Brief Overview of Swarm

SWARM

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

Swarm Development Group

Published 10 February 1999
Release 1.4.1

Abstract 1: 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

Revision History

1999-01-25 overmeta.sgml *alex*
Move bibliodata entity outside of BOOKBIBLIO - so legalnotice links work.
1999-01-12 overmeta.sgml *alex*
Include the newly-defined {doc,swarm}-legalnotice entities from global.ent. Move CORPAUTHOR inside BOOKBIBLIO.
1998-12-26 over01.sgml, over02.sgml, over03.sgml, over04.sgml, over05.sgml *alex*
Standardize use of IDs according to conventions. Add IDs for all SECT1 elements.
1998-10-13 overbook.tex.in *mgd*
Use top_dossrcdir instead of top_srcdir.
1998-10-08 over04.sgml *mgd*
Remove duplicated "Scheduling a Model Swarm" section.
1998-10-08 overbook.tex.in *mgd*
Include tex/macros.tex.

1998-06-16 overmeta.sgml *mgd*
Use overbookrevhistory.sgml instead of overrevhistory.sgml. Scale graphic to 100%.
1998-06-16 Makefile.am *mgd*
(GENERATED_SGML): Use $(NAME)revhistory.sgml instead of overrevhistory.sgml. (EXTRA_DIST): over.ent renamed to overbook.ent.
1998-06-14 overbook.sgml *mgd*
Add JPEG notation.
1998-06-14 Makefile.am *mgd*
Include $(top_srcdir)/Makefile.common.
1998-06-11 overbook.sgml *mgd*
Include versions entity.
1998-06-11 overmeta.sgml *mgd*
Scale graphic to 75%.
1998-06-10 over.ent *alex*
Made all SYSTEM identifiers PUBLIC (listed in catalog now).
1998-06-10 Makefile.am *alex*
(GENERATED_SGML): Added for revhistory. (noinst_DATA): Added. (SGML, SGML_FILES): Separated out generated/non-gen files.
1998-06-10 over.ent *alex*
Created overrevhistory.sgml entity.
1998-06-10 overmeta.sgml *alex*
Replaced hardcoded revhistory with the new entity overrevhistory.sgml.
1998-06-08 overmeta.sgml *alex*
Changed corpauthor to be inserted from Global public identifiers.
1998-06-07 overbook.sgml *mgd*
Use public identifiers.
1998-06-03 over02.sgml, over03.sgml, over04.sgml *alex*
Fixed inappropriately positioned PARA tags.
1998-05-22 Makefile.am *mgd*
New file.
1998-05-22 Makefile *mgd*
Remove.
1998-05-22 overbook.tex.in *mgd*
New file.
1998-05-22 overbook.sgml *mgd*
Set ID of book to "overbook".
1998-05-21 *mgd*
Begin revision log.

Table of Contents

Chapter 1. Mag 1x: Experimental Procedure in a Computer

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

1. Set up the physical system to be studied.
2. Set up and calibrate the instrumentation
3. Run the experimental system and record the outputs of the instrumentation.
4. Analyze results.
5.
   - Change experimental and instrumental setup
   - Go to 3.
6. Publish paper -> tenure -> fame -> etc.....

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

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

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

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.

Chapter 2. 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.
4. Interact with the experiment via the data produced by the instrumentation objects to perform a series of controlled experimental runs of the system.
5. Depending on what is observed in stage 4), alter the experimental or instrumental "apparatus" and go back to 3).
6. Publish paper *including* detailed specification of the experimental set-up so that others can recreate your experiment and verify your results.

Chapter 3. Mag 3x: Swarm Structures

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

3.1. Model Swarms

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

In addition to the object collection, the model swarm also contains a schedule of activity on the model. The schedule defines the effect of passing time on the model. For the simple heatbugs schedule, the execution is simply to update the HeatSpace (diffusing heat across the world) and then telling each Heatbug agent to move itself.

Model swarms consist of a set of inputs and outputs. The inputs to the HeatbugModelSwarm are model parameters: things like the size of the world, the number of HeatBugs, and the diffusion rate of heat. The outputs of the HeatbugModelSwarm are the observables of the model: the individual Heatbugs, the distribution of heat across the world, etc.

3.2. Observer Swarms

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

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

3.3. 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.

Chapter 4. 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.

4.1. Building a Model Swarm

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

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

 id modelActions; // scheduling data structures
 id modelSchedule;

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

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

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

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

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

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

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

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

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

4.3. Building Agents

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

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

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

```
  outputHeat = [uniformRandom rMin: minOutputHeat Max: maxOutputHeat];

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

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

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

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

4.4. 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.

4.5. 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.

4.6. 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 exactly like building a model swarm.

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

  id displayActions; // schedule data structs
  id displaySchedule;
```

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

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

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

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

4.7. Building a Data Graph

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

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

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

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

In the case of an AverageSequence, an entire collection of objects is presented to it. The AverageSequence then extracts data from all the objects in the collection (in this case a List) using the provided message (in this case `getUnhappiness`), and generate a datapoint from these values by averaging them.

4.8. The main() function

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, const 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!

Chapter 5. 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 our examples and try to write your own applications. Good luck, and may your simulations be successful!