSel;
- getClass;

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

- (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;

- (void) setDisplayName: (const char *)displayName;
- (const char *)getDisplayName;

- (void) describe: outputCharStream;
- (void) xprint;
- (void) xprintid;
@end
```

Description

```
DefinedObject is the top-level supertype for all objects that follow the object programming
conventions of the defobj library.  The messages defined by this type are the only messages
which should be assumed to be automatically available on objects that follow these conventions.
In particular, use of messages defined by the Object superclass of the GNU Objective C runtime
should not generally be assumed because future implementations of some objects might not give
continued access to them.
```

from Object unnecessary. They are currently the only messages defined in Object that are recommended for use in Swarm.

```
- (void) setDisplayName: (const char *)displayName;
- (const char *)getDisplayName;
```

These messages assign a character string as a name that identifies an object for display or debug purposes. A global function, xsetname(), is also available to set the name by calling the function from the debugger.

```
- (void) describe: outputCharStream;
- (void) xprint;
- (void) xprintid;
```

The describe: message prints a brief description of the object for debug purposes to the object passed as its argument. The object passed as the outputCharStream argument must accept a catC: message as defined in String and OutputStream in the collections library. Particular object types may generate object description strings with additional information beyond the built-in default, which is just to print the hex value of the object id pointer along with the name of its class, and the display name of the object, if any. xprint prints the standard describe string to standard output; it is also called by the xprint function that may be called from the debugger. xprintid prints just a one-line describe string, consisting of the built-in default.

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

Last modified: 1998-01-08

) for standard ways in which create messages that combine several steps can be combined.

Inherited behavior

The Create type has no inherited behavior beyond the universal messages of the DefinedObject

(DefinedObject.html)
type.

Defined behavior

    + create: aZone;

The create: message creates a new instance of a type with default options. The zone argument specifies the source of storage for the new object. The receiving object of this message is a previously defined type object. The message is declared as a class message (with a + declaration tag) to indicate that the message is accepted only by the type object itself rather than an already created instance of the type (which a - declaration tag otherwise defines).

The create: message returns the new object just created.

Defined behavior

    + create: aZone;

The create: message creates a new instance of a type with default options. The zone argument specifies the source of storage for the new object. The receiving object of this message is a previously defined type object. The message is identified as a class message (a + declaration tag) to indicate that the message is accepted only by the type object itself rather than an already created instance of the type, which most messages of a type (those with a - declaration tag) otherwise define.

The create: message returns the new object just created. This object is an instance of some class selected to implement the type. The class which a type selects to implement an object may be obtained by the getClass message, but is not otherwise visible to the calling program. A caller never refers to any class name when creating objects using these messages, only to type names, which are automatically published as global constants from any @deftype declaration.

    + createBegin: aZone;
    - createEnd;

The createBegin: and createEnd messages bracket a series of messages that specify options for an object being created. The intermediate messages can set values defined as parameters of the type, or provide other forms of specification using available messages. A particular object type defines the specific messages that are valid for sending during this interim creation phase.

objects all need the same create-time options, it is often simpler (and can also be faster) to create a customized version of a type first, and then create further instances from that type.

Customizing a type object does not modify the original type object, but instead creates a new type object that has the customizations built-in. A create: message on the new type object creates a new instance as if the same sequence of create-time messages had been sent to the original type object using createBegin: and createEnd. A type is customized by bracketing the sequence of create-time messages not with the createBegin: and createEnd messages used to create a new instance, but with customizeBegin: and customizeEnd messages instead. customizeBegin: returns an interim value for receiving create-time messages much like createBegin:, and customizeEnd returns the new, customized version of the original type. Following is sample code sequence that illustrates the use of these messages:

```
newArrayType = [Array customizeBegin: aZone];
  [newArrayType setCount: 100];
  newArrayType = [newArrayType customizeEnd];
  array1 = [newArrayType create: aZone];
  array2 = [newArrayType create: aZone];
  ... // [array1 getCount] and [array2 getCount] are both 100
```

The customizeCopy: message creates a new copy of the interim object returned by customizeBegin: which may be used for further customizations that do not affect the customization already in progress. It may be used to branch off a path of a customization in progress to create an alternate final customization. Following is a code fragrment which illustrates this usage:

```
newArrayType1 = [Array customizeBegin: aZone];
  [newArrayType1 setCount: 100];
  newArrayType2 = [newArrayType2 customizeCopy: aZone];
  [newArrayType2 setDefaultMember: UnsetMember];

  newArrayType1 = [newArrayType1 customizeEnd];
  newArrayType2 = [newArrayType2 customizeEnd];
  array1 = [newArrayType1 create: aZone];  // no DefaultMember option
  array2 = [newArrayType create: aZone];   // DefaultMember option set
```

customizeCopy may be used only on an interim object returned by customizeBegin: and not yet finalized by customizeEnd. The new version of the interim object being customized may be allocated in the same or different zone as the original version, using the zone argument required by customizeCopy:

The zone passed to customizeBegin: is the same zone from which storage for the new, finalized type object will be taken. This zone need not be the same as any instance later created from that type, since a new zone argument is still passed in any subsequent create message on that type.

Whether a customized version of a type can be created depends on the implementation of the type itself. If a type does not support customization, a customizeBegin: message on the type raises an error. All types defined by an @deftype declaration may be relied on to support at least one cycle of customization to create a new type object. Whether an already customized type object (returned by customizeEnd) supports a further

Name

Drop – end existence of an object

Synopsis

```
//
// Drop -- end existence of an object
//
@deftype Drop
- (void) drop;
@end
```

Description

The Drop supertype defines the drop message, which is a standard message for indicating that an object no longer exists and will never again be referenced. Any future attempt to reference a dropped object is an error. This error may or not produce predictable effects depending on the level of debug checking and other factors.

Immediate effects of the drop message depends on the subtype of Zone (Zone.html) used to provide storage for the object. For some zone types, the drop message immediately deallocates storage for the object and makes the freed storage available for other use. Subsequent use could include the allocation of a new object at precisely the same location, resulting in a new object id identical to a previously dropped one. Other zone types (..none implemented yet..) might delay any reclamation of freed storage until a more comprehensive check can be made that other references no longer exist to the dropped object. When running in debug mode (see debug support (globaldef.html#debug)) the storage for a dropped object is filled with predictable contents so that an error is raised if there is any immediate attempt to us it again.

The Drop type may be inherited by any type that provides drop support for its instances. In addition to freeing the storage and invalidating the object, a drop message may release other resources acquired or held within the object. Not every object which can be created can also be dropped, and some objects can be dropped which are not directly creatable. Some objects may be created as a side effect of other operations and still be droppable, and some objects may be created with links to other objects and not droppable on their own. A type independently inherits Create or Drop types, or both, to indicate its support of these standard interfaces to define the endpoints

Name

Zone – modular unit of storage allocation

Synopsis

```
//
// Zone -- modular unit of storage allocation
//
@deftype Zone <Create, Drop, GetOwner, CREATABLE>
CREATING
- (void) setReclaimPolicy: reclaimPolicy;
- (void) setStackedSubzones: (BOOL)stackedSubzones;
- (void) setPageSize: (int)pageSize;
USING
- getReclaimPolicy;
- (BOOL) getStackedSubzones;
- (int) getPageSize;

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

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

- getComponentZone;

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

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

-                getPopulation;
- getSubzones;
- (void) mergeWithOwner;

- (BOOL) containsAlloc: (void *)alloc;
- getSubzone: (void *)alloc;

- (void) reclaimStorage;
- (void) releaseStorage;
```

```
@deftype Zone &lt;Create, Drop, CREATABLE&gt;
```

A zone is created using standard create messages just like other objects. This means that a zone must identify another zone from which it obtains its storage. Storage is typically obtained from this other zone in large units called pages, which are then managed by the local zone to support internal allocations. The getZone message of the Drop
(Drop.html)
type returns the zone which provides these base pages.

Since a new zone always requires that an existing zone be identified, no new zones could be created unless there were some zones that already existed. Several such zones are predefined as part of the defobj library; see predefined zones
(globaldef.html#zones)
for a summary.

Create-time behavior

```
- (void) setReclaimPolicy: reclaimPolicy;
- (void) setStackedSubzones: (BOOL)stackedSubzones;
- (void) setPageSize: (int)pageSize;
USING
- getReclaimPolicy;
- (BOOL) getStackedSubzones;
- (int) getPageSize;
```

ReclaimPolicy

(.. Not implemented yet. Will define the various types of zone implementations, all the way to zones that support full automatic storage reclamation. ..)

StackedSubzones

(.. Not implemented yet. Specifies that no further allocations are made in the zone after a subzone is created, until that subzone itself is dropped. Supports an especially efficient of implementation. ..)

PageSize

PageSize specifies the size of pages within which a zone manages its internal allocation. Its default is typically a natural page size (perhaps 4K) for the local machine architecture. The default should be overridden only when tuning storage allocation for very specific situations. Allocations within a zone are not limited to the page size, since any requests that exceed the page size are simply passed up to the owner zone within which the zone was allocated.

Object allocation

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

alloc: allocates a new storage block much like the malloc function of the C library. The storage is aligned according to the most restrictive requirements for any data type on the local machine architecture. The storage is not initialized to any known contents.

free: releases a block of storage previously allocated using alloc:. The size of the block is not required as an argument because alloc: has saved this size as necessary as part of the initial allocation. free: may be used only to free a block allocated by alloc:, and a block allocated by alloc: may be freed only by free:.

allocBlock: allocates a new storage block similar to alloc:, except that the size of the block allocated must be passed as an argument when freeing the block. copyBlock:blockSize: allocates a new storage with the same requirements on free as allocBlock:, but also fills the new storage with the contents of a block passed as the first argument, up to the size of the newly allocated block. freeBlock:blockSize: must be used to free any block previously allocated by allocBlock: or copyBlock:blockSize.

Subzone structure

```
-               getPopulation;
- getSubzones;
- (void) mergeWithOwner;

- (BOOL) containsAlloc: (void *)alloc;
- getSubzone: (void *)alloc;
```

getPopulation returns a collection all objects allocated in a zone using either allocIVars: or copyIVars: and not yet freed using freeIVars:. getObjects returns nil if the ObjectCollection option is false. The collection returned has the type OrderedSet
(../collections/OrderedSet.html)
as defined in the collections library
(../collections.html)
, with the ReadOnly option set true and the IndexSafety option set to SafeAlways. The members of this collection may change as objects are allocated and freed, but may not added or removed directly within the collection.

getSubzones returns a collection of other zones that all obtain their storage from the local zone. mergeWithOwner drops the local zone, but reassigns all the zone allocations to the zone from which all local storage was obtained. (..neither message currently implemented..)

containsAlloc: tests if a particular allocation was made by the local zone, or any zone which obtains storage from that zone. The pointer argument must point to an allocation previously made by any zone. The message returns true if the allocation was made by either the local zone or any of its subzones. getSubzone is similar, except that it returns the particular zone in which the pointer was allocated, or nil if the pointer was not allocated in the local zone or one of its subzones. (..neither message currently implemented..)

Name

Symbol – object defined as a distinct global id constant

Synopsis

```
//
// Symbol -- object defined as a distinct global id constant
//
@deftype Symbol <Create, GetName, CREATABLE>
CREATING
+ create: aZone setName: (const char *)name;
- (void) setName: (const char *)name;
@end
```

Description

A Symbol is an object created with a fixed name. It has no behavior except to get the name with which it was created. A Symbol is typically used to define unique id values which are assigned to global constant names. These names, capitalized according to the recommended convention for global object constants, are used by some libraries as flags or enumerated value codes in arguments or return values of messages.

Ordinarily, a symbol is created with its character string name matching the global id constant to which it is assigned. These global program constants can then provide a minimal level of self documentation as objects. Subtypes of Symbol can extend the base of a named, global id constant to establish further components of a global, constant definition.

Inherited behavior

A symbol is fully creatable using standard Create messages. A character string name must be supplied for any new symbol; there is no default. Symbol inherits the getName

(misc.html#GetName)
message, which returns the symbol name.

Name

Warning, Error – report of some condition occurring during program execution

```
(..  This section of documentation is only a start.)
```

Synopsis

```
//
// EventType -- report of some condition occurring during program execution
//
@deftype EventType <Symbol, RecordType>
- (void) raiseEvent;
- (void) raiseEvent: (void *)eventData, ...;
@end

//
// Warning -- a condition of possible concern to a program developer
//
@deftype Warning <EventType, CREATABLE>
- (void) setMessageString: (const char *)messageString;
- (const char *)getMessageString;
@end

//
// Error -- a condition which prevents further execution
//
@deftype Error <Warning, CREATABLE>
@end
```

Description

```
Warning and Event are object types (both subtypes of EventType) that may be used to report
conditions detected during program execution.  A warning is a condition of possible interest
to a developer or user, but one that does not prevent further execution.  An error is a condition
which prevents further execution of the program.
```

```
Warning and Event also inherit from Symbol, so that each type of warning or error
may be given a unique global name.  Any library may publish its own warning and event
names as part of its documented library interface (see module definition conventions
```

```
  OutOfMemory,            // no more memory available for allocation
  InvalidAllocSize,       // no more memory available for allocation
  InternalError,          // unexpected condition encountered in program
  BlockedObjectAlloc,     // method from Object with invalid allocation
  BlockedObjectUsage;     // method inherited from Object superclass

//
// standard warnings
//
extern id <Warning>
  WarningMessage,         // message in the source defines warning
  LibraryUsage,           // invalid usage of library interface
  DefaultAssumed,         // non-silent use of default
  ObsoleteMessage,        // using feature which could be removed in future
  ObsoleteFeature,        // using feature which could be removed in future
  ResourceAvailability;   // resource from runtime environment not available
```

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

Last modified: 1997-01-08

```
@end
```

Description

The copy message creates a new object that has the same contents and resulting behavior as a starting object, except that an independent copy of the contents of the starting object is created so that further changes to one object do not affect the other. The zone argument specifies the source of storage for the new object. The message returns the id of the new object created.

> An object type that supplies the copy operation defines what it includes as the contents of an object copied. There is no global rule for what is considered "inside" a copied object vs. merely referenced by it. (There is no fixed notion of "shallow" vs. "deep" copy found in some object libraries.) After copying, the new object may still contain some references to other elements also referenced by the starting object, but in general the new object minimizes any dependencies shared with the starting object. Any object type supplying the copy message should also supply documentation on its rules for copied objects.

GetName Interface Description

Synopsis

```
//
// GetName -- get name which identifies object in its context of use
//
@deftype GetName
- (const char *) getName;
@end
```

Description

The getName message returns a null-terminated character string that identifies an object in some context of use. This message is commonly used for objects that are created once in some fixed context where they are also assigned a unique name. Constant objects defined as part of a program or library are examples. This message is intended only for returning a name associated with an object throughout its lifetime. It does not return any data that ever changes.

GetOwner Interface Description

Description

The SetInitialValue type defines a variety of messages relating to an initial or unmodifiable value established as part of an object. This message is typically provided when creation of a new object might be more easily accomplished by copying the value of an existing object rather than establishing a new value from scratch. As with the copy message, precisely what is considered the value of an existing object to copy is defined only by the particular object type that supplies these messages.

If an object has a value which can be established at create time, it is often useful (and can also enable significant optimization) to declare that no further modification will occur to this value during further use of the object. A restriction against modifying a value is referred to as a "read-only" restriction. This type supplies messages to declare a read-only restriction along with any initial value. For some object types, a read-only restriction can also be added or removed after an object has already been created.

The setInitialValue: message requires another object as its argument, from which the value of a newly created object is to be taken. Unlike a copy message, the object used as the source of the new value need not have the identical type as the new object to be created. A particular object type defines the types of initial value objects which it can accept, along with any special conversion or interpretation it might apply to such a value.

The ReadOnly flag specifies that the value of an object may not be modified further by other messages on an object. If such messages are attempted then an error is raised. If the ReadOnly flag is set at create time, the object is first created in read-only mode. Typically an initial value is always provided for an initial read-only object, since otherwise the created object would have only an empty or initial default value. The setReadOnlyValue: message is equivalent to a combination of a setInitialValue: message with an added ReadOnly restriction.

If a read-only restriction is established at create time, then for some object types this restriction may be required to remain for the entire lifetime of an object. For other object types, the read-only restriction may be added or removed any time after initial object creation. If a particular type does not support a change to read-only status, a request to do so raises an error.

Standard forms of create combination messages
(combo.html)
are provided to create an object in one message expression from the value of another object, with or without a read-only restriction.

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

Last modified: 1998-01-08

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 module 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 module, 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 module, 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 module. The library modules of Swarm mostly follow this structure.

Object Type Definitions

The interfaces to objects defined by a library module 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 syntax. A key difference from protocols is that object types are published as real external objects that may be used at runtime to create instances of a type. A futher

```
#import "grid.xt"
```

An object type is defined by an @deftype declaration. 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 module header file: This line adds no information to the header file, but is required as part of the system by which definitions are automatically generated.

```
#import "grid.xt"
```

Documentation Structure

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

- 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.
- 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 Notes
  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.
- Subclassing Notes
  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 also discuss tradeoffs considered along with references to related or supporting work.

Marcus G. Daniels <mgd@santafe.edu>

Defobj Library: Create Combination Messages

The create phase of an object type may contain messages that are defined simply by identifying them as a combination of other defined messages. Such messages are referred to as "create combination messages." Typically, such messages have the following form:

```
-                create: aZone setOption: optionValue;
```

For this message to be a valid create combination message, the object type must already have the create: and setOption: messages defined (either locally or by inheritance). The meaning of this message is then identical to the following sequence (where SomeType is the name of a type implementing the combination message):

```
-   anObject = [SomeType createBegin: aZone];
-   [anObject setOption: optionValue];
-   anObject = [anObject createEnd];
```

Create combination messages are provided as a convenience for commonly used options. Some combination messages also have their own custom implementing methods so that creation is faster when using the combination message. Any such implementation, however, is required to implement the parts of the combination message so that its meaning is identical to the results that would be obtained if the separate parts were requested in separate message.

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

Last modified: 1998-01-08

Defobj Library: Global Definitions

Contents

The following global definitions are documented below:

- M() macro
- func_t type definition
- initModule() macro
- global zones
- output files
- debug support

M() macro

Synopsis

```
//
// M() -- macro to abbreviate @selector()
//
#define M( messageName )  @selector( messageName )
```

The M() macro is currently nothing more than an abbreviation for the Objective C syntax used to obtain a selector for a particular message name. It is provided as a convenience for use in messages that combine a selector with other arguments.

func˙t type definition

Synopsis

```
//
// func_t -- function pointer type
//
typedef void (*func_t)( void );
```

Output Files

Synopsis

```
extern FILE *_obj_xerror;     // output file for error messages
extern FILE *_obj_xdebug;     // output file for debugging messages
```

The file pointers _obj_xerror and _obj_xerror define standard output files used for error messages and debugging messages, respectively. If not set otherwise by the time of defobj module initialization, they both default to the standard error output defined by the C library.

Debug Support

Synopsis

```
extern BOOL _obj_debug;       // if true then perform all debug error check-
ing

extern void xsetname (id anObject, const char *name);  // debug set dis-
play name

extern void xprint (id anObject);              // debug object print
extern void xprintid (id anObject);            // debug object id print

extern void xfprint (id anObject);             // debug foreach object print
extern void xfprintid (id anObject);           // debug foreach id print

extern void xexec (id anObject, const char *name);  // debug method exec
extern void xfexec (id anObject, const char *name); // debug fore-
ach method exec
```

The _obj_debug variable enables debug-level error checking in libraries that support optional levels of checking. Its default value is true. The default may be overridden either before initialization of the defobj module, or by means of a compiler macro setting (-D_obj_debug=0). If set to zero by a compiler macro setting, optimized compilation of code that refers to it will see only the constant zero and so avoid even generating code to test the variable.

The xsetname function sets the display name for an object using its setDisplayName: message. This string may be used to identify the object in a user-meaningful way; if defined it will appear in the identification string generated by a standard describe: message.

Defobj Library: Subclassing Reference

The defobj library currently includes the CreateDrop and Object_s superclasses used by other libraries. For user objects in Swarm simulations the SwarmObject superclass in the objectbase
(../objectbase.html)
library packages all required behavior of these superclasses into a single, simpler superclass.

The defobj library also provides support for custom-generated classes, such as those which implement separate phases of the standard create protocol. (See generated phase classes
(adv.html#phase-classes)
in the Advanced Usage Guide
(adv.html)
.) Subclassing from custom-generated classes is still not officially supported, because the framework for custom class generation is still being finalized. Custom-generated classes are currently used only in the defobj, collections, and activity libraries; these libraries document the specific classes that provide subclassing support.

In general, classes that implement a library are not automatically available for use as superclasses. Any library must document those specific superclasses that are valid for user classes to subclass from, along all the rules that a user subclass must follow when implementing new behavior.

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

Last modified: 1998-01-08

Defobj Library: Implementation Notes

(This section of documentation is not yet available.)

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

Last modified: 1998-01-08

(mailto:mgd@santafe.edu)

Last modified: 1998-01-08

all basic capability. A second priority is to complete the complementary "Usage Guide" documents. Unlike the reference documents, the Usage Guide will have a task-oriented organization, and will lead the initial user through actual code examples in the rough order a typical user is likely to need them. It will serve the role of a tutorial on each library.

The Usage Guide code examples have not yet been developed. For the time being, a directory of test programs (GridTurtle test programs (../grid.html) , contained within the documentation release directory) provides code examples of many of the basic features of the defobj, collections, and activity libraries. These code examples also help indicate the portions of the libraries which are fully implemented and working, since they are run on each new release of these libraries.

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

Last modified: 1998-01-08

Collections 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

```
   /       \        /           \
Stack
 (Stack.html)
    Queue
 (Stack.html)
    Set,
 (Set.html)
         Map,
 (Map.html)

            SetIndex
 (Set.html)
     MapIndex
 (Map.html)

                             |

          OrderedSet
 (OrderedSet.html)
```

(need a graphic with image map, but this will do for now)

Contents:

Name

Collection - supertype of all collection types

Synopsis

```
//
// Collection -- supertype of all collection types
//
@deftype Collection <Create, SetInitialValue, Copy, Drop>CREATING
- (void) setMemberType: aDataType;
- (void) setIndexSafety: indexSafety;
- (void) setIndexUpdateHandler: (func_t)handler : (void *)arg;
- (void) setReplaceOnly: (BOOL)replaceOnly;
USING
- getMemberType;
- getIndexSafety;
- (func_t) getIndexUpdateHandler: (void *)arg;
- (BOOL) getReplaceOnly;

- (int) getCount;

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

- getFirst;
- getLast;

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

- begin: aZone;

- (void) forEach: (SEL)aSelector;
- (void) forEach: (SEL)aSelector : arg1;
- (void) forEach: (SEL)aSelector : arg1 : arg2;
- (void) forEach: (SEL)aSelector : arg1 : arg2 : arg3;
@end
```

Member types

The default member type of any collection is the id type defined by Objective C. All messages which add, remove, or access members declare their member value with the id type. None of the collection types, however, requires that the member value actually point to a valid object. The member values of any collection may contain an arbritrary bit pattern, provided these bits fit in the space available within an id pointer.

In the few cases when default operations assume that a member points to a valid object, there are alternate ways to avoid these defaults and use an arbritrary bit pattern as a member value instead. When using an arbritrary bit pattern, the member value must be cast to or from the id type when adding or retrieving it from the collection. By global portability assumptions
(../global-port.html)
, the number of bits in an id value may be safely assumed to be at least as large as an int (signed or unsigned), but not necessarily as large as a long.

There is also an option to override the default member type so that it can be larger than a id pointer. Use of this option, however, changes the way member values must be passed to or from a collection. More information is given with the MemberType option below.

Inherited behavior

@deftype Collection <Create, SetInitialValue, Copy, Drop>
(.. SetInitialValue and Copy behavior is currently implemented for Array and List, but not for Set or Map.)

The Collection type inherits standard messages from the defobj library for creating and customizing objects. These messages are defined as part of the defobj Create supertype. The only collection types which are directly creatable, however, are those directly tagged as such (with the CREATABLE type tag).

In addition to the standard create messages inherited from the defobj Create supertype, Collection inherits the SetIntialValue, Copy, and Drop types (also from defobj). Copying or dropping a collection has no effect on objects which might have be contained in a collection as members; only the references to objects contained in the collection are copied or freed.

SetInitialValue defines special create messages to load the initial members of a newly created collection from an existing collection, and optionally to make the members of a collection read-only after this initialization. If the members of a collection are read-only, all collection messages which add, replace, or remove members will raise an InvalidOperation error. Even if a collection was not initially created with ReadOnly status, this status may also be set later. If a collection is created with the ReadOnly restriction set, however, this restriction cannot later be unset.

A collection used as an initial value need not have the same type as a collection

IndexSafety

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

The IndexSafety option has one of the following values:

```
id <Symbol>  Unsafe, UnsafeAtMember, SafeAlways;
```

If not specified, the default value depends on the specific collection subtype. In typical cases, the default value is Unsafe. A non-default value is required for indexes to have a guarantee of safety when independent modifications to the underlying collection are also performed.

Without a guarantee of index safety, the effect of any operation which modifies the underlying collection (adding, removing, or replacing members) could be to disrupt the position assumed by any existing index. The effect of such disruption is, in general, completely unpredictable: it could be an immediate crash, a wrong sequence of members, or other difficult-to-debug conditions. To avoid such unpredictability, several means for assuring the safety of indexes under independent collection update are provided.

The simplest way to guarantee index safety is to set the IndexSafety option to the value SafeAlways when a collection is created. With the value of SafeAlways, any active index on the collection will be updated automatically to reflect any change to the underlying collection whenever such a change occurs. This automatic update continues to occur throughout the lifetime of any index on the collection; it ceases to occur only after an index is dropped. This is the easiest option to use, but consumes extra resources to accomplish (both time and space within the collection), so is most suitable when the extra resource usage deesn't matter or there's no better way.

With an IndexSafety value of SafeAlways, there is also an additional option to declare an external index update handler. The handler is specified using the setIndexUpdateHandler:: message. If an IndexUpdateHandler is specified, the value for IndexSafety is also automatically set to SafeAlways.

The first argument of setIndexUpdateHandler:: specifies a function which is called whenever a modification to the underlying collection occurs which could require an update to existing indexes. The second argument establishes a pointer value which the handler function may use to identify the indexes which need to be updated, or any other information needed for the update.

(.. Exact call conventions for the IndexUpdateHandler are still being established, but the activity library will soon be converted to use this technique, so the conventions will be resolved during that conversion.)

Besides the extreme values of Unsafe and SafeAlways, another valid value for IndexSafety is UnsafeAtMember. This value indicates that the only modification of the collection which could disrupt existing indexes is to remove the member contained at the current index position, or to change the member on either side of an index positioned between members. For some collection subtypes, this level of safety is automatically assured, and for others it can be requested with less overall resource impact than SafeAlways.

sequentially through all members of the collection until a matching member is found. For other subtypes, some form of direct indexing from the member value may be supported. The message is supported regardless of its speed.

The remove: message removes the first member in the collection with a value matching the value passed as its argument. If there is no such member, a nil value is returned. As with the contains: message, the speed of this operation may vary from very low to linear in the number of members, depending on the collection subtype.

The removeAll message removes all existing members of a collection and sets its member count to zero. The collection then remains valid for further members to be added. This message has no effect on the objects which might be referenced by any removed member values. If resources consumed by these objects also need to be released, such release operations (such as drop messages) can be performed prior to removing the member values.

```
- begin: aZone;
```

The begin: message is the standard method for creating a new index for traversing the elements of a collection. All further information about indexes is documented under the Index type.

```
- (void) forEach: (SEL)aSelector;
- (void) forEach: (SEL)aSelector : arg1;
- (void) forEach: (SEL)aSelector : arg1 : arg2;
- (void) forEach: (SEL)aSelector : arg1 : arg2 : arg3;
```

The forEach messages supply a convenient shorthand for repeatedly performing the same message on all objects contained as members in a collection. The message to be sent is identified by the argument aSelector. This selector must define the same number of arguments as contained in any remaining argument slots of the forEach message. The argument types of the message to be sent must be either the id type, or some other type that will fit in the same number of bits as the id type. By global portability assumptions, the argument type could be as large as an int (signed or unsigned), but not necessarily as large as a long. To use the message, any non-id value must be cast to the id type as part of the call expression.

The forEach: messages are implemented by a simple loop through all members of a collection, using an internal, temporary index. If any operation more complex than a simple message send is required, this operation should just be coded directly using a loop that traverses its own index.

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

Last modified: 1998-01-08

Many indexes provde the ability modify the collection they refer to, in addition to simply traversing members. An index often provides the complete means for maintaining the contents of a collection, more than could otherwise be performed on the collection itself. The position or other status of the index is automatically updated to reflect any changes made through the index itself.

If changes to a collection are made while other indexes exist, those other indexes could be affected in potentially catastrophic ways. Each collection type documents which kinds of changes can be made without affecting or invalidating existing indexes. The IndexSafety option of Collection provides additional ways to protect indexes against possible effects of independent updates.

Inherited behavior

```
@deftype Index <DefinedObject, Copy, Drop>
```

Each index is a stand-alone object allocated within a zone passed as an argument in the message that created it. This zone need not match the zone of a collection. It is common for index lifetimes to be shorter than their collection. For example, indexes can be created in a temporary scratch zone for use only within a local loop.

Because messages to a collection are the only valid way to create an index, create messages and create-time options are not used with index types. All valid processing on an index is determined by characteristics of the collection from which it is created. Index types are typically named after the type of collection they are created from, and serve principally to define the specific messages valid for an index on that type of collection.

Index objects support the universal messages of the top-level DefinedObject supertype, along with the standard drop: and getZone messages. Even though they cannot be created except from a collection, new index objects can be created from an existing index using the standard copy: message. Each copy refers to the same collection as the initial index, and starts at the same position in its enumeration sequence. In all other respects, however, the new copy is an independent index that maintains its own position under any further processing.

Index behavior

```
- getCollection;
```

getCollection returns the collection referred to by an index. This collection never changes during the lifetime of the index.

```
- next;
- prev;
```

The next and prev messages are the basic means to traverse through the members of a collection.

The getLoc message returns a symbol constant which indicates the type of location at which an index is currently positioned. This index location symbol has one of the following values:

```
extern id <Symbol>  Start, End, Between, Member;
```

The Start symbol indicates the special position preceding all members in the enumeration sequence for the collection. This is the location at which an index is positioned when it is first created. The End symbol indicates the special position following all members in the collection. This is the location at which an index is positioned just after a next message has returned nil, as a result of moving beyond the last member. The Between symbol indicates the special position between two other members, at which an index is positioned after a current member, between two other members, is removed. The Member symbol indicates that the index is positioned at some current member in the enumeration sequence of a collection.

The getLoc message is needed to traverse a collection which could contain nil values for its members. Without getLoc, there would be no way to distinguish a nil value returned by next as either a valid member value or the special value returned at the end of members. With getLoc, a loop that traverses a collection can test specifically for the end (or start) of members. Following is a simple loop which illustrates such usage: index = [aCollection begin: aZone]; do { member = [index next]; // do something with member ... } while ([index getLoc] == Member); The setLoc: message may be used to reset the current location of an index to either Start or End. It may be used to reprocess a collection using an existing index after some other location has already been reached. It may also be used to position an index at the end of all members prior to traversing members in reverse order using prev.

Besides Start and End, setLoc: accepts the special argument values of BetweenAfter and BetweenBefore, which are also defined symbols. These argument values are only valid if the index is positioned at a current member. They reposition the index to the special location between the current member and its immediately following or preceding member.

```
- (int) getOffset;
- setOffset: (int)offset;
```

Provided there is no major computational cost, an index also maintains the integer offset of its current member within the enumeration sequence of the collection. These integer offset values have the same definition as in the atOffset: messages of Collection. The getOffset message returns this current offset value. If the index is current positioned at the Start or End location, getOffset returns -1. If the index is positioned at a Between location, getOffset returns the offset of the immediately preceding member. If the offset is not currently available from the index, getOffset returns the special value UnknownOffset. This value is defined by a macro as the maximally negative value of a 32-bit, 2's-complement integer.

An offset is always available from an index if its current position has been reached by repeated operations from an initial Start or End position, and there has been no other modification to the underlying collection. Some forms of direct member access operations supported by some index types, however, may result in an integer offset not being available. These restrictions are noted with the individual index type.

Using the setOffset: message, an index may be positioned directly to a member using the offset of the member within its enumeration sequence. The speed of this operation

Name

Array – collection supporting access to fixed group of members

Synopsis

```
//
// Array -- collection supporting access to fixed group of members
//
@deftype Array <Collection, CREATABLE>CREATING
+ create: aZone setCount: (int)count;
+ create: aZone setMemberBlock:(id *)members setCount:(int)count;
SETTING
- (void) setCount: (int)count;
- (void) setDefaultMember: memberValue;

- (void) setMemberBlock: (id *)members setCount: (int)count;
USING
- getDefaultMember;
- (id *) getMemberBlock;
@end
```

Description

An array is a collection of members that are all created as members of the collection at the same time. Existing member values may be replaced with new values, but the members themselves are fixed at constant offsets within the collection. The fixed structure of an array permits very fast access to members by integer offset positions, since the location of each member may be directly calculated.

The Array type is one of the simplest collection types in the collections library, and the closest to a data structure directly supported in C. Unlike C arrays, the group of members belonging to the array is not necessarily fixed for the lifetime of the array, but may be dynamically resized to contain a different number of members. When an array is dynamically resized, existing member values are preserved as much as possible.

The Array type adds few messages to the generic messages inherited from Collection. This type is provided partly so that a fixed-structure array can be accessed with the same uniform set of basic messages as any other kind of object collection. It also handles all required memory allocation within the collection. As an option,

```
- (void *) getMemberBlock;
```

All the Array create-time options can also be set after the array is already created, subject to restrictions noted below.

Count

The Count option sets the number of members which belong to the collection. Any non-negative value including zero is valid. If the array already exists, the any existing members up to the new count will preserve their existing values. If the new count is greater than the existing count, or a new array is being created, all members will be assigned an initial default value of either nil, or a value previously specified for DefaultMember.

The inherited getCount message of Collection obtains the current value set for this option. There is no way to add or remove members of an array except using the setCount: message directly.

DefaultMember

When this option is set, the initial value of all new members will be set to the member value given (otherwise the default is nil). This option gives a convenient way to distinguish members which have never been set from any other valid member value, which could include nil. This option may be reset after array creation only if some setting for the option was given at create time. (The initial, explicitly set value can still be the default nil, but a value must be set explicitly for the option to be resettable later). The get message for this option always retrieves the current setting, but this value has no effect except when the count of an array is increased, so that new members need to be initialized.

MemberBlock

This option provides a means to wrap an existing C array for access as an object collection. If this option is given, dynamic resizing is not supported. The current C array being wrapped, however, can be replaced by giving a new setting for MemberBlock after an array already exists. A new setting can be given only if some setting was initially given at create time.

Even if a setting was not given for MemberBlock at create time, the get message for MemberBlock always returns a pointer to whatever internal memory array is currently being used by the array object. If a MemberBlock setting was given, the pointer returned will be the same as the one previously given. In either case, the pointer returned may be used to manipulate member values in any way desired using native C expressions. After an array has been created, there is no way to determine whether the MemberBlock pointer was established externally or by internal allocation. If the pointer was established by internal allocation, however, the external program must make no attempt to free or otherwise reallocate this memory.

A count must always be supplied with an external member allocation, using the setCount: argument of the compound message. If an external allocation is being used, the only way to reset the count is also to reset MemberBlock; any attempt to use the setCount: message by itself will raise an error. Whenever an external member allocation is

Name

List, ListIndex – collection of members in linear sequence

Synopsis

```
//
// List - collection of members in linear sequence
//
@deftype List <Collection, CREATABLE>CREATING
- (void) setEndsOnly: (BOOL)endsOnly;
USING
- (BOOL) getEndsOnly;

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

- removeFirst;
- removeLast;
@end

//
// ListIndex -- index with insertion capability at any point in list
//
@deftype ListIndex <Index>- (void) addAfter: anObject;
- (void) addBefore: anObject;
@end
```

Description

```
A list is a collection of members that are all maintained at some externally assigned position
in a linear sequence of all members.  The sequence is established by the position at which
members are added:  members can be added at the start of list, at the end, or at any point
in the middle.
```

```
A list is also one of the most dynamic of basic collections, in that the cost of adding
and removing members is very low.  Members can be removed from any position just as
easily as they can be added.  A list automatically grows and shrinks to reflect the
number of members at any one time, and there is no fixed capacity which limits the
size to which a list may grow.
```

ListIndex behavior

```
//
// ListIndex -- index with insertion capability at any point in list
//
@deftype ListIndex <Index>- (void) addAfter: anObject;
- (void) addBefore: anObject;
@end
```

The addAfter: and addBefore: messages add members at a particular point in the sequence of members maintained by a list. The current location of an index determines the point at which a new member will be added. The addAfter: message adds a member at the list position immediately following the current index location. addBefore: adds a member to the immediately preceding location. Neither message changes the current location of the index, except that an index can change from a Start or End location to a location of Between.

Since an index may be positioned to any location in a list, these messages enable the construction of any desired sequence of members. Since the current index location remains unchanged, multiple members may all be inserted successively at some point in a list; previously added members are just pushed out one-by-one as new members are added.

An index with a location of Start, End, or Between is just as valid a location for addAfter: or addBefore: as an index positioned at a member. In these cases, there is no member at the current location of the index, so the new member is just inserted directly at the current index location, and the index is left positioned between the new member and the member that was previously adjacent in the opposite direction. If the previous location was Start and the message addAfter:, or the location was End and the message addBefore:, the index location remains Start or End.

If either the addAfter: or addBefore: message is requested with the EndsOnly option, and the index location is not Start or End, or if remove is requested on an index that is not positioned at either the first or last member, an invalid operation error is raised.

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

Last modified: 1998-01-08

Stack behavior

```
- (void) push: anObject;
- pop;
...
```

Queue behavior

```
- (void) enqueue: anObject;
- dequeue;
...
```

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

Last modified: 1998-01-08

```
- createIndex: aZone at: aKey;
- createIndex: aZone setMember: aMember;

- (void) getPredecessors: aMember;
- (void) getSuccessors: aMember;
- (void) getCostarts: aMember;
- (void) getCoends: aMember;
@end

//
// KeyedIndex -- index behavior shared by Set and Map types
//
@deftype KeyedIndex <Index>
- (int) getCurrentLevel;
- getIndexAtLevel: (int)level;

- at: aKey;

- setMember: aMember;
@end
```

Description

A keyed collection is a collection in which each member can be compared with some other value that identifies the member. This value is referred to as the member key. The key value may be determined either by the member value itself, which defines a Set, or by external association with the member when the member is first added, which defines a Map.

The keyed collection type establishes the common behavior shared by both Set and Map. Standard options are provided to declare ordering of members in the collection, which can be based either on an inherent ordering of the key values (a sorted collection) or external maintenance of a total or partial order. Additionally, options are provided to declare whether duplicate key values should be permitted, and if so, how.

The options for handling of duplicate keys provide considerable flexibility. Entire structures of nested collections can be created that subdivide the key space through progressive levels. By accepting duplicate keys, and organizing members of the same identity into subgroups, keyed collections support the abstract data type known as a bag or multiset. The index on a keyed collection includes the ability to traverse throughout the contents of any of these nested collections.

DupOption

The value of DupOption specifies how duplicate key values are to be handled within a keyed collection. This option has one of the following values:

```
id <Symbol>  DupIsError, DupRejected, KeepDuplicates, KeepCountOnly;
```

If the DupIsError value is specified, any attempt to insert a duplicate key is an error, and effects of such an attempt depend on the level of debug checking. If debug checking for this error is turned on, the collection will check whether any new key duplicates an existing one, and raise a DuplicateKey error if it does. If debug checking is turned off, there is no guarantee that any check will be made, and the effects could be unpredictable in or catastrophic or insidious ways.

With a value of DupRejected, attempts to insert a duplicate key are always detected, and the attempt rejected. (All insert messages return a boolean flag as to whether a duplicate occurred.)

With a value of KeepDuplicates, attempts to insert a duplicate key are always accepted, and all members having the same key are retained in a specially created internal collection containing just the members that all have that same key. This collection then becomes the single member internally associated with the key value.

With a value of KeepCountOnly, duplicate members are not retained. Instead, only a count of the net number of insertions and removals at that key value are retained within the collection. With this value, member values cannot be retrieved from the collection, but only a count of members for a given key value, using different messages. This value is accepted only for the Set subtype of a KeyedCollection.

GroupType

The GroupType option specifies the type of internal collection to be created for members that all have the same key. This option is valid only if DupOption is also set to a value of KeepDuplicates.

Without this option, the type of internal collection created for members with the same key depends on the specific subtype. This option allows an external program to customize the type of internal collection permitted. The argument must be a previously created subtype of Collection. Typically, the argument is a customized version of one of the standard collection types such as List, Set, or Map; the argument would be the type object returned by customizeEnd after a customizeBegin: sequence.

More information about all the ways this option can be interpreted is provided with each subtype.

SingletonGroups

If this option is true, the type of collection created for members with duplicate keys is also created even when there is only member for a particular key. This option can be convenient to assure uniformity of a keyed collection with nested structure. This option is valid only if DupOption is also set to a value of KeepDuplicates.

A bucket function accepts a single key value as its argument, and return an integer that identifies a bucket. As with a compare function, the key value is declared as an id pointer type, but may actually contain whatever kind of value is used within the collection. Following is the type of the bucket function pointer:

```
typedef int (*bucket_t)( id );
```

A new bucket function may be established any time during the lifetime of a collection. A new bucket function can ensure that keys remain suitably distributed to buckets as the total number of keys in a collection grows or shrinks.

BucketCount

A value for MaxBuckets must be given in the same message that sets a bucket function. Its value specifies the number of buckets to which a bucket function assigns its keys. The maximum value returned by a bucket function should be one less than this value.

PartiallyOrdered

A partially ordered collection supports the maintenance of individual ordering relations among its members. Each such relation is an assertion that one member should always precede another in the enumeration sequence of the collection. When an index traverses the collection, it will guarantee that all the predecessors of a member have already been visited before the member itself is reached.

If true, PartiallyOrdered specifies that messages for maintaining a partial order are enabled on the collection. An error will be raised if individual members do not have a unique identity within the innermost collection within which they are contained. This means that if duplicate keys are permitted by the collection, a collection type must also be specified for GroupType which does not itself accept duplicates. Every member in a partial order must have a unique identity so that the member itself is sufficient to identify it within the network of partial orders.

PartialOrderContext: aKeyedCollectionCollection

Ordering relations may be maintained not only between members belonging to the same collection, but between members and other collections, as long as all participants in an ordering are contained somewhere within a larger common collection. An ordering relation with a nested collection specifies that all members in the collection are successors or predecessors of the other member. Ordering relations can also be established between members of a larger, containing collection, and individual members of a nested collection.

The keyed collection specified as the value for PartialOrderContext must specify another collection in which the local collection is maintained. It is an error if the PartialOrderContext collection does not actually contain the local collection.

the number of member entries which are all associated with its key value argument. This number is zero if the key value is not contained in the collection, and will never exceed one one unless a collection accepts duplicate entries for a key.

The removeKey: message removes a member matching a key value from the collection, and returns the member just removed. It returns nil if there is no key value in the collection which matches. If more than one entry was present for the key value, it removes and returns the first member in the internal collection created for duplicate members.

```
- createIndex: aZone at: aKey;
- createIndex: aZone setMember: aMember;
```

(.. Neither of these messages is currently implemented.)

The createIndex:at: message creates a new index on the collection and also sets its current position to the first member matching the key value given as its final argument. It returns the index just created.

The createIndex:at: message creates a new index on the collection and also sets its current position to the member passed as its final argument. This message is valid only if an internal member slot was defined for the collection with the MemberSlot option.

```
- (void) getPredecessors: aMember;
- (void) getSuccessors: aMember;
- (void) getCostarts: aMember;
- (void) getCoends: aMember;
```

(.. None of these messages is currently implemented.) These messages maintain the ordering relations within a partially ordered collection.

KeyedIndex behavior

```
- (int) getCurrentLevel;
- getIndexAtLevel: (int)level;
```

An index to a keyed collection traverses all members of the collection, regardless of whether these members belong to collections of members entered under duplicate key values. Internally, however, an index keeps track of any specific subcollection it is currently processing. The getCurrentLevel message returns the level of nesting, within collections of duplicate entries for a current key value, to which the current position of an index belongs. The level returned is an integer that starts at zero for the top-level collection, and increases by one for each level of nested collection that is currently active.

While processing any nested collections, the index for a keyed collection actually maintains a separate, internal index for each level of collection that is currently active. The index at each level may be retrieved using the getIndexAtLevel: message.

```
- at: aKey;
```

The at: message repositions the index to the first member that matches the key value passed as its argument.

Name

Set – collection of members each having a defined identity

Synopsis

```
//
// Set -- collection of members each having a defined identity
//
@deftype Set <KeyedCollection, CREATABLE>CREATING
- (void) setKeyFunction: (id (*)(id))aFunction;
USING
- (id (*)(id)) getKeyFunction;

- (BOOL) add: anObject;
- (BOOL) add: anObject setIndex: anIndex;
- replace: anObject;
@end
```

Description

Set is a subtype of KeyedCollection in which the key value associated with each member is determined by the member value itself. The key value may be identical to the member itself, or may be defined as a a function of the member using a create-time option.

Inherited behavior

```
@deftype Set <KeyedCollection, CREATABLE>
```

The Set type inherits most of its interface from the KeyedCollection supertype. Set defines no create-time options beyond those already defined by KeyedCollection. If a custom compare or bucket function is specified, the member value is passed as the key value arguments of these functions. These functions determine what part of the member value is part of the key value, by determining which key values will compare equal to any member.

Set behavior

```
- (BOOL) add: anObject;
- (BOOL) add: anObject setIndex: anIndex;
- replace: anObject;
```

Name

OrderedSet – set of members in an externally assigned linear sequence

```
(..  This type is currently implemented only using the low-level option of an internal
member slot, and the messages for that option do not match the documentation in KeyedCollection.
If you need one of these objects, then either use a List or wait for some other implementation.)
```

Synopsis

```
//
// OrderedSet -- set of members in an externally assigned linear sequence
//
@deftype OrderedSet <KeyedCollection, List, CREATABLE>@end
```

Description

An OrderedSet is a totally ordered collection of members in which every member also has a distinct identity as defined by comparison against a key value.

Inherited behavior

```
@deftype OrderedSet <KeyedCollection, List, CREATABLE>
```

The sequence of members of an OrderedSet is established using the same messages that maintain member sequence in a List. An OrderedSet supports customization and access by key as defined by Set and KeyedCollection. The union of messages from all these sources defines the total interface of an OrderedSet. Members with duplicate keys, however, are not valid for an OrderedSet. Each member must have a unique position within the member sequence

---

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

Last modified: 1998-01-08

Inherited behavior

For the Map type, key values are independent of the member values with which they are associated. Map defines two additional options to document information about its key values. Map also defines its own messages to distinguish the key value from member value in any operation which involves both.

```
@deftype Map <KeyedCollection, CREATABLE>
```

Create-time options

```
CREATING
- (void) setKeyType: aDataType;
- (void) setKeySlot: (int)byteOffset;
USING
-         getKeyType;
- (int) getKeySlot;
```

(..None of these options is currently implemented.)

Map behavior

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

- at: aKey replace: anObject;
```

at:insert: inserts an entry into a Map containing the key and member values given as its arguments. It returns true if the key was not previously contained in the collection. If the key was already present, the action of the message depends on the value in effect for the DuplicatesOption. With default options, the attempt to insert a duplicate key is simply rejected and false is returned.

at:insert:setIndex: is similar to at:insert:, except that the index passed as the final argument is repositioned to the member just inserted. If further operations are required on the entry in a map that was just inserted, an index provides faster repeated access to this entry than a message involving further key comparison, such as at:, would require. (.. This message is not currently implemented.)

at:replace replaces an existing member value associated with a key value by a new value given as its final argument. The message returns the member value which was formerly associated with the key value. The message returns nil if more than one member is present with the same key.

```
- createIndex: aZone setKey: aKey;
```

Name

String – character string object

(.. This type will be more fully documented when it is generalized beyond its current status as nothing more than an object wrapper for a null-terminated character array.)

Synopsis

```
//
// String -- character string object
//
@deftype String <Create, Drop, Copy, CREATABLE>CREATING
+ create: aZone setC: (const char *)cstring;
SETTING
- (void) setC: (const char *)cstring;
USING
- (const char *)getC;
- (void) appendC: (const char *)cstring;

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

Description

The String object type packages a null-terminated, C-format character string into an object. All memory allocation needed to hold the string value is handled by the object. This type currently defines only the most rudimentary operations for initializing and appending C-format character strings. These are sufficient for its current limited roles in places that need a uniformity between character strings and other kinds of allocated objects.

Inherited behavior

```
@deftype String <Create, Drop, Copy, CREATABLE>
...
```

Name

OutputStream – stream of output bytes

(.. This type will be more fully documented when it is generalized beyond its current status as a placeholder used only to direct debuggging output.)

Synopsis

```
//
// OutputStream -- stream of output bytes
//
@deftype OutputStream <Create, Drop, CREATABLE>CREATING
+ create: aZone setFileStream: (FILE *)fileStream;
- setFileStream: (FILE *)fileStream;
USING
- (FILE *) getFileStream;
- (void) appendC: (const char *)cstring;
@end
```

Description

The OutputStream type currently supports only the writing of a character string to a file. It is a placeholder for more general stream types. A stream is a collection that supports only sequential addition of members (an output stream) or sequential removal of members (an input stream).

Inherited behavior

```
@deftype OutputStream <Create, Drop, CREATABLE>
...
```

Create-time options

```
CREATING
- setFileStream: (FILE *)fileStream;
USING
- (FILE *) getFileStream;
```

Collections Library: Subclassing Reference

Until the collections library has been fully implemented, subclassing conventions from collections implementation classes are still in flux. In general, these classes will be among the most complex uses of multiple classes selected to implement an independent object type. (See library definition conventions (../defobj/library.html) for a summary of the distinction between types and classes.) New methods are being developed to simplify subclassing from such implementations. In the meantime, if you need to use to a collection within the implementation of your own class, just put an instance variable in your class and put the collection in that, and pass through the messages of the collection you want to have available on your class to this variable. In many if not most cases, this is better design anyway, because you control all use of the underlying structure.

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

Last modified: 1998-01-08

Collections Library: Implementation Notes

(This section of documentation is not yet available.)

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

Last modified: 1998-01-08

(activity/design.html)

- Implementation Notes
(activity/impl.html)

---

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

Last modified: 1998-01-08

Activity Library: Usage Guide

(.. This is an overview of activity library function. Code examples needed by a full usage guide will follow later.)

Role of the activity library in Swarm

The activity library provides a foundation for dynamic, object-oriented simulation in Swarm. Swarm assumes that a user defines an object-oriented representation for the structure of a world to be simulated. Once such a representation is established, activity library components are responsible for generating all state changes and flow of information within it. These state changes must be carefully controlled to ensure that they occur only at appropriate times and places in the model. The activity library provides mechanisms to establish this control.

Once the simulation of a dynamic model begins, everything that happens to the model occurs as a direct result of messages sent to world objects by activity library components. These components are also represented by objects, since this is the most effective means for their representation as well, but their purpose is not to represent the current state of the simulated world, but rather to generate changes in world objects by sending messages in a valid order. The activity library has two basic categories of components: those that represent messages to be sent, including constraints on permissible order for sending them, and those that execute the message sends these representations specify, making sure they conform with all constraints.

All changes to a model occur as a result of messages sent to objects within it. These messages invoke compiled methods of receiving objects, which may change local state or send messages to other objects to propagate effects as widely as needed. Only top-level function calls or message sends that initiate model processing are the concern of the activity library. Because the activity library finally interacts with the world model just by invoking its defined methods, the model can prepackage as much behavior as it can in the form of fast compiled methods. The activity library just triggers the prepackaged behavior at proper times under its own explicit, dynamically interpreted representation.

As conditions shift during execution of a model, the model can also examine and alter the representation of its future behavior contained in the activity structures that drive it. Any changes to this future behavior, however, still occur only as a result of some currently executing action initiated by the activity library.

Activity library components

The activity library defines one set of basic object types to provide a very rich representation of the kinds of message sending patterns it could generate. Its other major category of object

Action plan components

The first set of activity library components represent messages to be sent to objects in a model, together with constraints in the order in which they may be sent. All these components are defined as subtypes of the one generic type called ActionPlan.

The two basic kinds of action plans are a simple group of actions to be performed in some order, called an ActionGroup, and a series of actions to be performed at discrete points in time, called a Schedule. The basic element of an action plan is a simple object called an Action. An action defines a particular message to be sent to an object or objects.

In its representation of action plans, Swarm relies heavily on the dynamic message sending capabilities of the Objective C language. The support of Objective C for dynamic message sends is absolutely crucial to Swarm's implementation of generic activity structures. Objective C defines a special kind of data value called a selector, which identifies a message according to its name. One of these selector values is stored in each action of an action plan. During execution of the plan, Objective C machinery performs the actual message send using its selector.

Action plans are free-standing objects that may be created directly by a user program, using a variety of selectable create-time options. Once created, action plans may also refer to each other, by containing special kinds of actions to start another action plan or to perform all the actions within it.

Individual actions can be created only as completely controlled components of some action plan. They are created not by a standard create message, but by special messages (each containing the phrase createAction in its name) sent to an action plan that creates the action as part of the action plan. If the same action is needed in more than one plan, it has to be created in each plan where it is needed. If the entire action plan is dropped, all its actions are also automatically dropped. So the only action plan components which need to be directly managed by an application are the action plans themselves.

Action plans are relatively straightforward components, because they are entirely passive. The only actions they ever contain are those which an application has created in them. These actions stay in them indefinitely unless a special option is requested to clean them out as each one is executed. Because these plans are passive, read-only components to all execution machinery, if the same pattern of actions needs to be triggered at multiple points in a model, it's perfectly valid to create one copy of the actions in an action plan, and then refer to the plan anywhere the actions may be needed.

Even though action plans are passive, containing only what has been placed in them, there is no requirement that their contents remain fixed. Both of the action groups and schedules are implemented directly as collections of their actions. New actions may be added at any time, and existing ones may be dropped or moved from position to another. The contents of action plans may be as dynamic as they need to be to represent the future actions needed in a model. None of these shifting contents has any effect on the model until actually processed during model execution.

processing the other action plan completes. If one action plan starts another, the new action plan is started as an autonomous activity controlled only by a higher-level, containing activity.

A Swarm is an activity that exists only to control and coordinate other started subactivities. Unless the subactivities have some special form of explicit synchronization, none of their internal actions has any required ordering relative to those of other plans except as explicitly established during activity execution. The swarm can serve as a simple container of started subactivities which only occasionally synchronize for messages they send to each other. The swarm can be used to hold collections of objects as well as its subactivities as needed to help them coordinate with one another.

One special form of synchronization within a swarm is built into the virtual processing machinery. This synchronization interleaves the actions that occur for every successive time value during processing of timeline subactivities. This merging of actions is often relied on to interleave display and analysis processing with the scheduled actions of a base model. Since no other mechanisms for subactivity coordination are implemented in the current version of Swarm, synchronizing subschedule activities is the major current role of the Swarm activity type. In later versions of Swarm, a swarm will also serve as an important means for organizing a large models into clusters of more densely interacting components, and will also provide a basis of decomposition for parallel execution.

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

Last modified: 1998-01-08

Activity Library: Interface Reference

Dependencies

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

```
#import <collections.h>
```

The activity library follows the Swarm library interface conventions
(../defobj/library.html)
of the defobj library
(../defobj.html)
. The activity library relies heavily on the basic collection types defined in the collections library
(../collections.html)
, and the collection interfaces are an integral part of the interfaces defined within this library. Initialization of the activity library is done automatically by the containing Swarm libraries, and automatically initializes the collections and defobj libraries as well.

List of Reference Pages

- ProcessType
(ProcessType.html)
- ActionType
(ActionType.html)
- CompoundAction
(CompoundAction.html)
- ActionGroup
(ActionGroup.html)
- Schedule
(Schedule.html)
- SwarmProcess
(SwarmProcess.html)

- Action Types
(Action.html)
- Activity Types
(Activity.html)

Name

ProcessType – specification of a process

Synopsis

```
//
// ProcessType -- specification of a process
//
@deftype ProcessType
@end
```

Description

(.. For now, ProcessType is just a placeholder for future capability. ..)

ProcessType will eventually define support for parameterization of all processes. A process is a uniquely identified course of events that conforms to a specification of external and internal behavior.

Processes include both objects and executable actions. Objects change their state and behavior based on a sequence of externally initiated actions. An action, in contrast, typically has no external behavior other than to be executed in its entirety as a unit (yielding either an observable result or some change of state in a surrounding environment).

Parameterization will be based a uniform framework that defines all input parameters, internal state variables, and/or final results, and that provides for binding of these data in a context of execution. This parameterization framework is still being established, but ProcessType objects will eventually record the specification of any process, to the extent that this specification is defined.

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

Last modified: 1998-01-08

ActionType behavior

```
- activateIn: swarmContext;
```

The activateIn: message is used to initialize a process for executing the actions of an ActionType. This process is controlled by an object called an Activity. The activateIn message initializes an activity to run under the execution context passed as the swarmContext argument, and return the activity object just created. If the execution context is nil, an activity is returned that allows complete execution control by the caller. Otherwise, the execution context must be either an instance of SwarmProcess or SwarmActivity. (These objects are always maintained in one-to-one association with each other, either one of the pair is equivalent to the other as a swarmContext argument.)

If a top-level activity is created (swarmContext is nil), the created activity may be processed using activity processing commands such as run, step, etc. If an activity is created to run under a swarm context, the swarm itself has responsibility for advancing the subactivity according to its requirements for synchronization and control among all its activities. Activating a plan for execution under a swarm turns over control to the swarm to execute the subactivity as a more-or-less autonomous activity.

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

Last modified: 1998-01-08

Create-time options

```
CREATING
- (void) setDefaultOrder: aSymbol;
- (void) setAutoDrop: (BOOL)autoDrop;
USING
- getDefaultOrder;
- (BOOL) getAutoDrop;
```

DefaultOrder

The DefaultOrder option indicates the ordering to be assumed among actions of the plan when no other explicit ordering has been assigned. Beyond this initial ordering, additional ordering constraints can be added selectively using partial order specifications on individual actions. (.. Partial order order constraints are not yet implemented.)

The value for DefaultOrder is a symbol that may have one of the following values:

```
extern id <Symbol>  Concurrent, Sequential, Randomized;
```

The Concurrent value of the DefaultOrder option indicates that can actions be run in any order (including at the same time, if hardware and software to do this is available) without no impact on the net outcome of the actions. The claim that action results are independent of their execution order gives the processing machinery explicit leeway to execute them in any order it chooses. In the current implementation on a single, serial processor, actions are always processed sequentially even if marked concurrent, because that is the only way they can be. In future versions, however, special runtime processing modes may be defined even for a serial processor, which would mix up execution order just to confirm the independence of model results.

The Sequential value for the DefaultOrder option is the default. It specifies that the actions must always be executed in the same order as they occur in the plan. This order is ordinarily the same order in which actions are first created in the plan, unless actions are explicitly added elsewhere the collection that underlies a plan. This option is always the safest to assure predictability of results, but it excludes the ability to run the actions in parallel. To better understand and document a model design, it is worth annotating action plans with an explicit indication as to whether they do or do not depend on a Sequential order.

The Randomized value for the DefaultOrder option specifies that the model results do depend on execution order, but that the order in which the actions were created or added has no special significance. Instead, the method of dealing with order dependence is to generate a random order each time a collection of same-time actions is processed. The random order will be generated from an random number generator internal to the processing machinery.

AutoDrop

The AutoDrop option specifies that as soon as any action been processed by a running activity, the action is removed from the plan and dropped so that it will never appear again. This option is useful for plans that are created for a one-time use, never to be used again. This option

Name

ActionGroup – collection of actions under partial order constraints

Synopsis

```
//
// ActionCreating -- protocol shared by ActionGroup and Schedule
//
@deftype ActionCreating
- createAction: anActionType;

- createActionCall: (func_t)fptr;
- createActionCall: (func_t)fptr : arg1;
- createActionCall: (func_t)fptr : arg1 : arg2;
- createActionCall: (func_t)fptr : arg1 : arg2 : arg3;

- createActionTo: target message: (SEL)aSel;
- createActionTo: target message: (SEL)aSel : arg1;
- createActionTo: target message: (SEL)aSel : arg1 : arg2;
- createActionTo: target message: (SEL)aSel : arg1 : arg2 : arg3;

- createActionForEach: target message: (SEL)aSel;
- createActionForEach: target message: (SEL)aSel : arg1;
- createActionForEach: target message: (SEL)aSel : arg1 : arg2;
- createActionForEach: target message: (SEL)aSel : arg1 : arg2 : arg3;
@end

//
// ActionGroup --
//   a collection of actions under total or partial order constraints
//
@deftype ActionGroup <CompoundAction, ActionCreating, OrderedSet, CREAT-
ABLE>@end
```

Description

An action group is an action plan whose basic representation is a sequence of actions that have been created within it. Even though a sequence is used to represent the actions of the collection, this does not necessarily mean that the execution machinery will process them in this order, since the permissible order of execution is still controlled by the DefaultOrder must still be declared to receive the type of argument you actually pass, before you cast it to the id type.

(.. Alternative approaches to argument typing are currently in development, but these will supplement rather than replace the current forms of createAction messages.)

The createAction: message specifoes that processing of another action type is to be performed by the action. The referenced action type is performed in its entirety, from start to finish, as the effect of the single created action.

The createActionCall: messages are similar to the createActionTo messages, except they specify the action to be performed as a binding of a C function to a list of argument values. The correct number of arguments for the function pointer passed as the initial argument must be supplied.

The createActionForEach: messages define a message to be sent in the same way as the createActionTo messages, but they assume that the object passed as the target argument is a collection object. They specify that each object available from that collection, using the standard messages of the Collection
(../collections/Collection.html)
type in the collections library, is to receive the specified message.

The createAction: messages specify that processing of another action plan is to be performed by the action. The referenced action plan is performed in its entirety, from start to finish, as the effect of the single created action. If parameters are defined on the action plan, arguments for the action can also be given. (.. Currently this is not supported, so only the single argument form of createAction: should be used.)

Each of the createAction messages returns the action object which it creates. Each different kind of createAction message returns a different type of Action object with a matching name. These Action objects provide access to all the information with which the Action was initialized. The complete set of Action object types is defined below together with the messages that may be used to access their contents. (.. The implementation of the Action objects is currently undergoing change as the responsibility for parameter and return value typing gets taken over by ActionType in defobj.)

The action objects are an integral, controlled component of the action plan in which they are created. If they are removed from the action plan collection using a remove message, the only collection in which they may be reinserted is the same collection from which they came. It is permissible, however, to modify the base representation sequence by removing from one position and reinserting at another.

The contents of the action objects would ordinarily be referenced only by general-purpose tools to display or analyze an action plan structure. The action object id, however, can also be used to establish additional ordering constraints on any of the actions in a plan. These constraints supplement any initial order constraints established by the DefaultOrder option. Constraints may be established not only within the same plan, but between actions in different plans. (.. Partial order constraints are not yet implemented.)

Name

Schedule – collection of actions ordered by time values

Synopsis

```
//
// Schedule -- collection of actions ordered by time values
//
@deftype Schedule <CompoundAction, ActionCreating, Map, CREATABLE>CREATING
- create: aZone setRelativeTime: (BOOL)relativeTime;
- create: aZone setRepeatInterval: (timeval_t)repeatInterval;
SETTING
- (void) setConcurrentGroupType: groupType;
- (void) setSingletonGroups: (BOOL)singletonGroups;

- (void) setRelativeTime: (BOOL)relativeTime;
- (void) setRepeatInterval: (timeval_t)repeatInterval;
USING
- getConcurrentGroupType;
- (BOOL) getSingletonGroups;

- (BOOL) getRelativeTime;
- (timeval_t) getRepeatInterval;

- at: (timeval_t)tVal createAction: anActionType;

- at: (timeval_t)tVal createActionCall: (func_t)fptr;
- at: (timeval_t)tVal createActionCall: (func_t)fptr:arg1;
- at: (timeval_t)tVal createActionCall: (func_t)fptr:arg1:arg2;
- at: (timeval_t)tVal createActionCall: (func_t)fptr:arg1:arg2:arg3;

- at: (timeval_t)tVal createActionTo: target message: (SEL)aSel;
- at: (timeval_t)tVal createActionTo: target message: (SEL)aSel:arg1;
- at: (timeval_t)tVal createActionTo: target message: (SEL)aSel:arg1:arg2;
-
at: (timeval_t)tVal createActionTo: target message: (SEL)aSel:arg1:arg2:arg3;

- at: (timeval_t)tVal createActionForEach: target message: (SEL)aSel;
- at: (timeval_t)tVal createActionForEach: target message: (SEL)aSel:arg1;
-
at: (timeval_t)tVal createActionForEach: target message: (SEL)aSel:arg1:arg2;
- at: (timeval_t)tVal createActionForEach: target message: (SEL)aSel:
                                                                arg1:arg2:arg3;
```

default that times are considered absolute, as if the RelativeTime option had also been specified at the same time. All scheduled times must be less than the specified repeat interval, or an error will be raised. The RepeatInterval option can continue to be reassigned to different values after a schedule has been created, but the interval value must always be greater than the scheduled times of any actions which it contains.

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

ConcurrentGroupType

The ConcurrentGroupType option is used to control handling of multiple actions which end up being scheduled at the same time value. As far the schedule representation is concerned, these actions are assumed by default to be concurrently executable, and the processing machinery is free to process them as such if no ConcurrentGroupType option is specified.

If a different interpretation of actions at the same time step is needed, the ConcurrentGroupType option may be specified. The argument of this option must be an object that when given a standard create: message will return an object having all the structure of a standard ActionGroup object. For example, if you want to specify that actions at the same time step are to be processed in random order, you could use the following procedure:

```
concurrentGroupType = [ActionGroup customizeBegin: aZone];
  [concurrentGroupType setDefaultOrder: Random];
  concurrentGroupType = [concurrentGroupType customizeEnd];

  aSchedule = [Schedule createBegin: aZone];
  [aSchedule setConcurrentGroupType: concurrentGroupType];
  aSchedule = [aSchedule createEnd];
```

Note that instead of using the normal createBegin/End sequence to create an instance, the concurrentGroupType is initialized using the customizeBegin/End sequence to customize a type without creating an instance. For more information, see the defobj library documentation.

In addition to overriding the standard ActionGroup type, the concurrent group type may be implemented by a custom subclass of the ActionGroup implementation which you supply yourself. A custom subclass is free to implement custom rules for how to combine all the actions which happen to arrive at the same time value. For example, it could decide that some actions are not to be executed at all, or it could create entirely new actions to be executed instead of or in addition to those which were originally scheduled. (.. Specific rules for writing a custom ActionGroup subclass will be documented at a future time, but all the apparatus to do so is present today. A concurrent action group is currently used to maintain the proper order of execution among subswarms of an owner swarm.)

SingletonGroups

The SingletonGroups option indicates that an action group should be created for every time value which is present, even when only a single action is present at the time value.

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

Last modified: 1998-01-08

If subactivities running within a swarm are indexed by time (running either schedules or other swarms), the default coordination performed by the swarm is to merge all the different scheduled actions so that the actions of one time value are all performed before the actions of any later time value. If the actions are not indexed by time, or if an subactivity has been released from strict time synchronization with other subactivities in the swarm, the default coordination is limited to partial order constraints among the action plans being run. (.. Currently only the merging of schedules and subswarms by time values is supported.)

A public superclass called Swarm is supplied in the objectbase library
(../objectbase.html)
for use in implementing custom swarm classes. Such a class can override the method for the activateIn: message of ActionType
(ActionType.html)
to automatically start subactivities to run within a swarm when the swarm is first activated.

Inherited behavior

```
@deftype SwarmProcess <ActionType, Zone, CREATABLE>
```

SwarmProcess inherits the messages of both ActionType
(ActionType.html)
and Zone
(../defobj/Zone.html)
. Inheritance of zone behavior means that a swarm can be used as the argument of a create:
or createBegin: message, for creation of an object within the internal zone of a swarm.

The default value for the SynchronizationType option is not a generic action group, but a special predefined subtype of ActionGroup called ActivationOrder. This concurrent group type guarantees that actions scheduled to occur at the same time from different action plans running in the same swarm are executed in the same order in which the action plans were first activated.

Create-time options

```
CREATING
- setInternalZoneType: aZoneType;
- (void) setSynchronizationType: aScheduleType;
- (void) setInternalTimeMultiplier: (timeval_t)multiplier;
USING
- getSynchronizationType;
- (timeval_t) getInternalTimeMultiplier;
```

Name

Action – element of a compound action

Synopsis

```
//
// Action -- messages shared by ActionCall, ActionTo, and ActionForEach
//
@deftype Action <GetOwner>- (int) getNArgs;
- (void) setArg1: arg1;
- getArg1;
- (void) setArg2: arg2;
- getArg2;
- (void) setArg3: arg3;
- getArg3;
@end

//
// ActionCall -- an action defined by calling a C function
//
@deftype ActionCall <Action>- (void) setFunctionPointer: (func_t)fptr;
- (func_t) getFunctionPointer;
@end

//
// ActionTo -- an action defined by sending an Objective C message
//
@deftype ActionTo <Action>- (void) setTarget: target;
- getTarget;
- (void) setMessageSelector: (SEL)aSel;
- (SEL) getMessageSelector;
@end

//
// ActionForEach --
an action defined by sending to every member of a collection
//
@deftype ActionForEach <ActionTo>@end
```

Activity Types

Contents

The following object types are documented below:

- Activity
- GroupActivity
- ForEachActivity
- ScheduleActivity
- SwarmActivity

Name

Activity – state of processing within an action plan

Synopsis

```
//
// Activity -- state of processing within an action plan
//
@deftype Activity <DefinedObject, Drop>- run;
-               stop;
- (void) terminate;

- next;
- step;
- stepEntry;
- stepExit;

- getStatus;
- getHoldType;

- getAction;
- getActionType;
```

state of processing may be represented by a single stack of current activities that traces a path to a single leaf of the runnable activity tree. When running in serial mode, messages are available to obtain the currently running leaf activity or useful context information such as the current time available from it.

Activity behavior

(.. Specific messages for activity objects are still subject to change as some of their underlying support migrates to the defobj

(../defobj.html)
library. For now, this section will document just the basic messages to start and control processing within an activity.)

```
- run;
-               stop;
- (void) terminate;
```

The run message continue processing of an activity from any state in which its execution has been suspended. An error is raised if the activity is already running. run returns either a Stopped or Completed status that the activity has when it is no longer eligible to be run further.

The stop message causes the a currently running tree of activities to stop further processing and return a Stopped status. terminate also stops a running tree of activities, but sets all activities within the tree to a status of Completed whenever the tree is next run. terminate may be used on either a running or stopped tree of activities. It is the standard way to terminate schedule that is endlessly repeating under the RepeatInterval option.

```
- next;
- step;
- stepEntry;
- stepExit;
```

The next executes actions within a single compound action while skipping over all processing of any complete action plans executed by those actions. The step message executes a single action within a tree of activities. The stepEntry message executes multiple actions within a single currently runnable activity, but stops the activity when a new level of action plan has just been started within the tree of activities. stepExit executes multiple actions in the same action plan until the activity for that action plan has completed, or cannot otherwise be run further. (.. Currently, stepEntry and stepExit are unimplemented.)

```
- getStatus;
- getHoldType;
```

The getStatus message returns one of the following codes for the current run status of a particular activity:

```
extern id <Symbol>  Initialized, Running, Stopped, Holding,
```

Synopsis

```
//
// GroupActivity -- state of processing within an ActionGroup
//
@deftype GroupActivity <Activity>@end
```

Name

ForEachActivity – state of processing within an ForEach action

Synopsis

```
//
// ForEachActivity -- state of execution within a ForEach action
//
@deftype ForEachActivity <Activity>
- getCurrentMember;
@end
```

Name

ScheduleActivity – state of execution within a Schedule

Synopsis

```
//
// ScheduleActivity -- state of execution within a Schedule
//
@deftype ScheduleActivity <Activity>
- (timeval_t)   getCurrentTime;

- stepUntil: (timeval_t)tVal;  // unimplemented
@end
```

Activity Library: Subclassing Reference

Subclassing is supported by the activity library as an integral technique for extending the framework it implements. There are two specific places where subclassing is expected as the normal technique of extension: definition of concurrent action groups, and definition of customized swarm objects. Concurrent action groups are not yet fully documented because their full interface is still being finalized. Swarm subclasses should inherit from the Swarm superclass defined in the objectbase (../objectbase.html)
library; the activity library provides the underlying support packaged by this superclass.

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

Last modified: 1998-01-08

Activity Library: Implementation Notes

(This section of documentation is not yet available.)

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

Last modified: 1998-01-08 (mgd)

Overview

The *objectbase* library encapsulates various fundamental aspects of the Swarm object model and defines the probing machinery used to take data from Swarm objects. Most of the underlying functionality of the classes defined here is contained in the *defobj*

(defobj.html)
and *activity*
(activity.html)
libraries.

The main classes defined here that are intended to be subclassed by users are Swarm and SwarmObject. The probing functionality provided here is mainly for use within the Swarm kernel. However, this probe machinery should be used when designing any interface between Swarm and any other agent or device.

SwarmObject

SwarmObject is the base class for all interactive agents in Swarm. It defines the standard behavior for Swarmstyle agents, which includes hooks for creation, probing, zoned memory allocation, and destruction.

Swarm

The *Swarm* class encapsulates all of the abstract execution machinery in the *activity* library, as well as the notion of a group of related objects. Metaphorically, a "Swarm" is a combination of a collection of objects and a schedule of activity over those objects. Hence, the *Swarm* class provides the behavior necessary for creating such an object and starting up the schedule.

Probes

Probes are idealized entities that are intended to allow the user to monitor and modify the innards of objects without explicitly providing the functionality to do so at compile time. Hence, they allow dynamic interaction with an objects instance variables and methods. Most of the functionality of probes is implemented here; but, at present, they are intimately linked to the *simtools*

(simtools.html)
library, which contains all the widgetry needed to use probes from a GUI. Despite this intimacy, probes are intended to be a general purpose mechanism for any agent or device to interact dynamically with Swarm objects. This Introduction to Probes

Objectbase Library: Documentation and Implementation Status

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

Last modified: 1998-01-08

```
ProbeMap * myProbeMap;
```

can be changed to

```
id <ProbeMap> myProbeMap;
```

This both allows the freedom from the static typing and adds to the readibility of the code. However, this still results in compile time checking for messages sent to that object with the protocol designated. So, any message you intend to send to that object must be defined or inherited by the given protocol. So, in some cases, one may want to go with the ultimate freedom in

```
id myProbeMap;
```

which, obviously, means the compiler will not second guess the programmer in what messages can be sent to that object. And if your variables are named appropriately, this doesn't really detract from the readibility of the code.

2. When subclassing from a class defined in the Swarm Kernel, it is necessary to import the `@interface` of that class. Before, when `objectbase.h` included all those interfaces, it was the only file needed if you wanted to subclass from a class in objectbase. Now, however, simply importing `objectbase.h` doesn't help at all. So, in order to subclass from a class that's a member of objectbase, you need to import the specific interface file to that class.
   The solution: If you had an app that worked before when you subclassed from something (say, Swarm) in objectbase but doesn't work now, simply replace the

```
#import <objectbase.h>
```

in the header file for the subclass with

```
#import <objectbase/Swarm.h>
```

Note: In order to reduce the number of broken apps due to this, we've explicitly imported `objectbase/SwarmObject.h` and `objectbase/Swarm.h`at the end of the `objectbase.h` file. This allows the subclassing of SwarmObject or Swarm by an import of `objectbase.h` without the explicit import of the interface files. But, this was done for backward compatibility and may go away in the future. So please subclass by explicit importation of the interface file of the class.

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

Last modified: 1998-01-08

```
(CustomProbeMap.html)
   \      Complete
(CompleteVarMap.html)
                         Probe
(MessageProbe.html)
   Probe
(CustomProbeMap.html)
    \       Var
(CompleteVarMap.html)
                                    Map
(CustomProbeMap.html)
      \       Map
(CompleteVarMap.html)
                                            Complete
(CompleteProbeMap.html)
                                             Probe
(CompleteProbeMap.html)
                                              Map
(CompleteProbeMap.html)
```
(don't want a graphic with image map to avoid excessive download times!)

Contents:

SwarmObject Class

```
//
// SwarmObject --
//   Superclass of most objects in a Swarm simulation.
//   Provides support for probing.
//
@protocol SwarmObject <Create, Drop>
USING
- (const char*)getInstanceName;
- getProbeMap;
- getCompleteProbeMap;
- getProbeForVariable: (const char *)aVariable;
- getProbeForMessage: (const char *)aMessage;
@end
```

A SwarmObject is an object that is intended to be a member of a Swarm. That is, its behavior will be perpetuated by messages sent from the schedule of events defined in the context of a Swarm object.

The SwarmObject is where the models of all the agents of a simulation will reside. Hence, most of the burden on defining the messages that can be sent to any agent lies with the user. SwarmObject inherits its basic functionality from the Create and Drop object types defined in the defobj library. It is useful to mention these messages here to show the complete span of messages that can dictate the behavior of an instance of a SwarmObject; but it is not useful to explain their function in detail. The messages SwarmObject inherits from defobj are:

- DefinedObject
  - - getType;
  - - getClass;
  - - (BOOL)respondsTo: (SEL)aSel;
  - - describe: anOutputStream;
  - - (BOOL)setDisplayNameTo: (const char *)aName;
  - - (const char *)getDisplayName;
  - - createIDString: aZone;
- Create
  - + create: aZone;

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

Last modified: 1998-01-10

Gets the current display precision set in the ProbeLibrary instance. Mostly used internally.

- setSavedPrecision: (int)nSigSaved
  Sets the number of significant digits saved for floating-point and double floating-point numbers through ObjectSaver. This function sets the global default precision for all floating (and double floating) point numbers. This floating point precision affects all numbers saved via the ObjectSaver class. There is currently no way to override this global default for an individual probe.

-(int) getSavedPrecision
  Gets the current saved precision set in the ProbeLibrary instance. Mostly used internally.

Note: if neither of these methods are used to set the defaults, the precision in both cases defaults to six (6) significant figures, which should simulate the default precision before the introduction on this feature *on most platforms*. Previously, the sprintf "%g" default format was used to print out floats and doubles, which is not considered to have entirely portable behavoir.

- (BOOL) isProbeMapDefinedFor: (Class) aClass returns True
  If there is a non-nil value in the *ProbeLibrary* for that class and False otherwise.

- getProbeMapFor: (Class)aClass
  returns a *ProbeMap* for the *aClass* class. If a specific *ProbeMap* has been designed and installed in the *ProbeLibrary* for that class, then that *ProbeMap* is returned. If not, then a *CompleteProbeMap* is created and returned.

- getCompleteProbeMapFor: (Class)aClass
  This method returns a *ProbeMap* containing *Probe*s for all the instance variables and messages of the given Class (including inherited variables and messages). Since we are assuming that normal, single-level *ProbeMap*s will be more widely used, the current implementation of *ProbeLibrary* does not cache *CompleteProbeMap*s. However, if we find that there is a demand for *CompleteProbeMap*s, we will cache them as well.

- getCompleteVarMapFor: (Class)aClass
  This method returns a *ProbeMap* containing Probes for all the instance variables of the given Class (including inherited variables) but does not include any *MessageProbe*s.

- getProbeForVariable: (const char *)aVariable inClass: (Class)aClass

- getProbeForMessage: (const char *)aMessage inClass: (Class)aClass
  These methods are used to "check out" the appropriate Probes from the probe library. Note that the returned probes will be cached, so you must make sure to clone them if you want to make modifications to the probes without affecting the result of future requests for those same probes.

- setProbeMap: aMap For: (Class)aClass

ActivityControl Class

```
//
// ActivityControl -- This class provides an object interface to
//                    an activity.  It contains a subset of the messages
//                    that any activity responds to and provides explicit
//                    control over that activity.
//
@protocol ActivityControl <SwarmObject>
CREATING
- attachToActivity: anActivity;
USING
- run;
- stop;
- next;
- step;
- stepUntil: (timeval_t)stopTime;
- (void)terminate;
- updateStateVar;
- getStatus;
- (const char *)getInstanceName;
@end
```

The ActivityControl class specifies an object that can be attached to an activity (regardless of how or where that activity is created) for the purpose of explicitly controlling the execution of actions on that activity's action plan. There is nothing that available through this class that is not already available through the variables or messages available via the activity itself. However, it packages up the main control messages and makes them available to other objects that may need control over an activity, thereby shielding the activity from directly receiving messages from outside objects and saving the user from having to parse the more complex interface to the activity.

Create Phase Protocols

attachToActivity: anActivity sets an instance variable inside the ActivityControl object that points to the Activity to be controlled. It then creates a Schedule upon which it places a message to itself to update its own variables.

Usage Phase Protocols

(Interface.html)

(UsageGuide.html)

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

Last modified: 1998-01-10

(const char *)getProbedType returns the typing of the probed variable/message. The typing is represented using the string-format provided by the Objective-C runtime system ('c' for char, '@' for id, and so on). In general the programmer will know the typing of the variable so this method will probably only be used internally by Swarm kernel code.

clone: aZone returns a clone of the probe. It is useful in case you got the probe by Library Generation or by the default version of Object Generation. In both cases you will have been given a unique shared instance of the probe - so if you wanted to make changes to the safety status of the probe etc. without affecting the other potential users of the probe, you would first clone it and then make the required changes.

(ProbeLibrary.html)

(SwarmObject.html)

(VarProbe.html)

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

Last modified: 1998-01-10

of the variable referent. For example, if we were probing a variable called *unhappiness*, then the string would be "unhappiness". This method must be called during the create phase.

Set Phase Protocol

- setStringReturnType: returnType sets the format that will be used to print the variable.

Usage Protocol

- setFloatFormat: (const char *)
  Given an sprintf style formatting string (eg. "%.3f") to a floating-point (or double-floating-point) probe will use this formatting *only* when probing on a GUI display widget. See section on probing precision (../probing.html#precision) when probing floating point number in the documentation on probes.
- setSafety
- unsetSafety
  In order to ensure efficiency, the default behaviour of the *VarProbe* is to "trust" that the object it is being presented with is of an appropriate class (the one specified in *setProbedClass*, or a subclass of it). However, by calling *setSafety*, we force the probe to inspect the class of the target object before probing it.
- setNonInteractive
- (int)isInteractive
  By setting a *VarProbe* to be non-interactive we ensure that the user will not be able to change the value of the probe, but only observe it. The default is for variables of type (char *), char, unsigned char, int, unsigned int, floatand double to be interactive and for all the others to be non-interactive (with the exception of the id type this follows because there is no simple way to set such things as structs or arrays by typing values into an entry field). For the time being, setting the VarProbe to be non-interactive will not interfere with the drag&drop capability of objects into the *VarProbe* entry-field... This is considered an omission and will be corrected by the next release. Also, for the time being there is no way to disable a *VarProbe* once it has been displayed (i.e. after a *ProbeDisplay* containing the *VarProbe* has been created). The ability to do this will also be included in the next release.
- (const char *)getProbedVariable
  returns a string matching the identifier of the referent.
- (int)getDataOffset
  This is another example of a method used(?) exclusively by the Swarm kernel, it returns the offset used to extract the variable from the target object.

a string or a pointer. However, if any user should require more types (such as the choice provided in the *probeAs* series), please email (mailto:swarm@santafe.edu) us and I will include them in the following release!

(Probe.html)

(SwarmObject.html)

(MessageProbe.html)

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

Last modified: 1998-01-10

Note that due to a bug in gcc-2.7.2 it is unadvisable (though we have not disallowed it) to make a message probe using setProbedMessage. The next release of gcc will contain a patch for this bug. If you absolutely require the ability to create messageprobes using setProbedMessage, please email (mailto: swarm@santafe.edu) us.

Usage Protocol

(const char *)getProbedMessage
: returns the string matching the identifier of the message being probed.

(int)getHideResult
: setHideResult: (int)val By setting hideResult to 1, the user is indicating that the result field, in a graphical representation of the message probe, should not be shown. In other words, if a probe display attempts to represent the message probe, it should hide the result window. This is useful, particularly when the return value is defined to be self.

(const char *)getArg: (int)which
: returns a string representation of the n'th argument.

(int)getArgNum
: returns the number of arguments the message takes.

(const char *)getArgName: (int)which
: returns a string representation of the argument with the given name.

setArg: (int)which To: (const char *)what
: sets the n'th argument of the message - the argument must be provided in string form.

(int)isResultId
: This method is used by the Swarm internals to find out whether the return value of the message is of type object.

(int)isArgumentId: (int)which
: This method is used by the Swarm internals to find out whether a given argument of the message is of type object.

updateMethodCache: anObject
: The message probe will always attempt to optimise the way in which it dispatches dynamic calls to the probed objects. In practice, this means that when the probed message takes no arguments, then the message probe will cache the actual implementation of the method and generate direct C function calls (as opposed to calling perform, for example) if the user promises that only one class of objects is being probed. The user does this by calling the updateMethodCache: with an example object of the appropriate target class. This method can be called as often as necessary throughout the lifetime of the message probe.

dynamicCallOn: target

CustomProbeMap Class

```
//
// CustomProbeMap --
//   A subclass of ProbeMap, whose initial state is empty unlike the default
//   probeMap initial state which contains all the VarProbes
//   of the requested target class.
//
@protocol CustomProbeMap <ProbeMap, CREATABLE>
+ create: aZone forClass: (Class)aClass withIdenti-
fiers: (const char *)vars, ...;
@end
```

This subclass of the ProbeMap is used to create probe maps which are initialised in an emtpy state or with the VarProbes and MessageProbes intended. In other words, the probed class is set, as is the case with the normal ProbeMap class but upon createEnd no VarProbes or MessageProbes will be present within it. This feature is useful when creating a probe map from scratch (e.g. to be used in conjunction with the setProbeMap:For: message of the ProbeLibrary).

An additional convenience method was added to help compress the amount of code needed to create a custom probe.

**+ create: aZone forClass: (Class)aClass withIdenti-
fiers: (const char *)vars, ...;**

This is a variable argument message which takes a Zone, a Class name, and a colon-delimited list of instance variable names and message names valid for that class. The instance variable identifiers are listed first followed by a string consisting solely of a ":" followed by the message identifiers. The general format would be:

```
[CustomProbeMap create: aZone
               forClass: aClass
        withIdentifiers: "variable1", "variable2", ...,
                         ":",
                         "message1", "message2", ...,
                         NULL];
```

All strings up to the ":" are read as instance variable names and placed in the probe map and all strings after the ":" are read as message names and placed in the probe map. The NULL string terminates the list. An example of this message being used is given in the Advanced Usage Guide
(./AdvancedUsage.html)
.

CompleteProbeMap Class

```
//
// CompleteProbeMap --
//   A subclass of ProbeMap, whose initial state contains the VarProbes and
//   MessageProbes of the requested target class but also those of all its
//   superclasses.
//
@protocol CompleteProbeMap <ProbeMap> @end
```

Upon creation, this subclass of the ProbeMap will contain all the variables and all the messages of a given class (including the inherited ones).

(EmptyProbeMap.html)

(Interface.html)

(CompleteVarMap.html)

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

Last modified: 1998-01-10

```
  float randomMoveProbability;

  Grid2d * world;
  int worldXSize, worldYSize;
  HeatSpace * heat;
  Color bugColor;
}
```

The @interface keyword indicates that you are beginning the definition of the part of an object (a Heatbug in this case) that will be visible to other objects. The Heatbug: SwarmObject indicates that you are calling this object Heatbug and it is a subclass of SwarmObject. What follows between the curly braces ({}) are the instance variables defined for the Heatbug class above and beyond those inherited from the SwarmObject class.

Inside this "agent," we have defined several parameters associated with either the agent, itself, or the space in which it sits. Any data that will need to be present throughout all the behavior and lifetime of the agent should be declared here. Also, anything declared here will be accessible to the probe machinery, and so will be capable of being manipulated and viewed from outside the agent.

Next come the message prototypes to which this agent will respond. And it is worth noting again that these are in addition to those declared in the SwarmObject superclass. So, not only will other objects be able to send messages to this agent that are declared here, but other objects will be able to send all the messages declared in the objectbase/SwarmObject.h imported previously. The messages prototyped here will dictate what the compiler thinks this object can respond to. Hence, if any part of any of these prototypes differs from the corresponding function definition in the Heatbug.m file, then the compiler will say something like Object: aHeatbug does not respond to xyz, where "xyz" is the name of the message that is being sent to the Heatbug. A script is provided with the Swarm distribution that fixes header file prototypes to match the message declarations in the corresponding ".m" file. This script should be in the $SWARMHOME/bin directory and is called m2h.

One more thing to notice about these prototypes is that some of them are duplicates of what appears in the objectbase/SwarmObject.h file. This means that when the message is called on a Heatbug object, it will execute the method defined here and not the one in the SwarmObject class. In the objectbase library, the following messages are intended to be overridden, as necessry: create:, createBegin:, createEnd, customizeBegin:, customizeEnd, customizeCopy:, describe:, and getInstanceName. Each of these messages do specific things that may change from subclass to subclass of SwarmObject. In this case, however, we're only overriding createEnd. The differences between we implement it in Heatbugs and the default is not that significant. But, it should be pointed out that when overriding certain messages, like createBegin: and createEnd, the new method should call the superclass' version of the message, as well. This is done using the default pointer to the superclass, designated super. The syntax in the Heatbugs case is:

```
[super createEnd];
```

It should only be used to instantiate the control context and quit the entire simulation. That means that sometime in the future, the `Go` and the `Stop` buttons will be removed from the ControlPanel display. They have been left in for backwards compatibility so that applications that do not use the new ActivityControl will retain their (albeit handicapped) controllability. Also, the current `Time Step` button will be renamed to `Start` to be consistent with it's new purpose.

In order to use the new control mechanism, you must place code like the following in the top-level Swarm. (This code was taken from a modified mousetrap demo app.)

```
observerActCont = [ActivityControl createBegin: [self getZone]];
  observerActCont = [observerActCont createEnd];
  [observerActCont attachToActivity: [self getSwarmActivity]];
  [probeDisplayManager createProbeDisplayFor: observerActCont];
```

This creates an ActivityControl and attaches it to the top-level activity (in this case an `observerSwarm`). It also creates a display for the controller. (The probe map for the ActivityControl class is designed within the ActivityControl, itself. This is done because all of these objects are expected to look the same to any outside object.) With this activity controller, you will then be able to `run`, `stop`, `next`, `step`, `stepUntil`, and `terminate` that activity.

There are some tricky aspects to successfully using an ActivityControl object that the Advanced Usage Notes
(AdvancedUsage.html)
will cover.

(Interface.html)

(../objectbase.html)

(AdvancedUsage.html)

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

Last modified: 1998-01-10

```
                        "listOfFriends", "myColor", NULL];
```

And if the user wanted messages in the probe as well, it could be extended to:

```
probeMap = [CustomProbeMap create: [self getZone]
                           forClass: [person class]
                   withIdentifiers: "room", "party", "name",
                        "stillhere", "listOfFriends", "myColor",
                        ":",
                        "setWorld:Room:Party:",
                        "setPerson:Topic_array:ShowSpeech:",
                        NULL];
```

At present, this message doesn't search the superclasses for the message names listed here. But, that will soon be rectified.

2. ActivityControl Issues: It is completely reasonable to assume that explicit control can be had over all the activities in a simulation. However, at present, this control is limited because the context in which an activity runs determines how it behaves. To understand how an ActivityControl is to be used, we will have to explore the behavior of activities in context. (For a more complete explanation of the behavior of activities, see the activity library (../activity.html) documentation.)

   There are two ways to get an activity started, one can activate the activity in `nil` or in some other activity. So called "top-level" activities are activated in `nil` and are intended to be independent of the sophisticated scheduling activity that dictates the execution of actions in any other context in the simulation. I.e. the only activities that should be activated in `nil` are those sets of events that are expected to preserve the same behavior no matter what goes on in any other part of the simulation.

   The other type of activity, those activated in some other activity, is intended to be an integral part of its owner activity. However, this doesn't mean that it must depend on the outcome of actions happening in the owner activity. In fact, an ActionPlan can be designated as containing actions that are capable of being processed in parallel, via code like the following:

```
[anActionPlan setDefaultOrder: Concurrent];
```

But these activities are still intended to be meshed with their owner activities. In other words, they are part and parcel of the same model or simulation.

Now, the operational effect of activating an activity in `nil` is that it will not be meshed with the rest of the Swarm activity structure. This gives the user (or process) complete control over the execution of that activity. A `run` on that activity will run either to completion or until a stop flag is set by a sequence of events purely internal to that activity. Or, one can stop it from the outside with a message from something like an ActivityControl.

What all this means is that, while one can attach an ActivityControl to any activity, only the "top-level" activities (those having been activated in `nil`) are going to respond well to it. Any sub-activity will respond half-heartedly, if at all. For example, in the Mousetrap demo distributed with Swarm, an ActivityControl has been placed on both the ObserverSwarm and the ModelSwarm activities. Now,

Swarmobject Library: Interface Design Notes

Unfortunately, this interface has not undergone a rigorous design review. As such it is subject to change in the future. However, there are rumours that this library will be integrated into the defobj library
(../defobj.html)
anyway. So, even though little thought was given to the design of this interface (and it is not likely to be worthwhile designing a robust interface at this time), it was implemented in order to provide a first step towards bringing all the various libraries in line with the standard set by defobj.

Along these lines, a few notes are relevant.

1. Probes may become an inherent part of any object.
2. ActivityControls will become a part of a larger set of tools used for debugging Swarm models.
3. The explicit importation of the SwarmObject interface should be removed. This is only provided for backward compatibility and when all the libraries follow the interface conventions, then those conventions should be encouraged.

(AdvancedUsage.html)

(../objectbase.html)

(Implementation.html)

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

Last modified: 1998-01-10

Swarm Space Library

The Swarm Space library is the beginnings of a library to assist in building environments for interacting agents. In general, environments can be just as varied as the agents themselves (in one view, the environment itself is simply another agent). However, many simulations have similar types of environments that can be helpfully supported by generic code.

> The current space library only addresses simple kinds of discretized 2d space. Improvement is planned in the future: see the todo list
> (../todo.html)
> for ideas. Briefly, coordinates need to be elevated to the status of objects, which should hopefully allow spaces of different scales and boundary conditions to interact through a common reference system. In addition, other types of spaces are desired: continuous coordinates, other dimensions, arbitrary graphs, etc.

Discrete2d

Root class of all 2d discrete spaces. A Discrete2d is basically a 2d array of ids. Subclasses add particular space semantics onto this. Currently Discrete2d grids are accessed by integer pairs of X and Y coordinates.

Creation

- `- setSizeX: (int)x Y: (int)y`
  Set the world size.
- `- createEnd`
  Create the lattice, precompute the offsets based on Y coordinate.

Querying Discrete2d state

- `- (int)getSizeX` and `- (int)getSizeY`
  Get the size of the lattice.

Global Operations

- `- fastFillWithValue: (int)aValue`
- `- fastFillWithObject: anObject`
- `- fillWithValue: (int)aValue`

Miscellaneous

These methods are used in `- createEnd` to actually allocate the array. Subclasses might need to use these.

- `- makeOffsets`
  Given an array size, compute the offsets array that caches the multiplication by ysize. See the `discrete2dSiteAt` macro.
- `- (id *)allocLattice`
  Allocate memory for the lattice.

Grid2d

Grid2d is a generic container class to represent agent position on a 2d lattice. It gets most of its behaviour from Discrete2d, adding extra code to check that you don't overwrite things by accident. Grid2d is pretty primitive: only one object can be stored at a site, no boundary conditions are implied, etc. A fancier Grid2d is in the works.

New Methods

- `- setOverwriteWarnings: (BOOL)b`
  If set to true, then if you try to store something at a site that doesn't have 0x0 there, a warning will be generated.

Overridden methods

- `- putObject: anObject atX: (int)x Y: (int)y`
  Replaces the Discrete2d method. First check to see if it should do overwrite warnings, and if so if you're going to overwrite: if both conditions are true, print out a warning message. Regardless of the check, it writes the new object in.
- `+ createBegin: aZone`
  Make overwrite warnings be on by default.

DblBuffer2d

DblBuffer2d augments Discrete2d to provide a form of double buffered space. Two lattices are

Check that numStates has been set.

ConwayLife2d

Classic 2d Conway's Life CA.

Overridden Methods

+createBegin: aZone
    Set number of states to 2.

- initializeLattice
    Initialize lattice to random 1/3 in state 1.

- stepRule
    Run Conway's Life rule (simpleminded version).

Diffuse2d

Discrete 2nd order approximation to 2d diffusion with evaporation. Math is done in integers on the range [0,0x7fff].

New Methods

- setDiffusionConstant: (double)d
    Set the diffusion constant. Values over 1.0 might not be valid.

- setEvaporationRate: (double)d
    Set the evaporation rate. Values over 1.0 don't make much sense.

Overridden Methods

+createBegin: aZone
    Set diffusion constant and evaporation rate to 1.0, numStates to 0x7fff.

- initializeLattice
    Initialize world to 0.

- stepRule
    Run discrete approximation to diffusion. Roughly, it's

Value2dDisplay

Value2dDisplay helps display 2d arrays of values. Value2dDisplay goes through a given Discrete2d array, turn states into colours, and draws them into a Raster widget.

Creation Phase

- setDisplayWidget: (Raster *)r Colormap: (XColormap *)c
  Set the display widget and the colourmap to use to draw the value array.
- setDiscrete2dToDisplay: (Discrete2d *)c
  Set which array to draw.
- setDisplayMappingM: (int)m C: (int)c
  Linear transform of states to colours for drawing.
  color = state / m + c
  If not set, assume m == 1 and c == 0.

Use

- display
  Draw the array on the given widget. Note that you still have to tell the widget to draw itself afterwards. The code for display uses the fast macro access in Discrete2d on the cached return value from getLattice. It also caches the drawPointX:Y: method lookup on the display widget - this is a nice trick that you might want to look at.

Int2dFiler

The Int2dFiler class is used to save the state of any Discrete2d object (or a subclass thereof) to a specified file.

A Note on Output Formats

The issue of what format(s) the class should know about is still open. For the purposes that I have encountered, simply storing the values in space-delimited line format:

```
value0_0 value0_1 ...
value1_0 value1_1 ...
...
...
```

Random Library

Documentation and Implementation Status

Note: The new random library does not work in the same way as the old one. This means that some applications that used the random library provided with the 1.0.2 release will be broken. However, porting these applications to the new random library will be fairly easy since large efforts were made to adhere to the standard set with the last version and some backwards compatibility hooks were incorporated.

Porting to Random 0.7

Overview

First, a few thoughts on pseudorandom number generation. It's hard to do right. There are many problems: the root cause, of course, is that computer algorithms themselves are not random. But there are also problems with defining "random", coming up with good tests for generators, and implementing algorithms correctly and efficiently. The history of pseudorandom number generation in simulation work is mostly embarassing. This library attempts to do a decent job of generating random numbers, as well as documenting how things work and what shortcomings there are. If you want to learn more about random number generation, the bibliography [See: $(SWARMHOME)/src/random/docs/SOURCES.for.0.7] has useful notes. Knuth is the main reference in this realm, but too old to describe most of the particular generators used here.

The code here represents an effort to implement several efficient, reasonably safe generators. The algorithms come from reading the literature [See: $(SWARMHOME)/src/random/docs/SOURCES.for.0.7]: these algorithms have been implemented as accurately as possible and run through some simple tests. There is always a chance that some algorithm here is no good - there's also the chance that it is implemented incorrectly.

Random Library: Documentation and Implementation Status

This is version 0.7 of *Random*. It was donated by Sven Thommesen. Version 0.6 was a reimplementation of most of Nelson Minar's original random with many changes and a new interface. This version adds many more generators and distributions and changes the interface somewhat.

The implementation may be a little unstable. It hasn't been fully tested. Sven has also contributed a test suite which we will make available on the web site. But, as with any pseudo-random number generation library, results obtained from it should be examined very closely. But we are reasonably sure the generators and distributions have been correctly implemented.

Documentation

More documentation is available within the Swarm source distribution, itself [Ref: $(SWARMHOME)/src/random/docs/. These docs consist of:

WHATS.NEW.in.0.7
: Additions to Random since v. 0.6

WHATS.CHANGED.in.0.7
: Changes to Random since v. 0.6

WHATS.LEFT.TODO
: Improvements that may make it into v. 0.8

README.Generators.v07
: Usage Guide for generator objects

README.Distributions.v07
: Usage Guide for distribution objects

CHOOSING.A.GENERATOR
: Notes on choosing a generator for your sim

DEFAULT.GENERATORS
: Default generators invoked by distributions

SOURCES.for.0.7

Random Library: Porting to Random 0.7

The changes from version 0.6 to 0.7 consist mostly of a restructuring of the implementation of the library. However, there are some interface changes, as well, and the porting of applications written to work with the 0.6 version might require changes to accomodate the new interface. Also, it is advised that you take a look at the new interface, regardless of whether your application compiles and runs with the new random because the old interface is deprecated and will eventually be removed and because some of the functionality of the library changed such that the execution of your application may differ.

Minimalist Porting Notes:

What is described here is the minimum a normal random user would have to change to get her apps running with the 0.7. For complete documentation of the changes made to the random library, check the files:

$(SWARMHOME)/src/random/docs/WHATS.CHANGED.in.0.7
and
$(SWARMHOME)/src/random/docs/WHATS.NEW.in.0.7

1. The 'magic numbers' used by setState/getState to identify generators and distributions have changed. The sizes of the records returned by getState have also changed. Therefore version 0.7 generators will refuse to setStatea generator based on version 0.6 data.
2. The setState method has been renamed to setStateFrom, and getState to putStateInto.
3. The verifySelf methods have been exorcised. Their code has been moved to a separate test program.

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

Last modified: 1998-01-26

The two files 'methods.SimpleGenerators.h' and 'methods.SplitGenerators.h' in directory /random/docs show the complete sets of methods implemented for the two most common types of generator.

Generators

- InternalState
(Prot.html#InternalState)
- SingleSeed
(Prot.html#SingleSeed)
- MultiSeed
(Prot.html#MultiSeed)
- Simple
(Prot.html#Simple)
- SimpleOut
(Prot.html#SimpleOut)
- SplitSingleSeed
(Prot.html#SplitSingleSeed)
- SplitMultiSeed
(Prot.html#SplitMultiSeed)
- Split
(Prot.html#Split)
- SplitOut
(Prot.html#SplitOut)
- SingleShortGenerator
(Prot.html#SingleShortGen)
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed>
- SingleLongGenerator
(Prot.html#SingleLongGen)
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed, MultiSeed>
- CombinedShortGenerator
(Prot.html#CombinedShortGen)
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed, MultiSeed>
- CombinedLongGenerator
(Prot.html#CombinedLongGen)
<Create, Drop, InternalState, SimpleOut, Simple, SingleSeed, MultiSeed>
- SingleShortSplitGenerator
(Prot.html#SingleShortSplitGen)
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed>
- SingleLongSplitGenerator

- LCG2gen
  (LCG1-3gen.html#LCG2gen)
  : LCGgen <LCGgen, CREATABLE>
- LCG3gen
  (LCG1-3gen.html#LCG3gen)
  : LCGgen <LCGgen, CREATABLE>

- MRGgen
(MRG5-7gen.html)
<SingleLongGenerator>

  - MRG5gen
    (MRG5-7gen.html#MRG5gen)
    : MRGgen <MRGgen, CREATABLE>
  - MRG6gen
    (MRG5-7gen.html#MRG6gen)
    : MRGgen <MRGgen, CREATABLE>
  - MRG7gen
    (MRG5-7gen.html#MRG7gen)
    : MRGgen <MRGgen, CREATABLE>

- MT19937gen
(MT19937gen.html)
<SingleLongGenerator, CREATABLE>
- MWCAgen
(MWCA-Bgen.html)
<SingleLongGenerator, CREATABLE>
- MWCBgen
(MWCA-Bgen.html#MWCBgen)
<SingleLongGenerator, CREATABLE>
- PMMLCGgen
(PMMLCG1-9gen.html)
<SingleShortGenerator>

  - PMMLCG1gen
    (PMMLCG1-9gen.html#PMMLCG1gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG2gen
    (PMMLCG1-9gen.html#PMMLCG2gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG3gen
    (PMMLCG1-9gen.html#PMMLCG3gen)
    : PMMLCGgen <PMMLCGgen, CREATABLE>
  - PMMLCG4gen

- TGFSRgen
(TGFSRgen.html)
<SingleLongGenerator>

  - TT403gen
  (TGFSRgen.html#TT403gen)
  : TGFSRgen <TGFSRgen, CREATABLE>
  - TT775gen
  (TGFSRgen.html#TT775gen)
  : TGFSRgen <TGFSRgen, CREATABLE>
  - TT800gen
  (TGFSRgen.html#TT800gen)
  : TGFSRgen <TGFSRgen, CREATABLE>

Distributions

- ProbabilityDistribution
(Prot.html#ProbabilityDist)
<Create, Drop, InternalState>
- BooleanDistribution
(Prot.html#BooleanDistribution)
<ProbabilityDistribution>
- IntegerDistribution
(Prot.html#IntegerDistribution)
<ProbabilityDistribution>
- UnsignedDistribution
(Prot.html#UnsignedDistribution)
<ProbabilityDistribution>
- DoubleDistribution
(Prot.html#DoubleDistribution)
<ProbabilityDistribution>
- Normal
(Prot.html#Normal)
<DoubleDistribution>
-
  RandomBitDist
  (RandomBitDist.html)
  <BooleanDistribution, CREATABLE>
- BernoulliDist
(BernoulliDist.html)
<BooleanDistribution, CREATABLE>

Random Library: Protocol Reference

InternalState

```
//
// InternalState --
//   Methods to save the internal state of an object (genera-
tor, distribution)
//   to a memory buffer allocated by the calling pro-
gram, and to set the state
//   of an object from previously saved state data, provided in a mem-
ory buffer.
//   Method -getStateSize specifies the minimum buffer size needed.
//
@protocol InternalState
- (unsigned) getStateSize; // size of buffer needed
- (void) putStateInto: (void *) buffer; // save state data for later use
- (void) setStateFrom: (void *) buffer; // set state from saved data
- (void) describe: (id) outStream; // prints ascii data to stream
- (char *)      getName; // returns name of object
- (unsigned) getMagic; // object's 'magic number'
@end
```

The InternalState protocol provides attributes common to the generators and distributions provided in the random library. This protocol could be used for any class of objects that might contain an internal state, including things unrelated to pseudo random number generation.

The "state" of a given object depends on the type of object it is. In the case of random, both the bit generators and the distributions are capable of carrying state. The state is, in all cases, a structure that is understood by the object with which it is associated. The details of this structure can be found in the code for the given generator or distribution; but, in general, the user is not expected to understand the structure. Where the state must be saved and used to restart, the user is expected only to read and write the block of memory from and to the disk and pass the pointer to and from the object with which the state is associated. [See the Usage Guide (./UsageGuide.html) for examples of how to do this.] Please note that different computer architectures store binary data in memory in different ways (using different byte orders). Since this version of the InternalState protocol stores data in binary form, you cannot expect to be able to use these methods to transfer data between different computers. We hope to lift this restriction in a future version.

(unsigned) -getStateSize returns an unsigned integer indicating how many bytes of memory are necessary to save the state of the object.

```
USING
- setStateFromSeeds: (unsigned *) seeds;
- (unsigned)  lengthOfSeedVector;
- (unsigned *) getMaxSeedValues;
- (unsigned *)  getInitialSeeds;
@end
```

Simple

```
@deftype Simple
CREATING
+ createWithDefaults: aZone;
USING
- setAntithetic: (BOOL) antiT;
- (BOOL) getAntithetic;
- (unsigned long long int) getCurrentCount;
@end
```

SimpleOut

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

SplitSingleSeed

```
@deftype SplitSingleSeed
CREATING
+ create: aZone
   setA: (unsigned) A  // # of virtual generators
   setv: (unsigned) v  // log2(#segments/generator)
   setw: (unsigned) w // log2(segment length)
setStateFromSeed: (unsigned) seed;
USING
- setStateFromSeed: (unsigned) seed;
- (unsigned) getMaxSeedValue;
- (unsigned) getInitialSeed;
@end
```

SplitMultiSeed

```
@deftype SplitMultiSeed
CREATING
+ create: aZone
   setA: (unsigned) A  // # of virtual generators
   setv: (unsigned) v  // log2(#segments/generator)
```

No extra messages added.

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

No extra messages added.

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

No extra messages added.

SingleShortSplitGenerator
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed>

No extra messages added.

SingleLongSplitGenerator
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

No extra messages added.

CombinedShortSplitGenerator
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

No extra messages added.

CombinedLongSplitGenerator
<Create, Drop, InternalState, SplitOut, Split, SplitSingleSeed, SplitMultiSeed>

No extra messages added.

Distribution Protocols

ProbabilityDistribution
<Create, Drop, InternalState>

```
// ProbabilityDistribution --
//
// A process for generating a sequence of random numbers matching the
//   frequencies defined by a specific distribution function.  The
//   process is driven by input from a supplied uniform random generator.
//
@deftype ProbabilityDistribution <Create, Drop, InternalState>
CREATING
+ createWithDefaults: aZone;
+ create: aZone setGenerator: (id) generator;
+ create: aZone setGenerator: (id) generator
setVirtualGenerator: (unsigned) vGen;
+ createBegin;
USING
```

```
//   values as represented by double-precision floating point values.
//
@deftype DoubleDistribution <ProbabilityDistribution>
- (double) getDoubleSample;
@end
```

Normal
<DoubleDistribution>

```
//
// NormalDistribution -- a well-known continuous probability distribution
// LogNormalDistribution -- also well-known ...
//
@deftype Normal <DoubleDistribution>
CREATING
+ create: aZone setGenerator: generator
                setMean: (double) mean setVariance: (double) variance;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setMean: (double) mean setVariance: (double) variance;
USING
- setMean: (double) mean setVariance: (double) variance;
- (double) getMean;
- (double) getVariance;
- (double) getStdDev;
- (double) getSampleWithMean: (double) mean
withVariance: (double) variance;
@end
```

(./Interface.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

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: [1,2^32-2]

Cycles:  | With properly chosen parameters, these generators
| have a cycle of length about 2^r.

Output Quality:  | Low order bits are poor, and there are other problems.
| SWB is a better generator in the same class, and it is
| only slightly slower.

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

ACGgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                      2.702 uS
|   -getFloatSample, getDoubleSample:         4.482 uS
|   -getFatDoubleSample, getLongDoubleSample: 8.299 uS

Relative speed:
| Speed 1.369 (time 0.731) relative to MT19937 getUnsignedSample
--------------- | ----------------------------------------------------------
---

Parameters:  | r = 55
| s = 24

Reference for | These values recommended by Knuth.
parameters:  |

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

Output Quality:  | Low-order bits are poor.
```

C2LCGXgen Class

```
//
// C2LCGX --
//   short component based generator with splitting facilities. Recommended.
//
@deftype C2LCGXgen  <CombinedShortSplitGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | C2LCGX

Description:  | Combined random generator using 2 (PMM)LGC generators

Reference:  | Pierre L'Ecuyer and Serge Cote,
| "Implementing a Random Number Package with Splitting
| Facilities." ACM Transactions on Mathematical Software
| vol. 17 no. 1, March 1991, pp. 98-111.
| Their name for the generator: 'split.c'.
| See also <http://www.iro.umontreal.ca/~lecuyer>.

Comments:  | This portable generator is based on a backbone generator
| which is a combination of 2 (PMM)LCG generators. It has a
| period length of almost 2^61 (2.3e18).

| The backbone generator's period can be split up into a
| number of 'virtual generators' (A), each of which can be set
| to access a number of 'segments' (V) of length W, subject to
| the constraint that A * V * W <= 2^60.

| This is possible because the backbone generator has
| 'jump-ahead facilities' -- i.e. a quick method of calculating
| the state of the generator at a given point in its cycle
| without having to actually draw a large number of variates.

Algorithm | state(j) = seed(j)
for each | newstate(j) = (a(j) * oldstate(j)) mod m(j)
generator:  | output(j) = state(j)

Algorithm | output = (state(1) - state(2)) mod m(0)
for combination:|
```

| straightforward. Unfortunately, on x86 at least, there is
| a significant speed penalty for doing so.
| The output, however, would be identical. YMMV.

C2LCGXgen | On a 486/66, the following approximate measures were obtained:
speed: | -getUnsignedSample: 7.029 uS
| -getFloatSample, getDoubleSample: 9.049 uS
| -getFatDoubleSample, getLongDoubleSample: 16.725 uS

Relative speed:
| Speed 0.526 (time 1.900) relative to MT19937 getUnsignedSample
--------------- . --
--

Parameters: | a[COMPONENTS] = { 40014, 40692 }
| m[COMPONENTS] = { 2147483563, 2147483399 }

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

Cycles: | With the chosen parameters, this generator
| has a single full cycle of length close to 2^60 (2.3e18).

Output Quality: | The authors performed an exhaustive search for component
| generators that would yield the best statistical quality.

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

(./ACGgen.html)

(./Interface.html)

(./C2MRG3gen.html)

C2MRG3gen Class

```
//
// C2MRG3 --
//   long component based generator recommended for use.
//
@deftype C2MRG3gen <CombinedLongGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | C2MRG3

Description:  | Combination of 2 Multiple Recursive LCG Generators

Reference:  | Pierre L'Ecuyer,
| "Combined Multiple Recursive Random Number Generators."
| See <http://www.iro.umontreal.ca/~lecuyer>.

| For a discussion of MRG generators, see MRGgen.h.

Algorithm:  | For each generator:
| x_n = (a_1*x_n-1 + a_2*x_n-2 ... + a_k*x_n-k) mod m
| Combining them:
| x = (x1 - x2) mod m1

Comments:  | Combinations of like generators are shown to have better
| statistical properties than single generators.

Implementation:  | The components of this generator each has two nonzero
| multipliers (and one that's zero). They use different
| moduli (2^31-1, 2145483479.)

Parameters:  | unsigned int m[2] // modulus
| unsigned int a1[2], ak[2] // multipliers
| unsigned int k[2] // the order of the generator

State:  | unsigned int x[k][2] // 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.
```

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

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

(./C2LCGXgen.html)

(./Interface.html)

(./C2TAUS1-3gen.html)

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

Last modified: 1998-01-26

```
| unsigned int Mask[2] // 2^P - 1

State:  | unsigned int state[2] // 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 [0,2^31-1] may be returned
| (i.e. [0,2147483647].)

Valid seeds:  | Using create:setStateFromSeed: in [1, 2^32-1]
| Using create:setStateFromSeeds:
|  Seed[0] must be in [1,Mask[0]];
|  Seed[1] must be in [1,Mask[1]];

Cycles:  | These generators have a single full cycle
| of length Mask1 * Mask2 ~= 2^60 (1.15e18).

Output Quality:  | The authors conducted an exhaustive search among candidate
| Tausworthe generators to find the combination that had the
| best lattice structure. (Single Tausworthe generators do not
| have acceptable properties.) The 3 generators implemented
| here were retained as the best found.

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

C2TAUSgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        4.078 uS
|   -getFloatSample, getDoubleSample:          5.932 uS
|   -getFatDoubleSample, getLongDoubleSample: 10.920 uS

Relative speed:
| Speed 0.907 (time 1.103) relative to MT19937 getUnsignedSample
```

C2TAUS1gen Documentation

```
Parameters:  | P = 31 so Mask = 2^31-1
(component 1) | S = 12 so PmS = 19
| Q = 13

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

parameters: | op. cit., table III 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: |

(./C2MRG3gen.html)

(./Interface.html)

(./C3MWCgen.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,unsignedMax] = [0,2^32-1].

Valid seeds:  | Using create:setStateFromSeed:  [1, 2^32-1]
| Using create:setStateFromSeeds: each component may have
| a 'state' <= 0xffff and a carry <= (multiplier-1), so that
| for the first component the largest seed allowed is
| (a-1)<<16 + 65535.

Cycles:  | This generator is claimed to be strictly periodic, with a
| period > 2^118. There may be more than one cycle.

Output Quality:  | Good.

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

C3MWCgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        6.437 uS
|   -getFloatSample, getDoubleSample:          8.200 uS
|   -getFatDoubleSample, getLongDoubleSample: 15.562 uS

Relative speed:
| Speed 0.574 (time 1.741) relative to MT19937 getUnsignedSample
--------------- . -----------------------------------------------------------
--
```

(./C2TAUS1-3gen.html)

(./Interface.html)

(./C4LCGXgen.html)

C4LCGXgen Class

```
//
// C4LCGXgen --
//   short component based generator with splitting facilities. Recommended.
//
@deftype C4LCGXgen  <CombinedShortSplitGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | C4LCGX

Description:  | Combined random generator using 4 (PMM)LGC generators

Reference:  | Pierre L'Ecuyer and Terry H. Andres,
| "A Random Number Generator Based on the Combination
|  of Four LCGs", 17 December 1996.
| To appear in Mathematics and Computers in Simulation. See:
| <http://www.iro.umontreal.ca/~lecuyer/myftp/papers/clcg4.ps>.
| Their name for the generator: 'clcg4'.

Comments:  | This portable generator is based on a backbone generator
| which is a combination of 4 (PMM)LCG generators. It has a
| period length of (m1-1)(m2-1)(m3-1)(m4-1) / 8, or almost
| 2^121 (2.6e36).

| The backbone generator's period can be split up into a
| number of 'virtual generators' (A), each of which can be set
| to access a number of 'segments' (V) of length W, subject to
| the constraint that A * V * W <= 2^120.

| This is possible because the backbone generator has
| 'jump-ahead facilities' -- i.e. a quick method of calculating
| the state of the generator at a given point in its cycle
| without having to actually draw a large number of variates.

Algorithm | state(j) = seed(j)
for each | newstate(j) = (a(j) * oldstate(j)) mod m(j)
generator:  | output(j) = state(j)

Algorithm | output = (state(1) - state(2) + state(3) - state(4)) mod m(0)
for combination:|
```

```
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
| straightforward. Unfortunately, on x86 at least, there is
| a significant speed penalty for doing so.
| The output, however, would be identical. YMMV.

C4LCGXgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       11.688 uS
|   -getFloatSample, getDoubleSample:         13.644 uS
|   -getFatDoubleSample, getLongDoubleSample: 26.154 uS

Relative speed:
| Speed 0.316 (time 3.160) relative to MT19937 getUnsignedSample
--------------- | -----------------------------------------------------------
---

Parameters:  | a[COMPONENTS] = {      45991,     207707,
|      138556,      49689 }
| m[COMPONENTS] = { 2147483647, 2147483543,
|  2147483423, 2147483323 }

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

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length close to 2^121 (2.6e36).

Output Quality:  | The authors performed an exhaustive search for component
| generators that would yield the best statistical quality.

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

(./C3MWCgen.html)

(./Interface.html)

LCG1-3gen Classes

```
//
// LCG1, LCG2, LCG3 --
//   single short random number generators,
//   NOT recommended for serious use; these are included for
//   historical reasons (compatibility with earlier releases).
//
@deftype LCGgen      <SingleShortGenerator> @end
@deftype LCG1gen        <LCGgen, CREATABLE> @end
@deftype LCG2gen        <LCGgen, CREATABLE> @end
@deftype LCG3gen        <LCGgen, CREATABLE> @end
```

Use of these generators is NOT Recommended

General Documentation

```
Generator name:  | LCG

Description:  | Linear Congruential Generator

Reference:  | This is a classic generator.
| See for example Donald Knuth,
| The Art of Computer Programming,
| vol. II, "Seminumerical Algorithms."

| state = seed
Algorithm:  | newstate = ((a * oldstate) + c ) mod 2^32
| output = state

Comments:  | This generator relies on controlled overflow at 32 bits.
| This requires that unsigned be a 32bit value that follows
| ANSI C rules.
| Knuth claims that the adder c does not matter much, as
| long as it has no factors in common with the modulus 2^32.


Parameters:  | unsigned int c // adder
| unsigned int a // multiplier
```

LCG2gen Documentation

```
Parameters:  | a =        69,069
| c = 1,013,904,223

Reference for | The multiplier is from Knuth vol II (line 25 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:  |
```

LCG3gen Documentation

```
Parameters:  | a =   1,664,525
| c = 152,193,325

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:  |
```

(./C4LCGXgen.html)

(./Interface.html)

MRG5-7gen Classes

```
//
// MRG5, MRG6, MRG7 --
//   single long generators recommended for use.
//
@deftype MRGgen      <SingleLongGenerator>  @end
@deftype MRG5gen      <MRGgen, CREATABLE> @end
@deftype MRG6gen      <MRGgen, CREATABLE> @end
@deftype MRG7gen      <MRGgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | MRG

Description:  | Multiple Recursive LCG Generator

Reference:  | Pierre L'Ecuyer, Francois Blouin and Raymond Couture,
| "A Search for Good Multiple Recursive Random Generators."
| See <http://www.iro.umontreal.ca/~lecuyer>.

Algorithm:  | x_n = (a_1*x_n-1 + a_2*x_n-2 ... + a_k*x_n-k) mod m

Comments:  | These generators require k multipliers and k past values
| to be kept. In their paper, the authors investigate MRG's
| of order k from 1 to 7. They provide several sets of
| parameters which they recommend out of a large number that
| were tested. Generally, the quality of the generators
| increases with k.

Implementation:  | The authors discovered that most of the benefit of these
| generators can be had if only a_1 and a_k are nonzero and the
| rest zero; this yields faster execution since only 2 terms
| need be calculated. We implement 4 such generators here,
| all with modulus m close to 2^31.


Parameters:  | unsigned int m // modulus
| unsigned int a1, ak // multipliers
| unsigned int k // the order of the generator

State:  | unsigned int x[k] // state vector
```

```
|   -getFloatSample, getDoubleSample:         12.764 uS
|   -getFatDoubleSample, getLongDoubleSample: 24.627 uS
```

Relative speed:
| Speed 0.339 (time 2.951) relative to MT19937 getUnsignedSample

MRG5gen Documentation

```
Parameters:  | unsigned int k = 5 // lag order
| unsigned int m = 2^31 - 1 // modulus
| unsigned int a1 = 107374182 // multiplier
| unsigned int a5 = 104480 // 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)^5 - 1,
| i.e. 2^154 < cycle < 2^155.

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

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

MRG6gen Documentation

```
Parameters:  | unsigned int k = 6 // lag order
| unsigned int m = 2^31 - 1 // modulus
| unsigned int a1 = 177786 // multiplier
| unsigned int a6 = 64654 // 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)^6 - 1,
| i.e. 2^185 < cycle < 2^186.

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

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

MT19937gen Class

```
//
// MT19937 --
//   single *very* long generator recommended for use.
//
@deftype MT19937gen  <SingleLongGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | MT19937

Description:  | Twisted GFSR generator

Reference:  | Makoto Matsumoto and ??? Nishimura,
| code posted to the pLab web site, xx/xx/97.
| Site: <http://random.mat.sbg.ac.at>
| Code: <ftp://random.mat.sbg.ac.at/pub/data/mt19937b-int.c>
| Article reportedly submitted to ACM TOMACS.

|
Algorithm:  |
|

Comments:  |

Implementation:  |

Parameters:  | unsigned int N // size of state array
| unsigned int M // lag
| unsigned int a // "Magic numbers"
| unsigned int b,c //    "     "
| unsigned int u,s,t,l //    "     "

State:  | unsigned int state[N] // 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.)
```

(./MRG5-7gen.html)

(./Interface.html)

(./MWCA-Bgen.html)

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

Last modified: 1998-01-26

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

MWCBgen Documentation

Generator name: | MWCB

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&65535);

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

Implementation: | This generator implements an alternate manner of
| conjoining the two components (differs from MWCA).

(./Interface.html)

(./PMMLCG1-9gen.html)

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

Last modified: 1998-01-26

```
| The choice of a good multiplier a is crucial.

Parameters:  | unsigned int m // modulus
| unsigned int a // multiplier

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

Output types:  | unsigned int (we return the current state-1).
| 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:  | The basic algorithm yields output in the range [1,m-1].
| We subtract 1 to return output in the range [0,m-2]
| (i.e. [0,2147483645].)

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

Cycles:  | With properly chosen parameters, these generators
| have a single full cycle of length (m-1).
| (State=0 is an absorbing state which will not normally
| occur; we avoid setting the starting state to 0.)

Output Quality:  | All bits should be safe.
| (I.e. low order bits should not have small cycles.)
| With m prime, this generator is better than standard LCG.

Reference for | Park and Miller, 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
| straightforward. Unfortunagely, on x86 at least, there
| is a 35% speed penalty for doing so. The output, however,
| is identical. YMMV.

PMMLCGgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                        4.715 uS
|   -getFloatSample, getDoubleSample:          6.500 uS
|   -getFatDoubleSample, getLongDoubleSample: 12.152 uS

Relative speed:
| Speed 0.792 (time 1.263) relative to MT19937 getUnsignedSample
```

```
choice of  |
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:  |
```

PMMLCG4gen Documentation

```
Parameters:  | a =        45,991
| m = 2,147,483,647 (2^31-1)

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:  |
```

PMMLCG5gen Documentation

```
Parameters:  | a =       207,707
| m = 2,147,483,543 (2^31-105)

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).
```

PMMLCG8gen Documentation

```
Parameters:  | a =        40,014
| m = 2,147,483,563 (2^31-85)

Reference for | Pierre L'Ecuyer and Serge Cote,
choice of  | "Implementing a Random Number Package with Splitting
parameters:  | Facilities," ACM Trans. Math. Softw. 17 no 1 (March 1991).

| This is one of the component generators of their generator.

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 Cote, op.cit.
output quality:  |
```

PMMLCG9gen Documentation

```
Parameters:  | a =        40,692
| m = 2,147,483,399 (2^31-249)

Reference for | Pierre L'Ecuyer and Serge Cote,
choice of  | "Implementing a Random Number Package with Splitting
parameters:  | Facilities," ACM Trans. Math. Softw. 17 no 1 (March 1991).

| This is one of the component generators of their generator.

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 Cote, op.cit.
output quality:  |
```

(./MWCA-Bgen.html)

PSWBgen Class

```
//
// PSWB --
//   single long generator recommended for use.
//
@deftype PSWBgen   <SingleLongGenerator, CREATABLE> @end
```

Documentation

```
Generator name:  | PSWB

Description:  | Subtract-with-Borrow Congruential Generator
|    with Prime Modulus

Reference:  | George Marsaglia and Arif Zaman,
| "A New Class of Random Number Generators",
| Annals of Applied Probability vol 1 no 3 (1991), 462-480.
| SWB is an improvement on the Lagged Fibonacci generators.

| PSWB is an improvement on SWB in that the use of a prime
| modulus guarantees a single full cycle.
| It's slower, of course.

| state vector = f(seed)
Algorithm:  | state_n = (state_(n-s) - state_(n-r) - carry) mod m; r > s
| output = state_n

| 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
| unsigned int s // short lag
```

```
Cycles:  | With the chosen parameters, this generator
| has a single cycle of length close to 10^414.

Output Quality:  | Good.

Reference for | Marsaglia, op.cit., p. 479
output quality:  |
```

(./PMMLCG1-9gen.html)

(./Interface.html)

(./RWC28gen.html)

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

Last modified: 1998-01-26

| Using create:setStateFromSeeds: x1 and x2: [1, 2^32-1]
| The carry can only be [1,a-1].

Cycles: | This generator is claimed to have a period > 2^92.
| It may have rho sequences up to length r=2.
| It may have multiple cycles of the stated period length.

Output Quality: | Good.

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

RWC2gen | On a 486/66, the following approximate measures were obtained:
speed: | -getUnsignedSample: 7.445 uS
| -getFloatSample, getDoubleSample: 8.845 uS
| -getFatDoubleSample, getLongDoubleSample: 17.338 uS

Relative speed:
| Speed 0.497 (time 2.013) relative to MT19937 getUnsignedSample

RWC8gen Documentation

Generator name: | RWC8

Description: | 8-lag Recursion-With-Carry generator
| ("The Mother of all RNG's")

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: | x = sum(ai*xi) + carry mod 2^16
| y = sum(bi*yi) + carry mod 2^16
| return ((x<<16)+y)

Comments: | This generator is a combination of 2 16-bit
| 8-lag Recursion-With-Carry generators.

Parameters: | unsigned int a[8] =
| { 1941, 1860, 1812, 1776, 1492, 1215, 1066, 12013 }

| unsigned int b[8] =
| { 1111, 2222, 3333, 4444, 5555, 6666, 7777, 9272 }

(./SCGgen.html)

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

Last modified: 1998-01-26

```
| unsigned int s // short lag
| unsigned int m // modulus

State:  | unsigned int state[r] // 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 [0,10^9-1] may be returned
| (i.e. [0,999999999].)

Valid seeds:  | Using create:setStateFromSeed:  [1,2^32-1]
| Using create:setStateFromSeeds: [1,999999998]

Cycles:  | With properly chosen parameters, these generators
| have a cycle of length about 2^r.

Output Quality:  | Low order bits are poor, and there are other problems.
| SWB is a better generator in the same class, and it is
| only slightly slower.

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

SCGgen | On a 486/66, the following approximate measures were obtained:
speed:  |   -getUnsignedSample:                       3.328 uS
|   -getFloatSample, getDoubleSample:         5.058 uS
|   -getFatDoubleSample, getLongDoubleSample: 9.274 uS

Relative speed:
| Speed 1.111 (time 0.900) relative to MT19937 getUnsignedSample
--------------- | -----------------------------------------------------------
---

Parameters:  | r = 55
| s = 24
| m = 10^9

Reference for |
parameters:  |

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

SWB1-3gen Classes

```
//
// SWB1, SWB2, SWB3 --
//   single long generators recommended for use.
//
@deftype SWBgen   <SingleLongGenerator> @end
@deftype SWB1gen    <SWBgen, CREATABLE> @end
@deftype SWB2gen    <SWBgen, CREATABLE> @end
@deftype SWB3gen    <SWBgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | SWB

Description:  | Subtract-with-Borrow Congruential Generator

Reference:  | George Marsaglia and Arif Zaman,
| "A New Class of Random Number Generators",
| Annals of Applied Probability vol 1 no 3 (1991), 462-480.
| SWB is an improvement on the Lagged Fibonacci generators.

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

| 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.

Note:  | 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.)
```

```
| s = 24

Reference for | Marsaglia, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has 64 cycles of length 10^354

Output Quality:  | Good.

Reference for | Marsaglia, op.cit., p. 479
output quality:  |
```

SWB2gen Documentation

```
Parameters:  | r = 24
| s = 19

Reference for | Marsaglia, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has 1536 cycles of length 10^228

Output Quality:  | Good.

Reference for | Marsaglia, op.cit., p. 479
output quality:  |
```

SWB3gen Documentation

```
Parameters:  | r = 21
| s =  6

Reference for | Marsaglia, op. cit.
parameters:  |

Cycles:  | With the chosen parameters, this generator
| has 192 cycles of length 10^200

Output Quality:  | Good.

Reference for | Marsaglia, op.cit., p. 479
output quality:  |
```

TGFSRgen Classes

```
//
// TT403, TT775, TT800 --
//   single long generators recommended for use.
//
@deftype TGFSRgen    <SingleLongGenerator>  @end
@deftype TT403gen     <TGFSRgen, CREATABLE> @end
@deftype TT775gen     <TGFSRgen, CREATABLE> @end
@deftype TT800gen     <TGFSRgen, CREATABLE> @end
```

General Documentation

```
Generator name:  | TGFSR

Description:  | Twisted GFSR generator

Reference:  | Makoto Matsumoto and Yoshiharu Kurita,
| "Twisted GFSR Generators II".
| ACM TOMACS, vol 4 no 3 July 1994 pp. 254-266.
| Amended by M. Matsumoto, July 8 1996.

|
Algorithm:  |
|

Comments:  |

Implementation:  |

Parameters:  | unsigned int w // word length
| unsigned int N // size of state array
| unsigned int M // lag
| unsigned int s // 'magic vector'
| unsigned int t //    "     "
| unsigned int b //    "     "
| unsigned int c //    "     "
| unsigned int a //    "     "

State:  | unsigned int state[N] // state vector

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

```
| and empirical testing.

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

TT775gen Documentation

```
Parameters:  | unsigned int w=31 // word length
| unsigned int N=25 // size of state array
| unsigned int M=8 // lag
| unsigned int s=6 // 'magic vector'
| unsigned int t=14 //    "     "
| unsigned int b=0x1abd5900 //    "     "
| unsigned int c=0x776a0000 //    "     "
| unsigned int a=0x6c6cb38c //    "     "

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

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length 2^775 -1.

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

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

TT800gen Documentation

```
Parameters:  | unsigned int w=32 // word length
| unsigned int N=25 // size of state array
| unsigned int M=7 // lag
| unsigned int s=7 // 'magic vector'
| unsigned int t=15 //    "     "
| unsigned int b=0x2b5b2500 //    "     "
| unsigned int c=0xdb8b0000 //    "     "
| unsigned int a=0x8ebfd028 //    "     "

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

Cycles:  | With the chosen parameters, this generator
| has a single full cycle of length 2^800 -1.
```

RandomBitDistribution Class

```
//
// RandomBitDistribution --
//   A generator that returns uniformly distributed single bit values
//   (i.e. fair coin tosses).
//
@deftype RandomBitDist <BooleanDistribution, CREATABLE>
USING
- (BOOL) getCoinToss;
@end
```

Documentation

```
Name:  | RandomBitDistribution

Description:  | Distribution returning YES and NO with equal probability

Algorithm:  | Extract a middle bit from an unsigned random number

Reference:  | (none)

Parameters:  | (none)

State:  | unsigned long long int currentCount

RandomBitDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getBooleanSample:                         5.656 uS
|  -average # of generator calls per variate: 1.000 (U)
|  -distribution net time:                    1.958 uS

Relative speed:
| Speed 0.654 (time 1.529) relative to MT19937 getUnsignedSample
--------------- . ----------------------------------------------------------
--
```

BernoulliDistribution Class

```
//
// BernoulliDistribution --
distribution returning YES with given probability p
//
@deftype BernoulliDist <BooleanDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setProbability: (double) p;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setProbability: (double) p;
USING
- setProbability: (double) p;
- (double) getProbability;
- (BOOL)        getSampleWithProbability: (double) p;
@end
```

Documentation

```
Name:  | BernoulliDistribution

Description:  | Distribution returning binary value YES with probability p

Algorithm:  | Rejection method

Reference:  | Code contributed by Barry McMullin

Parameters:  | double Probability

State:  | unsigned long long int currentCount

BernoulliDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getBooleanSample:                        8.156 uS
|  -average # of generator calls per variate: 1.000 (tD)
|  -distribution net time:                   2.654 uS

Relative speed:
| Speed 0.453 (time 2.206) relative to MT19937 getUnsignedSample
```

UniformIntegerDist Class

```
//
// UniformInteger --
//   A generator of integral values uniformly distributed across a closed
//   interval [min,max]. (The interval includes both its endpoints.)
//   Setting minValue == maxValue is allowed (and returns minValue).
//
@deftype UniformIntegerDist <IntegerDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setIntegerMin: (int) minValue setMax: (int) maxValue;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setIntegerMin: (int) minValue setMax: (int) maxValue;
USING
- setIntegerMin: (int) minValue setMax: (int) maxValue;
- (int) getIntegerMin;
- (int) getIntegerMax;
- (int) getIntegerWithMin: (int) minValue withMax: (int) maxValue;
@end
```

Documentation

```
Name:  | UniformInteger

Description:  | Distribution returning integers over an interval [min,max]

Algorithm:  | Transformation of uniform unsigneds [0,unsignedMax]

Reference:  | (none)

Parameters:  | integer minValue
| integer maxValue

State:  | unsigned long long int currentCount

Implementation  | Setting minValue=maxValue is allowed.
comment:        | Then minValue is retured "with probability 1".

UniformIntegerDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:
```

UniformUnsignedDist Class

```
//
// UniformUnsigned --
//   A generator of non-negative integral values uniformly distributed across
//   a closed interval [min,max].  (The interval in-
cludes both its endpoints.)
//   Setting minValue == maxValue is allowed (and returns minValue).
//
@deftype UniformUnsignedDist <UnsignedDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setUnsignedMin: (unsigned) minValue setMax: (un-
signed) maxValue;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setUnsignedMin: (unsigned) minValue setMax: (un-
signed) maxValue;
USING
- setUnsignedMin: (unsigned) minValue setMax: (unsigned) maxValue;
- (unsigned) getUnsignedMin;
- (unsigned) getUnsignedMax;
- (unsigned) getUnsignedWithMin: (unsigned) minVal
           withMax: (unsigned) maxVal;
@end
```

Documentation

```
Name:  | UniformUnsigned

Description:  | Distribution returning unsigneds over an interval [min,max]

Algorithm:  | Transformation of uniform unsigned [0,unsignedMax]

Reference:  | (none)

Parameters:  | unsigned minValue
| unsigned maxValue

State:  | unsigned long long int currentCount

Implementation  | Setting minValue=maxValue is allowed.
```

UniformDoubleDist Class

```
//
// UniformDouble --
//   A generator of floating point values uniformly distributed
//   across a half-open interval [min,max). (The interval includes
//   the lower endpoint but excludes the upper endpoint.)
//   NOTE: Setting minValue == maxValue is allowed (and returns minValue).
//
@deftype UniformDoubleDist <DoubleDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setDoubleMin: (double) minValue setMax: (double) maxValue;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setDoubleMin: (double) minValue setMax: (double) maxValue;
USING
- setDoubleMin: (double) minValue setMax: (double) maxValue;
- (double) getDoubleMin;
- (double) getDoubleMax;
- (double) getDoubleWithMin: (double) minValue withMax: (double) maxValue;
@end
```

Documentation

```
Name:  | UniformDouble

Description:  | Distribution returning doubles over an interval [min,max)

Algorithm:  | Transformation of uniform double [0.0,1.0)

Reference:  | (none)

Parameters:  | double minValue
| double maxValue

State:  | unsigned long long int currentCount

Implementation | Setting minValue=maxValue is allowed.
comment:  | Then minValue is returned "with probability 1".

UniformDoubleDist | On a 486/66, the following measures were obtained,
```

ExponentialDistribution Class

```
//
// ExponentialDistribution -- a well-
known continuous probability distribution
//
@deftype ExponentialDist <DoubleDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
                setMean: (double) mean;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
                setMean: (double) mean;
USING
- setMean: (double) mean;
- (double) getMean;
- (double) getSampleWithMean: (double) mean ;
@end
```

Documentation

```
Name:  | ExponentialDistribution

Description:  | Exponential distribution returning double values

Algorithm:  | Transformation method

Reference:  | Russell 1992

Parameters:  | double Mean

State:  | unsigned long long int currentCount

ExponentialDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getDoubleSample:                          20.905 uS
|  -average # of generator calls per variate:  1.000 (D)
|  -distribution net time:                    10.746 uS

Relative speed:
| Speed 0.177 (time 5.653) relative to MT19937 getUnsignedSample
```

GammaDistribution Class

```
//
// GammaDistribution -- a well-known continuous probability distribution
//
@deftype GammaDist <DoubleDistribution, CREATABLE>
CREATING
+ create: aZone setGenerator: generator
setAlpha: (double) alpha setBeta: (double) beta;
+ create: aZone setGenerator: generator
setVirtualGenerator: (unsigned) vGen
setAlpha: (double) alpha setBeta: (double) beta;
USING
- setAlpha: (double) alpha setBeta: (double) beta;
- (double) getAlpha;
- (double) getBeta;
- (double) getSampleWithAlpha: (double) alpha
withBeta: (double) beta;
@end
```

Documentation

```
Name:  | GammaDistribution

Description:  | Gamma distribution returning double values

Algorithm:  | Transformation, rejection and xxx methods.

Reference:  | Watkins 1994.

Parameters:  | double Alpha
| double Beta

State:  | unsigned long long int currentCount

GammaDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getDoubleSample:                         119.424 uS
|  -average # of generator calls per variate:   6.224 (D)
|  -distribution net time:                     56.194 uS
```

Normal and LogNormalDistribution Classes

```
//
// NormalDistribution -- a well-known continuous probability distribution
// LogNormalDistribution -- also well-known ...
//
@deftype NormalDist    <Normal, CREATABLE> @end
@deftype LogNormalDist <Normal, CREATABLE> @end
```

NormalDist Documentation

```
Name:  | NormalDistribution

Description:  | Normal (Gaussian) distribution returning double values

Algorithm:  | The Box-Muller method

Reference:  | W.H. Press et al, "Numerical Recipes in C" (2nd ed.).
| Cambridge University Press, 1992.

Parameters:  | double Mean
| double Variance

State:  | double stored_value
| BOOL stored
| unsigned long long int currentCount

NormalDist | On a 486/66, the following measures were obtained,
speed:  | using MT19937 as the underlying generator:

|  -getDoubleSample:                          24.729 uS
|  -average # of generator calls per variate:  1.273 (D)
|  -distribution net time:                    11.797 uS

Relative speed:
| Speed 0.150 (time 6.687) relative to MT19937 getUnsignedSample
--------------- . ----------------------------------------------------------
--
```

LogNormalDist Documentation

(RandomBitDist.html)

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

Last modified: 1998-01-10

We then configure the generator as having a number (A) of 'vir-
tual generators',
each of which can address a number (2^v) of sub-segments of length 2^w.
These parameters (A,v,w) are user selectable when the generator is created.
(As an example, for C4LCGXgen the default values are A=128, v=31, w=41.)
The advantage is that the subsegments act as statistically independent
streams of random numbers.

In addition to the -getUnsignedSample method, generators now also supply
floating point output in the range [0.0,1.0), in the form of these methods:

```
   -(float)       getFloatSample; // using 1 unsigned value
   -(double)      getThinDoubleSample; // using 1 unsigned value
   -(double)      getDoubleSample; // using 2 unsigned values
   -(long double) getLongDoubleSample; // using 2 unsigned values
```

The last method is not portable across architectures, since the length of a
long double varies between machines.

Generators may now be started with a single seed, *or* with a vector of seeds
whose length is generator dependent. (PMMLCG requires 1 integer for a
seed, while MT19937 needs 624 of them.)

Generators now remember what seed values they were started with. They also
count how many variates they have delivered (i.e., how many calls to
-getUnsignedSample they have serviced.)

There are a few arbitrary seed values, DEFAULTSEED, DEFAULT-
SEED1, DEFAULTSEED2,
DEFAULTSEED3, DEFAULTSEED4 defined. There is the value RANDOMSEED, which will
be different every time it is invoked. And there is value START-
SEED, which will
by default equal DEFAULTSEED, but will instead be equal to RANDOMSEED if you
start your program with the '-varySeed' command line parameter.

The generators have gained a new creation method, '+createWithDe-
faults: aZone',
which creates the generator and initializes it with STARTSEED. Split
generators get default values for A,v,w.

CHANGES since v. 0.6:

The generator classes have changed names to where they all end in '-gen'.
A simple search-and-replace in your code will get you up and running again.

(Or perhaps you'll want to try one of the new generators?)

This allocates the object and initializes it with your seed value. If the object actually requires a vector of seed values to fill the state, this method generates the rest of the values needed using an inline PMMLCG generator.

You can find out later what seed value was used to initialize the generator:

```
myUnsigned = [ myGenerator getInitialSeed ];
```

And you can find out what the largest valid seed value is by calling

```
myUnsigned = [ myGenerator getMaxSeedValue ];
```

(For the generators defined in v. 0.7, this value is 2^32-1 for all of them. The seed may not be 0.)

You may reset the generator's state at any time using this method:

```
[ myGenerator setStateFromSeed: mySeedValue ];
```

(c) using a vector of seed values:

Assume we have defined a fixed array at compile time:

```
unsigned int mySeedVector [vectorLength];
```

Then we can do this:

```
   myGenerator = [ PSWBgen create: [self getZone]
setStateFromSeeds: mySeedVector ];
```

You can find out how many seed values are required by asking

```
myUnsigned = [ myGenerator lengthOfSeedVector ];
```

(Obviously, you must first successfully have created the object to do this, for example using createWithDefaults!)

And we allocate the seed vector dynamically this way:

```
unsigned int *mySeedVector;
mySeedVector = [[self getZone] alloc: [ myGenerator lengthOfSeedVector]];
```

You can find out what vector of seed values was used to initial-
ize the object:

```
unsigned int *myVector;
```

(e) generator output:

You obtain successive pseudorandom numbers from a generator by calling

```
   myUnsignedValue = [ myGenerator getUnsignedSample ];
```

The largest value thus returned can be found by asking

```
   myUnsignedValue = [ myGenerator getUnsignedMax ];
```

(The smallest value returned is always 0.)

If you would rather have floating point output in the range [0.0,1.0),
you call one of these:

```
// Using 1 unsigned value to fill the mantissa:

   myFloatValue  = [ myGenerator getFloatSample ];
   myDoubleValue = [ myGenerator getThinDoubleSample ];

// Using 2 unsigned values to fill the mantissa:

   myDoubleValue     = [ myGenerator getDoubleSample ];
   myLongDoubleValue = [ myGenerator getLongDoubleSample ];
```

NOTE that the last method is not portable, because the size of a long double
varies and hence the precision varies between architectures.

Finally, you can obtain a count of how many variates have been generated:

```
   myLongLongInt = [ myGenerator getCurrentCount ];
```

(currentCount is an unsigned long long int, which counts up to 2^64.)

SPLIT generators:

A 'split' generator requires us to specify the configuration (A,v,w)
at create time:

```
   myGenerator = [ C4LCGXgen create: [self getZone]
setA: 64 setv: 20 setw: 76
setStateFromSeed: mySeedValue ];
```

```
   [ myGenerator restartGenerator: myVgen ];
   [ myGenerator advanceGenerator: myVgen ];
   [ myGenerator jumpGenerator:    myVgen    toSegment: myLongLongIntValue ];
```

InternalState methods common to simple and split generators:

```
   // Print (most of) the object's state data to a stream:
   [ myNormalDist describe: myStream ];
```

The stream myStream may be created thus:

```
id myStream = [ OutStream create: [self getZone] setFileStream: stdout ]; or
id myStream = [ OutStream create: [self getZone] setFileStream: stderr ];

   // Get the (class) name of the object:
   myString = [ myNormalDist getName ];

   // Get the object's 'magic number', used by putStateInto / setStateFrom:
   myUnsigned = [ myNormalDist getMagic ];
```

SAVING AND RESETTING STATE:

You may save, and later restore, the internal state of a generator
using these methods:

```
   // Get the size of the memory buffer needed by put-
StateInto / setStateFrom:
   myUnsigned = [ myGenerator getStateSize ];

   // Extract the generator's state data into your memory buffer:
   [ myGenerator putStateInto: myBuffer ];

   // Set the generator's state from data in a memory buffer:
   [ myGenerator setStateFrom: myBuffer ];
```

To illustrate, assume the following data definitions:

```
   FILE *myFile;
   const char *myFileName = "MyGenFile.bin"; // or whatever
   int stateSizeG;
   id stateBufG;
   int status;
```

```
   // Close the file:
   status = fclose(myFile);
   if (status) { }; // error on close

   // Ask the generator set its state from the buffer data:
   [ myGenerator setStateFrom: (void *) stateBufG ];

   // Free the memory allocated to the buffer:
   [[self getZone] free: stateBufG];
```

TESTING GENERATORS:

Since v. 0.6 we have done some rudimentary statistical testing of the implemented generators, using Marsaglia's Diehard tests and the ENT tests.

The results of these tests are summarized in document /random/docs/doc.quality.generators, where test results as well as period length, state size and execution times are listed. You can use these data to select a generator that suits your simulation.

Some brief comments:

a) the tests show that old generators SCG and LCG are of poor quality and should be avoided.

b) the lagged-Fibonacci based generators (ACG, SWB, PSWB) all fail Diehard's 'Birthday spacings test', for reasons having to do with their lattice structure.
These generators are only conditionally recommended.

c) The rest of the 32-bit generators (i.e. generators that fill all 32 bits of an unsigned int) pass all tests, and are recommended at this time.

(Note that while a test may show that a generator is bad, passing a number of tests does not prove that a generator is good!)

d) The 31-bit generators all fail the same set of tests. Some of these cannot be passed by a generator whose output has a 'stuck' bit.
Until I clear up with Prof. Marsaglia how to interpret these results, I believe all the 31-bit generators are in the 'recommended' category.

However, a cautionary note: while the PMMLCG generators pass the tests, they have a very short period (less than 2^31) and should only be used for 'toy' simulations. You don't want your generator(s) to 'go around' and start repeating themselves !

UniformIntegerDist and UniformUnsignedDist now allow you to set parameter minValue equal to maxValue. In this case that value is returned every time.

UniformDoubleDist also allows this, even if the set [x,x) is mathematically suspect ...

NormalDist and LogNormalDist now allow you to specify zero Variance, in which case the values returned are the Mean and exp(Mean) respectively.

CHANGES since v. 0.6:

The distribution classes have changed names to where they all end in 'Dist'. A simple search-and-replace in your code will get you back up and running.

The strong distinction between 'frozen' and 'un-frozen' distribution objects in v. 0.6 has been softened considerably. You may now set and reset the default parameters as often as you wish, and you may make calls for variates with given parameters even if different default parameters have been set.

The generation of uniform(0,1) floating point values has been moved from the distribution objects into the generator objects. Thus, if all you need is a uniform(0,1) double, you have no need of a distribution but can get what you desire from a generator.

Note that the generators fill the mantissa of a double from two 32-bit unsigned values in a different manner from v. 0.6 distributions, so output will be a bit different in the new version.

A bug in LogNormalDist has been fixed.

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.

But note: these methods have also changed somewhat, as has the size of the data being saved. As a result, v. 0.7 distributions will refuse to 'setStateFrom' data saved by v. 0.6 objects.

There should be fewer changes like this in the next release.

USAGE notes:

You can create the generator at the same time as the distribution:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: [TT800gen create: [self getZone]
setStateFromSeed: 34453]         ];
```

(c) Without default parameters, using a split generator:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: mySplitGenerator
setVirtualGenerator: 7 ];
```

or perhaps

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: [C4LCGXgen createWithDefaults: [self getZone]]
setVirtualGenerator: 99 ];
```

A split generator can be thought of as comprising a set of virtual generators (streams of random numbers), and a distribution object must be 'connected' to one of these streams. You cannot re-assign the generator or the virtual generator after a distribution object has been created.

In all these cases (a) - (c), when we want to obtain a random variate from this distribution object we need to specify the parameters:

```
   myDouble = [ myNormalDist getSampleWithMean: 3.3 withVariance: 1.7];
```

You can use different parameters for every call. (And you can use this call even if default parameters have been set.)

(d) With default parameters, using a simple generator:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: mySimpleGenerator
setMean: 7.6 setVariance: 1.2 ];
```

(e) With default parameters, using a split generator:

```
   myNormalDist = [ NormalDist create: [self getZone]
setGenerator: mySplitGenerator
setVirtualGenerator: 33
setMean: 3.2 setVariance: 2.1 ];
```

In these cases, we do not need to specify parameters to get a random number:

```
   // Print (most of) the object's state data to a stream:
   [ myNormalDist describe: myStream ];
```

The stream myStream may be created thus:
id myStream = [OutStream create: [self getZone] setFileStream: stdout]; or
id myStream = [OutStream create: [self getZone] setFileStream: stderr];

```
   // Get the (class) name of the object:
   myString = [ myNormalDist getName ];

   // Get the object's 'magic number', used by putStateInto / setStateFrom:
   myUnsigned = [ myNormalDist getMagic ];
```

You may save, and later restore, the internal state of a distribution object using these methods:

```
   // Get the size of the memory buffer needed by put-
StateInto / setStateFrom:
   myUnsigned = [ myNormalDist getStateSize ];

   // Extract the distribution's state data into your memory buffer:
   [ myNormalDist putStateInto: myBuffer ];

   // Set the distribution's state from data in a memory buffer:
   [ myNormalDist setStateFrom: myBuffer ];
```

To illustrate, assume the following data definitions:

```
   FILE *myFile;
   const char *myFileName = "MyDistFile.bin"; // or whatever
   int stateSizeD;
   id stateBufD;
   int status;
```

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:
   stateSizeD = [ myNormalDist getStateSize ];

   // Allocate memory for the buffer:
   stateBufD  = [[self getZone] alloc: stateSizeD];

   // Ask the distribution object to put state data into the buffer:
```

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.

(Interface.html)

(../random.html)

(AdvancedUsage.html)

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

Last modified: 1998-01-10

Random Library: Interface Design Notes

Never has a library interface been debated as this one was (and still is). This document will attempt to provide some of the rational for the decisions made by echoing some of the controversial issues covered in the debates on the interface. (These discussions overlapped into implementation. But, the point, here, is to give people an idea of the complexity of the issues involved in useful pseudo-random number use in simulation.)

Issues:

- Different classes for generators or variants? Ref: http://www.sela.co.il:8080/swtrain/new/shai/tgp.ps
- Include non-recommended generators?
- Seed vs. State for generators?
- What is "state" and how do we handle it?
- Larger issue of debugging support via an object reporting its own configuration via a special method and how this relates to object state reporting and object saving.
- Separate classes for separate distribution types (e.g. double, integer, unsigned :: continuous/discrete)?
- Problems with the transformation from generator output to distribution output.
- What precision should be supported?
- Should the generator underlying a distribution be visible through the distribution?
- Efficiency!
- What distributions to provide?
- What generators to provide?
- Open vs. Closed intervals?
- Programmed default generators, seeds, and states?
- Programming by committee? (a.k.a. Software process!)
- Shorthand create methods?
- Potential phase optimizations?
- Random number streams, syncing limit cycles, and proper simulation vs. code debugging. (This is a HOT ONE.)
- Library interchangeability.

These are just the questions and issues. Some addition should follow (when there's time) addressing each issue and giving rationale for what we chose. But, that's a big job.

Random Library: Implementation Notes

The generators supplied with this release have been subjected to statistical testing using George Marsaglia's Diehard tests as well as John Walker's entropy tests (ENT). The results of these tests are summarized in the document `$(SWARMHOME)/src/random/docs/doc.quality.generators`. Other properties of the generators are summarized in `$(SWARMHOME)/src/random/docs/generators.table`. Some notes on how to choose a generator for your simulation are found in `$(SWARMHOME)/src/random/docs/CHOOSING.A.GENERATOR`.

The ENT test is included in the tarball of test programs found at the Swarm web site. The Diehard tests are copyright and hence are not, but they can be downloaded from the web at http://www.hku.hk/internet/randomCD.html (http://www.hku.hk/internet/randomCD.html)
.

The distribution objects have not been statistically tested.

(./InterfaceDesign.html)

(../random.html)

(../random.html)

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

Last modified: 1998-01-10

(simtools/Interface.html)

- Usage Guide
(simtools/UsageGuide.html)
- Advanced Usage Notes
- Interface Design Notes
- Implementation Notes

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

Last modified: 1998-01-08

Simtools Library: Interface Reference

The *simtools* 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 <simtools.h>:

```
#import <objectbase.h>
#import <random.h> // for global RNGs...
#import <tkobjc.h> // some methods return graphics stuff...
```

The *objectbase* library interface is included to provide the basic object support. The *random* library is included to provide support for default generators and distributions, the use of which is deprecated, but are included for backwards compatibility.

Special global functions - global.h

Users need to include simtools.h in their code in order to call `initSwarm()` but also in order to get access to a set of important pre-initialized objects which are generated in every simulation (e.g. `uniformIntRand`, `probeDisplayManager`, and so on).

void initSwarm(int argc, char ** argv)
: Initializes the Swarm libraries. This should be the call in any Swarm code you write. `argc` and `argv` are the input parameters to `main()`.

ProbeDisplayManager * probeDisplayManager
: A global instance of the ProbeDisplayManager object. You can use this object to create new probe displays, and should schedule an update message to it in your display schedule so as to update the probe displays it has created. See the documentation on probes
(../probing.html)
for more details.

int swarmGUIMode
: A flag that indicates whether graphics are enabled. Set automatically based on whether the switch `-batchmode` was present as one of the command line arguments starting the application (e.g. `heatbugs -batchmode`). If it is false, then

(NSelect.html)

- UName
(UName.html)
- ActiveGraph
(ActiveGraph.html)
- ActiveOutFile
(ActiveOutFile.html)

(../simtools.html)

(../simtools.html)

(UsageGuide.html)

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

Last modified: 1998-01-10

Name

ControlPanel - class to control the top level SwarmProcess

Synopsis

```
// ControlPanel --
// an class used to control the top-level SwarmProcess. It
// accepts manipulations both by the Swarm it's controlling and
// the ActionCache.
//

@protocol ControlPanel <SwarmObject>
CREATING
- createEnd;
USING
- getState;
- setState: s;
- waitForControlEvent;
- startInActivity: activityID;
- setStateRunning;
- setStateStopped;
- setStateStepping;
- setStateQuit;
- setStateNextTime;

// Deprecated methods
- doTkEvents;
- getPanel;
@end

// State Symbols for the ControlPanel.
extern id ControlStateRunning, ControlStateStopped;
extern id ControlStateStepping, ControlStateNextTime, ControlStateQuit;
```

Description

ControlPanel keeps track of the users requests to run, stop, quit, or time step the simulation. It cooperates with the GUISwarm to control the execution of activities in Swarm.

- getState

Name

ActionCache - a class to manage threads and Swarms

Synopsis

```
//
// ActionCache -
//
// A class that provides a smart bag into which actions can be
// thrown by other threads and Swarms intended for insertion on
// it's Swarm's schedule.
//

@protocol ActionCache <SwarmObject>
CREATING
- setControlPanel: cp;
- setScheduleContext: context;
- setWindowGeometryRecordName: (const char *)name;
- createEnd;

USING
- insertAction: actionHolder;
- deliverActions;
- sendActionOfType: (id <Symbol>)type toExecute: (const char *)cmd;
- sendStartAction;
- sendStopAction;
- sendStepAction;
- sendNextAction;
- sendQuitAction;
- verifyActions;

- createProcCtrl;
- getPanel;
- doTkEvents; //should change to pollGUI or something
@end

// Type Symbols for ActionCache
extern id <Symbol> Control, Probing, Spatial;
// Error Symbols for ActionCache
extern id <Symbol> InvalidActionType, ActionTypeNotImplemented;
```

Description

Name

ProbeDisplay - a class to display ProbeMaps

Synopsis

```
//
// ProbeDisplay
//
// A class which generates a GUI to a ProbeMap of probes applied to a
// given target object.
//

@protocol ProbeDisplay <SwarmObject>
CREATING
- setProbedObject: anObject;
- setProbeMap: (ProbeMap *)probeMap;
- setWindowGeometryRecordName: (const char *)name;
USING
- getProbedObject;
- getProbeMap;
- update; // implemented in SimpleProbeDisplay...
@end
```

Description

Create Phase Protocol

- setProbedObject: anObject
  This method *must* be called.

- setProbeMap: (ProbeMap *)probeMap
  This is an optional create phase method - if no probeMap is specified the ProbeDisplay will ask the probedObject for a ProbeMap using the getProbeMap method described below... The default behaviour of this method will be to return the probeLibrary's copy of the probeMap for the class of the target object.

- setWindowGeometryRecordName: (const char *)name

Name

CompleteProbeDisplay

Synopsis

```
//
// CompleteProbeDisplay --
// A class which generates a GUI to a complete ProbeMap of probes applied
// to a given target object (by complete we mean that all the probes for
// the target object's class and its superclasses are included).
//

@protocol CompleteProbeDisplay <SwarmObject>
CREATING
- setProbedObject: anObject;
- setWindowGeometryRecordName: (const char *)name;
USING
- getProbedObject;
- update;
- getMarkedForDropFlag;
@end
```

Description

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

Last modified: 1998-01-08

- createProbeDisplayFor: anObject
- createDefaultProbeDisplayFor: anObject
- createCompleteProbeDisplayFor: anObject
- createProbeDisplayFor: anObject setWindowGeometryRecordName: (const char *)name
- createCompleteProbeDisplayFor: anObject setWindowGeometryRecordName: (const char *)name;
- dropProbeDisplaysFor: anObject
  These methods allow a one-step creation/removal of ProbeDisplays onto a given object (dropProbeDisplaysFor: will actually *drop* the appropriate ProbeDisplays). The setWindowGeometryRecordName: creation variety can be used to set names for archiving geometry information.

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

Last modified: 1997-12-19

We use the term word instead of string because the returned value will not contain spaces, tabs and newlines -> thus getWord...

- (int)getLine: (char *)aLine
  This routine loads the argument string with the characters up to, but not including a newline character.
- (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;
  These routines take a pointer to the required type, and read into it an instance of that type from the open file. In case of failure these message returns 0.
- (int)unGetChar: (char) aChar
- (int)skipLine
  These routines are used by ObjectLoader, but are documented in case the user wishes to define his/her own file format which may require backtracking or skipping lines.
- (void)drop
  This method must be called when the user wants to close the file. Only by dropping the InFile object will the file truly be closed!

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

Last modified: 1998-01-08

- putUnsigned: (unsigned) anUnsigned;
- putLong: (long) aLong;
- putUnsignedLong: (unsigned long)anUnsLong;
- putDouble: (double)aDouble
- putFloat: (float)aFloat
- putChar: (char)aChar
- putTab: (char)aChar
- putNewLine

  These routines take an instance of the appropriate type and write it into the open file. The return value is simply self since no error detection is performed.

- (void)drop

  This method must be called when the user wants to close the file. Only by dropping the OutFile object will the file truly be closed!

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

Last modified: 1998-01-08

- putLong: (long)aLong;
- putUnsignedLong: (unsigned long)anUnsLong;
- putDouble: (double)aDouble
- putFloat: (float)aFloat
- putChar: (char)aChar
- putTab: (char)aChar
- putNewLine
  These routines take an instance of the appropriate type and write it into the open file. The return value is simply self since no error detection is performed.
- (void)drop
  This method *must* be called when the user wants to close the file. Only by dropping the AppendFile object will the file truly be closed!

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

Last modified: 1997-12-19

@end

Note: A single file can be used to initialize multiple objects by simply inserting more than one @begin @end
block.

+ load: anObject fromFileNamed: (const char *)aFileName This convenience class method will load anObject from the file named aFileName (the class will open the file, initialize the object with its contents and then close the file).
+ load: anObject from: aFileObject This convenience class method will load anObject from the previously opened aFileObject without returning an actual instance of the ObjectLoader class (also, the FileObject remains open after the method has been called).
- setFileObject: aFileObject If the user bothers to generates an actual instance of the ObjectLoader class, s/he can set the source fileObject which the instance should use by sending it this message.
- loadObject: anObject This is the message the user must send an instance of the ObjectLoader class in order to initialize the target object from the requested file.
- updateCache This method should be called if an ObjectLoader instance is going to initialize a large number of objects from the same class.
- setTemplateProbeMap: aProbeMap This method is used to specify which variables of the target object(s) should be loaded by the ObjectLoader instance to which this message was sent.

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

Last modified: 1998-01-08

As in the case of ObjectLoader we provide a set of convience class methods which will save the entire target object (or the subset of variables specified in a template) without actually returning an instance of ObjectSaver to the user.

- setFileObject: aFileObject
  If the user bothers to generates an actual instance of the ObjectSaver class, s/he can set the target fileObject which the instance should use by sending it this message.
- saveObject: anObject
  This is the message the user must send an instance of the ObjectSaver class in order to save the state of the target object into the requested file.
- setTemplateProbeMap: aProbeMap
  This method is used to specify which variables of the source object(s) should be saved by the ObjectSaver instance to which this message was sent.

Saving floats and doubles: The desired number of significant figures for all floats and doubles can be set in the global instance of the ProbeLibrary. However, if notset, it will default to a value of six (6) significant figures, which is identical to previous versions that did not have this feature. See the documentation on probes (../probing.html)
for how to set this precision.

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

Last modified: 1998-01-08

+(void) sortNumbersIn: aCollection using: (int(*)(const void*,const void*))comp_fun
    As above, except that a specific function is given.

+(void) reverseOrderOf: aCollection
    Reverses the current order of a collection. To make a "reversed" (i.e. decending) sort, simply call one of the appropriate "sort..." methods on a collection, then call this function on the same collection.

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

Last modified: 1998-01-08

Name

UName - class to generate unique names for objects/agents

Synopsis

```
//
// UName --
// A class used to generate unique names (e.g. "critter1", "critter2" etc).
//

@protocol UName <SwarmObject>
CREATING
+ create: aZone setBaseName: (const char *)aString;
+ create: aZone setBaseNameObject: aStringObject;

- setBaseName: (const char *)aString;
- setBaseNameObject: aStringObject;
USING
- (const char *)getNewName;
- getNewNameObject;
@end
```

Description

This class is used to generate unique names (agent0, agent1, agent2...) for objects in a simulation. The user will typically create an instance of the UName class initialized with a baseName presented either as a (const char*) or an object of class String

(../collections/String.html)
. The user can then request new names, again either as (const char *)'s or as instances of the String class. The user can also reset the counter used to generate the names in case s/he wants to restart naming objects with the same baseName.

> Note: Both in the case of initialization by (const char *) and initialization by an instance of the String class, the original is copied *not stored internally* so it is up to the user to free the original (const char *) or String instance if/when necessary!

+ create: aZone setBaseName: (const char *)aString

Name

ActiveGraph - provides continous data feed between Swarm and the GUI

Synopsis

```
//
// ActiveGraph --
// This class provides the continuous data feed between Swarm and the GUI.
//

@protocol ActiveGraph <MessageProbe>
USING
- setElement: ge;
- setDataFeed: d;
- step;
@end
```

Description

An active graph object is the glue between a MessageProbe (for reading data) and a BLTGraph GraphElement. ActiveGraphs are created and told where to get data from and send it to, and then are scheduled to actually do graphic functions. This class is used by EZGraph, and we expect to see less direct usage of it by end-users as more analysis tools (such as EZGraph) internalize its functionality.

- setElement: get
  Sets the graph element used to draw on
- setDataFeed: d
  Sets the object that will be probed for data.
- setProbedSelector: (SEL)s
  Sets the message sent to the object for graphing (inherited from MessageProbe).
- step
  Fires the probe, reads the value from the object, and draws it on the graph element. The X value is implicitly the current simulation time. Y is the value read.

Name

ActiveGraph - provides continous data feed between Swarm and a file

Synopsis

```
//
// ActiveOutFile --
// Provides the continuous data feed between Swarm and a File.
//

@protocol ActiveOutFile <MessageProbe>
USING
- setFileObject: aFileObj;
- setDataFeed: d;
- step;
@end
```

Description

This is the file I/O equivalent of ActiveGraph: it takes an OutFile object, a target (datafeed) object, and a selector, which it uses to extract data from the object and send it to the file. This class is used by EZGraph, and we expect to see less direct usage of it by end-users as more analysis tools (such as EZGraph) internalize its functionality.

```
 - setFileObject:  anOutFile
      Sets the file object to which the data will be sent.
 - setDataFeed:  d
      Sets the object that will be probed for data.
 - setProbedSelector:  (SEL) s
      Sets the message sent to the target object (inherited from MessageProbe).
 - step
      Fires the probe, reads the value from the object, and sends it to the file.
      The X value is implicitly the current simulation time.  Y is the value read.
```

Marcus G. Daniels <mgd@santafe.edu>

Simtools Library: Usage Guide

Overview

The *simtools* library contains a number of semi-related tools that are useful for writing simulations.

(Interface.html)

(../simtools.html)

(../simtools.html)

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

Last modified: 1998-01-10

to draw them on the screen. All widgets have three essential things: a widget name used when running Tcl code, an Objective C name when sending messages from Tcl to those objects, and a parent.

+ createParent: (Widget *)p and - setParent: (Widget *)p
: When a widget is created it needs to be given a parent. The parent widget will be the widget's containing window. If no parent is given (ie, a parent of nil), then a toplevel Frame will be allocated automatically

- setWindowTitle: (const char *)s
: Sets the window manager title of the top level window

- pack, - packWith: (const char *)c
: Roughly, packing a widget makes it draw on the screen. The Tk packer allows complicated options to control widget layout: you can pass options via -packWith:. See documentation on Tk to learn more about packing details.

- getWidth, - setWidth: (int)n, - getHeight, - setHeight: (int)n, - getPositionX, - getPositionY, - setPositionX: (int)n Y: (int)n, etc.
: These methods allow you to control the position and size of the widget.

- (const char *)getWidgetName, -(const char *)getObjcName
: Get various names for the widgets. The WidgetName is a Tcl name for the window: the ObjcName is the name of the object itself. These are used if you choose to send strings off to Tcl directly for evaluation.

Simple Widgets

There are a variety of simple widgets which do straightforward things.

Frame

Frames are boxes other widgets fit in. They correspond to the Tk "frame" and "toplevel" widgets. Frames can be new windows, or subwindows in an existing window. You only need to create frames yourself if building complicated composite widgets: by default, a frame will be built automatically for widgets without parents.

Label

Labels are text strings used for displaying short information.

- setText: (const char *)t
: set the label's text string and resize it to fit.

Compound Widgets

Some TkObjc widgets consist of several other widgets bound together into one object.

ButtonPanel

Several buttons bound together in one frame.

```
- addButtonName:  (const char *)n Command:  (const char *)c
     add a new button to the panel with the given name and command.
```

Data Display Widgets

TkObjc has two basic classes for displaying numerical data.

BLTGraph

A time series graph tool, based on BLT's graph widget. BLTGraph currently implements just
a basic graph with multiple datasets, but should eventually support scaling and scrolling.
For each BLTGraph you create one or many GraphElements, one per dataset to plot. GraphElements
can be configured for appearance, and data can be added to the element to draw.

```
- title:  (const char *)t, - axisLabelsX: (const char *)xl Y: (const char *)yl
     set labels and title on the graph.
- setScaleModeX: (int)xd Y: (int)ys
     takes two boolean values - whether to autoscale every timestep or instead
     to jump scale.
- setRangesXMin:  (double)xmin Max:  (double)xmax YMin:  (double)ymin Max:  (double)ymax
     sets the ranges for the graph, turns off autoscaling entirely
- (GraphElement *)createElement
     builds a new GraphElement for you to plot data with
- destroyElement:  (GraphElement *)g
     remove the given graph element, destroy the plot.
```

GraphElements are where the real work are, where you give it data to plot. One BLTGraph can
have multiple GraphElements.

```
- setLabel:  (const char *)label
     set the label for this element
```

The display widgets used here do not have Tk counterparts: they are raw Xlib code written for flexibility and (relative) speed.

XColormap

Mechanism used to map numbers in the range [0, 255] to colour names. Create an XColormap, allocate colours in it, and pass it to a Raster widget for drawing.

- (BOOL)setColor: (Color)c ToName: (const char *)colorName
  set the specified color to the given colorname.
- -(BOOL) setColor: (Color) c ToGrey: (double) g
  set the specified color to the grey value in [0, 1.0].
- -(BOOL) setColor: (Color) c ToRed: (double) r Green: (double) g Blue: (double) b
  set the specified color to the red, green, blue vector in the range [0, 1.0].

Raster

2 dimensional, colour pixel images. Raster is based on a Tk frame widget with our own code for fast display of images. You can draw coloured dots on a Raster, or generic <XDrawer>s (not documented). Raster widgets are double buffered - the pixels you draw are not actually put on the screen until - drawSelf is called. In addition, Rasters handle mouse clicks.

- erase
  erase the window to black
- drawSelf
  render the Raster on the screen (complete the double buffer)
- setColorMap: (XColorMap *)colormap
  sets the colourmap used to translate logical colors to screen colors.
- -(XColormap *) getColormap;
  returns the id of the colormap
- drawPointX: (int)x Y: (int)y Color: (Color)c;
  draw a point at the given coordinates with the given color.
- fillRectangleX0: (int)x0 Y0: (int)y0 X1: (int)x1 Y1: (int)y1 Color: (Color)c
  fill a whole rectangle with color
- setButton: (int)n Client: c Message: (SEL)s
  when button n (ButtonLeft, ButtonMiddle, or ButtonRight) is pressed, send the given message to the specified client object.
- (Display *)getDisplay
  returns the id of the display
- handleButton: (int)n X: (long)x Y: (long)y
  fields the button clicks

CanvasItem

This is the superclass of any item that might be drawn on a Canvas, e.g. the geometric shapes, textItems, etc. This is an abstract class.

- setCanvas: canvas
  designates the id of the Canvas in which this item resides
- setTargetId: target
  designates the object to which this item refers
- setClickSel: (SEL)sel
  sets the message that will be effected upon a click on this item
- setMoveSel: (SEL)sel
  sets the message that will effect the motion of the item on the canvas
- setPostMoveSel: (SEL)sel
  sets the message that will dictate what happens after the item is moved
- createItem
  passthrough method to be implemented by subclasses
- createBindings
  establishes the bindings for the buttons: button 3 registers a "click", button 1 up and down sets the focus, and button 1 held down initiates motion.
- createEnd
  calls createItem and createBindings
- clicked
  sends the clickselector message to the referent
- initiateMoveX: (long)delta_x Y: (long)delta_y
  prepares for movement of the item within the canvas

Circle : CanvasItem

- setX: (int)x Y: (int)y
  set the x,y coordinates for the center of the circle
- setRadius: (int)r
  set the radius of the circle

Line : CanvasItem

- setTX: (int)tx TY: (int)ty LX: (int)lx LY: (int)ly
  set the endpoints of the line

Rectangle : CanvasItem

- setTX: (int)tx TY: (int)ty LX: (int)lx LY: (int)ly

```
  - update
        redraw the link (esp due to motion of nodes)
```

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

Last modified: 1998-01-08

- update

    Update the entropy - send the message throughout the collection and update the entropy. Schedule this message when you need to collect the data.

EZGraph

This class allows the user to easily create graphs of various quantities in the model s/he is investigating. The user first creates the EZGraph, and then creates "Sequences"; (lines) which will appear in the graph. The sequences are generated based on data provided by a a single object or a collection of target objects, in reponse to a specified selector. So, for example, the following code generates a graph of statistics about the unhappiness of a list of heatbugs:

```
// 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: "average unhappiness"
                         withFeedFrom: [heatbugModelSwarm getHeatbugList]
                          andSelector: M(getUnhappiness)] ;

  [unhappyGraph createMinSequence: "minimum unhappiness"
                     withFeedFrom: [heatbugModelSwarm getHeatbugList]
                      andSelector: M(getUnhappiness)] ;

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

One of the features of the EZGraph is that it will automatically generate average, total, min, max and count sequences without the user having to mess with Averagers amd other low-level classes.

- setGraphics: (int)state

    If the user does not require a graphical display of the graph, 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.

- setFileOutput: (int)state

    If the user would like the data for the sequences to be sent to a file, s/he should call this method with state set to 1. The default state is 0 meaning that by default, no file I/O is carried out by the EZGraph class.

- setTitle: (const char *)aTitle

- setFileOutput: (int)state
  If the user would like the data for the histogram to be sent to a file, s/he should call this method with state set to 1. The default state is 0 meaning that by default, no file I/O is carried out by the EZBin class.
- setTitle: (const char *)aTitle
  The title is used both in the graphical version of EZBin as well as in the file I/O where it serves as the name of the output file (when graphics are enabled it appears at the top of the histogram window).
- setAxisLabelsX: (const char *)xl Y: (const char *)yl
  Sets the labels on the histogram. Only relevent for the graphical version.
- setBinNum: (int)theBinNum
  Sets the number of bins the histogram will have.
- setLowerBound: (double)theMin
  Sets the lower bound on the histogram range.
- setUpperBound: (double)theMax
  Sets the upper bound on the histogram range.
- (int)getBinNum
- (double)getUpperBound
- (double)getLowerBound
  These methods retrieve the appropriate settings.
- setCollection: aCollection
  Set the collection of target objects which will be requested to generate the dataset for the histogram.
- setProbedSelector: (SEL)aSel
  Set the selector which will be applied to the objects in the specified collection in order to generate the dataset (inherited from *MessageProbe*).
- reset
  This method resets the histogram.
- update
  This method increases the amount of data included in the final dataset. In other words, it is possible to poll the same collection of objects repetitively before generating output (either graphically or on file).
- output
  This methods causes the graphical display to be updated (such that it takes into account the information extracted by the latest calls to update). When file I/O is enabled the number of entries per bin are sent to the output file.
- (int *)getDistribution
  Returns an array of integers (containing the number of entries which landed in each bin of the histogram).
- (int)getCount
  Get the number of entries which landed within the bounds of the histogram.

Probing and Displaying the Contents of a SwarmObject

Contents:

Overview

The Framework

Probes, ProbeMaps and ProbeDisplays allow the user to dynamically interact with the objects in their simulation. By *interaction* we mean reading/setting instance variables as well as dynamically generating method calls. The main point being that these interactions are not hardwired into the program code, but occur due to user-generated requests, mainly through the provided GUI.

The key to this capability is the Probe: in general a probe takes an object and either extracts the value of a specific variable, or calls a specific method. For this purpose we provide two subclasses: VarProbe and MessageProbe.

There are two main uses for probes: they can be fed into data-collection objects and serve as interfaces to the objects about which data is being collected (thus keeping the data-collection objects as general as possible) - the Averager class, for example, directly subclasses MessageProbe. Or, they can be used in order to generate a GUI to the individual objects in the simulation (the more common usage).

In order to generate a graphical version of a given probe, the programmer must place it inside a ProbeDisplay, which will automagically generate a window with the appropriate interface. Since, more often than not, the programmer will want to generate windows with more than one variable, we have designed the ProbeDisplay to deal with ProbeMaps rather than individual probes. Thus, by generating a ProbeMap containing exactly the right probes, a user is able to customise the window generated by the ProbeDisplay.

In order to facilitate the creation of all these different objects, the Swarm kernel provides some alternative methods for Probe/Map/Display generation:

- Direct Generation - which leaves the user in charge of all the details.
- Object Generation (Probes/ProbeMaps) - where we use method calls (defined in SwarmObject) on a given target object to generate the Probe/ProbeMap for that object. This allows the object to be aware that it is being probed and to control directly the contents of the Probe/ProbeMap being requested...
- Library Generation (Probes/ProbeMaps) - where the programmer "checks out" a unique, shared copy of a Probe/ProbeMap from the probeLibrary object (of class ProbeLibrary) provided by the kernel. By shared we mean that a similar request made at a different point in the code, will return a reference to the very same probe instance.
  Note that, by default, Object Generation is equivalent to Library Generation since the default behaviour of SwarmObjects, when requested to create a ProbeMap, is to return a reference to the probeLibrary's unique copy!!!
- ProbeDisplayManager Generation (ProbeDisplays) - where the programmer generates a ProbeDisplay directly, by requesting it from the probeDisplayManager object (of class ProbeDisplay) provided by the kernel (in graphics mode). The probeDisplayManager will create the ProbeDisplay based on a ProbeMap given to it by the probed object. Note: since the probeDisplayManager is only created in graphics mode, it is documented in the simtools area of the documentation. Here we emphasise simply that in order to generate a ProbeDisplay onto an object the programmer need only write the following line of code:

```
id <ProbeMap> probeMap;

[...]

probeMap = [EmptyProbeMap createBegin: aZone];
[probeMap setProbedClass: [self class]];
probeMap = [probeMap createEnd];
// set the display defaults
[probeLibrary setDisplayPrecision: 3];
// typically saved precision would be higher than displayed precision
// for statistical and data analysis purposes
[probeLibrary setSavedPrecision: 10];

// Add in a bunch of variables, one per simulation parameters
[probeMap addProbe: [probeLibrary getProbeForVariable: "displayFrequency"
  inClass: [self class]]];
[...]

// Now install our custom probeMap into the probeLibrary.
[probeLibrary setProbeMap: probeMap For: [self class]];
return obj;
}
```

Setting precision for individual probes

The formatting for an individual probe can be set directly, using a sprintf-style formatting string. Typically, customProbeMaps are created in the + createBegin factory method for a Swarm or a SwarmObject. To set the formatting for a floating point probe, the method from VarProbe

(objectbase/VarProbe.html)
is used:

```
-setFormatFloat:  (const char *)floatFormat
      is applied to the instance of the VarProbe "checked-out" of the global probe
      library instance.  The sprintf-formatting string can "over-ride" the "%g"
      format set by the global precision (as above) (Typically "%g" chooses between
      the "%f" and "%e", depending on the size of the exponent - which is implementation-dependent
      - this method allows you to explicitly set the type of display).
```

In the following example, it is desired that the number of significant figures for the floating point variable randomMoveProbability is three (3). Currently (Swarm 1.0.5) this is only works for VarProbes and not MessageProbes, as yet.

Example Here's how it's done in HeatbugModelSwarm.m:

```
+ createBegin: aZone
{
```

```
- (const char*)getInstanceName
```

If you want to put a specific title on the ProbeDisplay window associated with your object, simply override this method. By default it returns the class name of the object. This is why the default ProbeDisplay window title is always the class name of the probed object.

Note: SwarmObjects do not actually have an instance variable called "instanceName" - so if you want to store a name for your object, you will need to provide your own storage at the subclass level and rewrite getInstanceName to refer to this variable.

```
- getProbeForVariable: (const char *)aVariable
```

```
- getProbeForMessage: (const char *)aMessage
```

```
- getProbeMap
```

```
- getCompleteProbeMap
```

Probe support resident in the *simtools* library:

ProbeDisplay

The ProbeDisplay class is meant to serve a dual role, represented by the two example windows on this page:

slot and release the mouse button).

- Also, if an instance variable/argument slot is defined to hold an object, then that object can be inspected by clicking the entry for that variable/argument slot with the third mouse button(a ProbeDisplay for that object will be generated).

Available only on the standard ProbeDisplay:

- Note that the sunken label at the top of the ProbeDisplay is also active. By clicking on it with the first mouse button you get a drag&drop'able representation of *self*. By clicking on it with the third mouse button you get a CompleteProbeDisplay to *self*.

Available only on the ProbeDisplay:

- The green "superclass" button can be used to display the succesive superclasses of the object being probed.
- The red "hide" button can be used to hide classes which are irrelevant thus reducing clutter.
- The hide button on the lowest class in the hierarchy has a special meaning since clicking on it dismisses the entire ProbeDisplay.

Interface definition

(simtools/ProbeDisplay.html)
from simtools
(simtools.html)
library.

ProbeDisplayManager

The normal Swarm simulation will probably only ever contain one instance of this class, namely the probeDisplayManager. This object is used for automatic generation of ProbeDisplays: given an object it will attempt to get a ProbeMap from the object. If the object does not respond to the method - getProbeMap (i.e. it does not inherit from SwarmObject), the probeDisplayManager will query the probeLibrary for a ProbeMap. If there is not a *ProbeMap* installed in the probeLibrary for the class of that object, it will creat a *DefaultProbeMap*. It will then create a ProbeDisplay to the target object, and from then on, until the Probe Display is removed it will update the ProbeDisplay when receiving an update message.

Interface definition
(simtools/ProbeDisplayManager.html)
from simtools
(simtools.html)