Akaroa 2.5
User’s Manual

G. Ewing and K. Pawlikowski
(Department of Computer Science)
and D. McNickle
(Department of Management)
University of Canterbury

TR-COSC 07/98, September 1998

The contents of this work reflect the views of the authors who are responsible for the facts and accuracy of the data presented. Responsibility for the application of the material to specific cases, however, lies with any user of the report and no responsibility in such cases will be attributed to the author or to the University of Canterbury.

Abstract

This document describes how to use Akaroa 2, a package for executing quantitative stochastic simulations on multiple processors using the method of Multiple Replications In Parallel (MRIP).
Akaroa 2.5
User’s Manual

Greg Ewing
Krzysztof Pawlikowski
Donald McNickle

September 4, 1998
AKAROA: Copyright © June 1992 (original version), June 1995 (AKAROA II), Department of Computer Science, University of Canterbury, Christchurch, New Zealand

The original version of AKAROA was designed at the Department of Computer Science, University of Canterbury in Christchurch, New Zealand, by Associate Professor K. Pawlikowski and Victor Yau (Computer Science) and Dr D. McNickle (Management). A contribution from Peter Smith (Computer Science) is also acknowledged. The project was partially sponsored by Telecom Australia Research Laboratories in Melbourne. The current version (AKAROA II) is a reimplementation by Dr Greg Ewing (Computer Science).

The AKAROA package can be used free of charge for teaching and non-profit research activities at universities and research institutes, but we would appreciate it if users of AKAROA clearly acknowledge using this package as a tool in their simulation studies when presenting or publishing their results.

Before using AKAROA for any other purposes, please consult Associate Prof. K. Pawlikowski, Department of Computer Science, University of Canterbury, Christchurch, New Zealand

Phone: (03) 364-2351 (national), +64 3 364 2351 (international)
Fax: (03) 364-2999 (national), +64 3 364 2999 (international)
Email: k.pawlikowski@cosc.canterbury.ac.nz
WWW: http://www.cosc.canterbury.ac.nz/ krys
Contents

1 Introduction 1
  1.1 Using Akaroa 2

2 Writing a simulation for Akaroa 3
  2.1 Example simulation program 3
  2.2 Compiling a simulation program 4
  2.3 Using a simulation program 4
  2.4 Observing more than one parameter 4
  2.5 Random Numbers 5
    2.5.1 Algorithm used by AkRandom 5
  2.6 Terminating Simulation vs. Steady-State Simulation 5

3 Running a simulation under Akaroa 7
  3.1 Parts of the Akaroa system 7
  3.2 Starting up the Akaroa system 7
  3.3 Running a simulation 8
    3.3.1 Running on particular hosts 9
    3.3.2 Passing options to the simulation program 9
    3.3.3 Controlling the random number seed 9
    3.3.4 Messages you may get from akrun 10
  3.4 Adding engines to a running simulation 10
  3.5 Monitoring the Akaroa system 11
    3.5.1 Examples 11
    3.5.2 Column headings 12
  3.6 Shutting down the Akaroa system 12
  3.7 Debugging a simulation 12
    3.7.1 Sending diagnostic information 12
    3.7.2 Running a simulation engine under a debugger 13
    3.7.3 Precautions against excessively short runs 13
  3.8 Graphical User Interface 13
    3.8.1 The main akgui window 13
    3.8.2 Starting a simulation 14
    3.8.3 Simulation window 14
    3.8.4 Examining an existing simulation 15
    3.8.5 Quitting akgui 15

4 The Akaroa Environment 17
  4.1 Environment Variables 17
  4.2 Environment Syntax 18
# CONTENTS

5 Akaroa Library Routines .......................... 21
  5.1 Random Number Distributions .................. 21
    5.1.1 Synopsis .................................. 21
    5.1.2 Descriptions .............................. 21
  5.2 Queues ....................................... 22
    5.2.1 Synopsis .................................. 22
    5.2.2 Using Queues ............................. 23
    5.2.3 Methods .................................. 23
  5.3 Priority Queues ................................ 23
    5.3.1 Synopsis .................................. 23
    5.3.2 Using Priority Queues ..................... 23
  5.4 Process Manager ............................... 24
    5.4.1 Synopsis .................................. 24
    5.4.2 Creating a process ......................... 24
    5.4.3 Stack size ................................ 25
    5.4.4 Scheduling ................................ 25
    5.4.5 Other routines ............................. 25
  5.5 Resources ..................................... 26
    5.5.1 Synopsis .................................. 26
    5.5.2 Methods .................................. 26
  5.6 AkSimulation .................................. 26
    5.6.1 Synopsis .................................. 27
    5.6.2 Using AkSimulation ......................... 27

6 Examples ........................................ 31
  6.1 An M/M/1 Queueing System ....................... 31
  6.2 A Multiprocessing Computer System ............. 32
  6.3 A Terminating Simulation ....................... 33

A Obsolete Facilities ............................. 37
  A.1 Event Manager ................................ 37
    A.1.1 Event Manager Routines .................... 37

B Adding an analysis method to Akaroa ............... 39
  B.1 Introduction ................................ 39
  B.2 What an analysis method does .................. 39
    B.2.1 Checkpoints .............................. 39
  B.3 Implementing an analysis method ............... 40
    B.3.1 Steps to implementing an analysis method . 40
    B.3.2 Class ParameterAnalyser .................. 40
    B.3.3 Declaring your analyser to Akaroa .......... 42
    B.3.4 Adding a value for AnalysisMethod .......... 42
    B.3.5 Adding your code to the Makefile .......... 42
    B.3.6 Recompiling Akaroa ........................ 42
  B.4 Accessing the Akaroa Environment ............... 43
    B.4.1 Retrieving Akaroa environment variables .... 43
    B.4.2 Defining new Akaroa environment variables . 43

Bibliography ....................................... 45
Chapter 1

Introduction

Quantitative stochastic simulation is a useful tool for studying performance of stochastic dynamic systems, but it can consume much time and computing resources. Even with today’s high speed processors, it is common for simulation jobs to take hours or days to complete.

Processor speeds are increasing as technology improves, but there are limits to the speed that can be achieved with a single, serial processor. To overcome these limits, parallel or distributed computation is needed. Not only does this speed up the simulation process, in the best case proportionally to the number of processors used, but the reliability of the program can be improved by placing less reliance on a single processor.

One approach to parallel simulation is to divide up the simulation model and simulate a part of it on each processor. However, depending on the nature of the model it can be very difficult to find a way of dividing it up, and if the model does not divide up readily, the gain from parallelising it will be less than proportional to the number of processors. Even in cases where the model can be parallelised easily, more work is required to implement a parallel version of the simulation than a serial one.

Akaroa takes a different approach to parallel simulation, that of multiple replications in parallel or MRIP [1-8]. Instead of dividing up the simulation program, multiple instances of an ordinary serial simulation program are run simultaneously on different processors.

These instances run independently of one another, and continuously send back to a central controlling process observations of the simulation model parameters which are of interest. The central process calculates from these observations an overall estimate of the mean value of each parameter. When it judges that it has enough observations to form an estimate of the required accuracy, it halts the simulation.

Since the simulations run independently, if there are \( n \) copies of the simulation running on \( n \) processors they will on average produce observations at \( n \) times the rate of a single copy, and therefore produce enough observations to halt the simulation after \( 1/n \)th of the time. So the MRIP technique can be expected to speed up the simulation approximately in proportion to the number of processors used.

MRIP also provides a degree of fault tolerance. It doesn’t matter which instance of the simulation the estimates come from, so if one processor fails, the program it was running can be restarted and the simulation continued without penalty. Alternatively, the simulation can simply be continued with one less processor and take proportionately longer to complete.

In summary, the advantages of the MRIP technique are that it can be applied to any simulation program without the need to parallelise or modify it in any way; it provides a speedup proportional to the number of processors; and it improves the reliability of the simulation.
1.1 Using Akaroa

To use Akaroa, the user writes a simulation program which models the system to be studied and, when executed, collects a series of observations of one or more parameters of the processes being simulated. Akaroa automatically launches and manages the execution of a number of copies of this program on available processors; each such copy is called a simulation engine. Each simulation engine runs independently of the others and generates its own sequence of observations, from which local estimates of the parameters are calculated. Akaroa collects these local estimates when they are produced and calculates a global estimate of each parameter.

The user specifies the required precision and confidence level for each parameter. When the global estimates of all parameters have reached the required precision at the required level of confidence, the simulation engines are automatically stopped, and the results are reported.

If any of the simulation engines fails for some reason, the rest are allowed to continue, and the global estimates are calculated using values from the remaining engines. Akaroa thereby provides a certain amount of fault tolerance - if one of the processors goes down, the simulation will continue, although it will take longer to complete.
Chapter 2

Writing a simulation for Akaroa

Writing a simulation program to run under Akaroa is very straightforward. You write a program in C++ to simulate the system you wish to study, using whatever techniques you would normally use. Whenever your program generates an observation of one of the parameters you are interested in, you make a call to the Akaroa library to communicate this observation to the Akaroa system.

2.1 Example simulation program

Here is an example of a very simple simulation program designed to run under Akaroa. It simulates a process which generates random numbers in the range 0 to 1, and gives each number to Akaroa as an observation. (The source of this program, and the other examples in this manual, can be found in the examples directory of the Akaroa installation directory. Consult your site administrator for the location of this directory.)

/*
 * uni.C - A very simple simulation engine
 */

#include <akaroa.H>
#include <akaroa/distributions.H>

int main(int argc, char *argv[]) {
  for (;;) {
    double x = Uniform(0, 1);
    AkObservation(x);
  }
}

This example demonstrates how to use one of the most important Akaroa library routines. AkObservation takes an observation and makes it known to the Akaroa system, which updates its estimate of the mean value. As long as the estimate has not yet reached the required accuracy, AkObservation will return and allow the simulation to continue. When the estimate reaches the required precision, Akaroa will automatically terminate the simulation.

This example also uses the routine Uniform, which returns uniformly distributed random numbers in the specified range. You should always use Akaroa library routines to obtain random numbers; for more information, see section 2.5.

1 You may also write the program in C and compile it with the C++ compiler, although you will not be able to use all of the modelling facilities provided with Akaroa.
2.2 Compiling a simulation program

The examples directory contains a Makefile for compiling the example programs. You can copy this Makefile to your own directory and use it for compiling your own simulation programs.

For example, if you have also copied the file uni.C from the examples directory, you can compile it with the command

```
% make uni
```

If your simulation program consists of a single source file, you can compile it with the command `make xxx`, where `xxx` is the name of the program, without making any changes to the Makefile. But if your program is built from more than one source file, you will have to add a rule for linking it to the Makefile. An example of such a rule is included at the bottom of the Makefile.

2.3 Using a simulation program

A simulation program may, without modification, be used in two ways. It may be launched manually and run stand-alone, or it may be launched automatically by Akaroa as a simulation engine. When run stand-alone, it will write a report of the final estimate of each parameter to standard output when finished. Here is an example of the output produced by running the uni program stand-alone:

```
% uni
Param  Estimate  Delta  Conf  Var   Count  Trans
 1  0.483686  0.0218746  0.95  8.55314e-05  756  252
```

*Estimate* is Akaroa’s estimate of the mean value of the parameter, *Delta* is the half-width of the confidence interval, *Conf* is the confidence level, and *Var* is the variance of the estimate. *Count* is the total number of observations collected, and *Trans* is the number of observations that were discarded during the transient phase, before the system settled down into a steady state.

2.4 Observing more than one parameter

If your simulation produces observations of more than one parameter, you need to call `AkDeclareParameters` before starting your simulation, and pass it the number of parameters you wish to estimate. Then, each time you call `AkObservation`, you pass it the parameter number along with the observation.

For example, here’s an extension of uni which generates observations of two parameters:

```c
/*
 * uni2.C - A very simple 2-parameter simulation engine
 */

#include <akaroa.H>
#include <akaroa/distributions.H>

int main(int argc, char *argv[]) {
    AkDeclareParameters(2);
    for (;;) {
        double x = Uniform(0, 1);
```
2.5 Random Numbers

When running multiple replications of a simulation model in parallel, it is important that each simulation engine uses a unique stream of random numbers, independent of the streams used by other simulation engines. For this reason, if your simulation requires random numbers, you should always obtain them from the Akaroa system, so that Akaroa can coordinate the random number streams received by different simulation engines.

The simplest way is to use the random number distribution routines provided in the Akaroa library, described in section 5.1. If you need a distribution that is not provided in the library, you will need to write your own distribution generator, using the routine AkRandom as a basic source of random numbers:

```c
unsigned long AkRandom();
```

Each time AkRandom is called, it returns a random integer \( n \) such that \( 1 \leq n < 2^{31} - 1 \).

2.5.1 Algorithm used by AkRandom

AkRandom uses a multiplicative linear congruential random number generator together with a series of multiplying coefficients to generate a sequence of random numbers made up of subsequences of length \( 2^{31} - 2 \), one subsequence for each multiplier. Currently 50 multipliers are available, for a total sequence length of 107,374,182,300 numbers.

These multipliers are taken from a list of optimal multipliers published by Fishman and Moore\(^2\), and they have been subjected to extensive statistical testing by those authors. For more information, including a list of the multipliers, see the on-line manual entry AkRandom(3).

2.6 Terminating Simulation vs. Steady-State Simulation

In steady-state simulation, the stream of observations produced by the simulation model is usually correlated. However, some types of simulation produce observations which are independent. An example is terminating simulation in which the simulation is run for a predetermined period, at the end of which a single data item is produced. To obtain a stream of data items for Akaroa to analyse as observations, the simulation must be repeated many times with different random number seeds. Because the repetitions are independent of each other, the data items produced are also independent.

In the case of independent observations, there is no transient phase, and there is no need to use a method such as Batch Means or Spectral Analysis to analyse the observations.

---

take advantage of these facts, Akaroa has an independent observation mode. This mode is selected by making the following call to the AkObservationType routine:

\[
\text{AkObservationType(AkIndependent);}
\]

You must make this call before calling AkDeclareParameters or calling AkObservation for the first time. (If you call it later, it will have no effect, and Akaroa will assume that the observations are correlated as usual.) For an example of a simulation which uses this routine, see Chapter 6.

When independent observation mode is selected, the setting of the AnalysisMethod environment variable is ignored. No transient observations are discarded, and the variance of the estimate of the mean is estimated using

\[
\hat{\sigma}^2_X = \frac{1}{N} \hat{\sigma}^2_{\bar{X}_i}
\]

where \(X_i\) is the \(i\)th data item and \(N\) is the number of independent data items, and

\[
\hat{\sigma}^2_{\bar{X}_i} = \frac{1}{N-1} \sum_{i=1}^{N} (X_i - \bar{X})^2
\]
Chapter 3

Running a simulation under Akaroa

This section explains how to run multiple replications of your simulation in parallel under the Akaroa system.

3.1 Parts of the Akaroa system

The Akaroa system consists of three main programs, akmaster, akslave, and akrun, plus three auxiliary programs akadd, akstat and akgui.

Akmaster is the master process which coordinates all other processes in the Akaroa system. Before you can use Akaroa, there must be an akmaster process of yours running on some host which can communicate with all the other hosts you wish to use.

There must be an akslave process running on each host that you wish to use to run a simulation engine. Akmaster uses the akslave to launch the simulation engine and to help establish communication with it.

The host on which akmaster is running may also, if you wish, run an akslave, and therefore be used to run a simulation engine.

Once the akmaster and any desired akslaves are running, you may use akrun to start a simulation. Akrun takes as arguments the name of the program you wish to run as a simulation engine, any arguments to be passes to that program, and the number of hosts on which you want to run it.

Akrun instructs akmaster to launch the simulation on the requested number of hosts. Akmaster chooses this many hosts from among those running akslaves, and instructs the akslaves on those hosts to launch the requested program as a simulation engine.

Akmaster collects local estimates from the simulation engines, calculates global estimates, and decides when to stop the simulation. When the simulation is over, akmaster sends the final global estimates back to akrun, which reports them to the user and exits.

Akadd (section 3.4) is used to add more simulations to a running simulation. Akstat (section 3.5) is used to obtain information about the state of the Akaroa system. Akgui (section 3.8) provides a graphical user interface for starting and monitoring simulations that can be used instead of, or in addition to, akrun and akstat.

3.2 Starting up the Akaroa system

To start up the Akaroa system:

1. Start akmaster running in the background on some host.
2. On each host where you wish to run a simulation engine, start `akslave` running in the background.

You may accomplish these steps either by using `rsh`, or by logging into the relevant hosts and running the programs directly. However, you should take care about the environment in which each akslave process runs. The program name that you give to akrun will be passed as-is to each akslave, and you must ensure that the akslave will be able to find it, either by using a full pathname, or by including the directory where it resides in your search path before launching the akslaves.

If you are going to launch akslaves using rsh, you must make any necessary additions to your search path in your `.cshrc` file (or, if you use `tcsh`, your `.tcshrc` file), not just in the shell from which you issue the rsh command.

**Example: Starting up Akaroa via `rsh`**

Here is an example of starting up Akaroa on two hosts, `purau` and `mohua`, with the akmaster running on a third host, `whio`. It assumes that the user is already logged into whio, and has set up her path variable in her `.cshrc` file to include the directory where her simulation programs reside, and the directory where the akaroa programs reside.

```
whio% akmaster &
[1] 14018
whio% rsh purau 'akslave &'
[1] 14117
whio% rsh mohua 'akslave &'
[1] 14136
whio%
```

Once an akslave is up and running, it breaks its links with the rsh. So, if the rsh command exits without any error messages, you know that the akslave has been launched successfully.

### 3.3 Running a simulation

The `akrun` command starts a simulation, waits for it to complete, and writes a report of the results to standard output. The basic usage of the akrun command is:

```
akrun -n num_hosts command [ argument... ]
```

where `num_hosts` is the number of hosts on which you wish to run simulations, `command` is the name of the program you wish to run as a simulation engine, and the `arguments` are the arguments, if any, that you want to pass to each simulation engine.

Once Akaroa is started up, you may run as many simulations as you like. You may even run more than one simulation at a time, although they will compete with each other for processing resources.

You can make a new host available for running simulation engines at any time by starting an akslave on that host (although it will only be available to simulations subsequently started, not to any already running).

**Example: Running `uni` under Akaroa**

Assuming that Akaroa has been started up in the manner of the previous example, here is an example showing how to run the `uni` program on two hosts, and the typical output produced:
3.3. **RUNNING A SIMULATION**

whio% akrun -n 2 uni
Simulation ID = 17
Simulation engine started: host = pukeko, pid = 23672
Simulation engine started: host = purau, pid = 434

<table>
<thead>
<tr>
<th>Param</th>
<th>Estimate</th>
<th>Delta</th>
<th>Conf</th>
<th>Var</th>
<th>Count</th>
<th>Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.503476</td>
<td>0.0157353</td>
<td>0.95</td>
<td>4.42582e-05</td>
<td>1530</td>
<td>255</td>
</tr>
</tbody>
</table>

whio%

### 3.3.1 Running on particular hosts

If you just specify a number of hosts to akrun with the `-n` option, the Akaroa system arbitrarily chooses this many hosts from among those running akslave processes. Akaroa will try to spread the simulation load that it is given evenly over the hosts available, but it only takes Akaroa processes into account. It doesn’t know about non-Akaroa processes, or even Akaroa processes belonging to another user.

If Akaroa’s simple method of load balancing is not sufficient, you can specify which hosts to use by giving `-H` options to akrun. Each `-H` option is followed by the name of a host. For example,

```
whio% akrun -H mohua -H raupo uni
```

will run simulation engines on the hosts *mohua* and *raupo* (provided they are both running akslaves).

### 3.3.2 Passing options to the simulation program

If your simulation program requires arguments that begin with a hyphen, you will need to separate them from the options to akrun by using a double hyphen, for example,

```
akrun -n 5 -- mysim -a 42 -b 6.8
```

All the arguments after `--` are taken to be part of the simulation command.

### 3.3.3 Controlling the random number seed

Each time you invoke akrun to start a simulation, Akaroa’s random number generator is started with the same seed. If you want to run a simulation several times using different invocations of akrun, with a different stream of random numbers each time, you will need to ensure that the random number generator is restored to the state it was in at the end of the previous run.

To find out the state of the random number generator at the end of a run, give the `-s` option to akrun, for example:

```
whio% akrun -n 1 -s uni
Repetition 1:
Simulation engine 3921 started on purau
Repetition 2:
Simulation engine 3922 started on purau
RandomNumberState: 0:20000
<table>
<thead>
<tr>
<th>Param</th>
<th>Estimate</th>
<th>Delta</th>
<th>Conf</th>
<th>Var</th>
<th>Count</th>
<th>Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.502473</td>
<td>0.0251216</td>
<td>0.95</td>
<td>0.000163424</td>
<td>503</td>
<td>0</td>
</tr>
</tbody>
</table>

whio%
```

Note the `RandomNumberState` (0:20000 in this example) written out before the report. This indicates the state of the random number generator at the end of the last repetition. To run the simulation again with the random number generator initialised to this state, give it to akrun using the `-r` option:
whio% akrun -n 1 -r 0:20000 uni
Repetition 1:
  Simulation engine 3928 started on purau
Repetition 2:
  Simulation engine 3929 started on purau

<table>
<thead>
<tr>
<th>Param</th>
<th>Estimate</th>
<th>Delta</th>
<th>Conf</th>
<th>Var</th>
<th>Count</th>
<th>Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.494674</td>
<td>0.0247054</td>
<td>0.95</td>
<td>0.000158099</td>
<td>535</td>
<td>0</td>
</tr>
</tbody>
</table>

whio%

This time the results are different, as expected, since they are based on a different random number sequence.

### 3.3.4 Messages you may get from akrun

Akrun will emit warning messages if certain events occur which could affect the progress of the simulation:

#### Loss of simulation engine

If a simulation engine crashes, a warning message is issued and the simulation is continued using the remaining engines. This will not affect the validity of the results, but the simulation may take longer to complete.

#### Exhaustion of random number stream

If the random number sequence provided by Akaroa is exhausted before the simulation completes, a warning message is issued and the sequence is reused starting from the beginning. This could affect the validity of the results, so results obtained after this has occurred should be treated with caution.

The total length of Akaroa’s random number sequence is about $10^{12}$.

### 3.4 Adding engines to a running simulation

The `akadd` command can be used to add simulation engines to a running simulation. You can use it to replace engines which have been lost for some reason, or to speed up the simulation if more hosts become available.

To start a given number of new engines, the usage is:

```
akadd -s sid -n num-engines
```

where `sid` is the simulation ID reported by `akrun` when the simulation was started. For example,

```
akadd -s 42 -n 5
```

will add 5 new engines to the simulation with ID 42.

To add simulation engines running on particular hosts, the usage is:

```
akadd -s sid -H hostname...
```

For example,

```
akadd -s 42 -H purau matata kahu
```

will add three new engines running on the hosts purau, matata and kahu.
3.5 Monitoring the Akaroa system

The `akstat` command can be used to obtain information about the status of the Akaroa system: what hosts are available, what simulations are running, and what progress each simulation is making.

There are two kinds of options to `akstat`. Upper case options control which kind of information to display, and lower case options restrict the information to particular simulations, engines or parameters.

The `-H` option produces a list of hosts which are running akslave processes, together with the number of simulation engines running on each host.

The `-S` option produces a list of the currently running simulations.

The `-G` option produces information about the current global estimates of parameters being observed.

The `-E` option produces information about the state of simulation engines.

The `-L` option produces information about the current local estimates of parameters from simulation engines.

Without any other options, the requested information is listed for all existing simulations, engines or parameters. The `-s` option restricts the listing to a particular simulation ID, `-e` to a particular engine number, and `-p` to a particular parameter.

Without any options at all, `akstat` assumes the `-H` and `-S` options.

### 3.5.1 Examples

```bash
akstat
List all hosts and all simulations.

akstat -S
List all simulations.

akstat -G
List global estimates of all parameters of all simulations.

akstat -G -s 27
List global estimates of all parameters of simulation ID 27.

akstat -G -s 27 -p 3
List global estimate of parameter 3 of simulation ID 27.

akstat -E
List all simulation engines of all simulations.

akstat -E -s 27
List all simulation engines of simulation ID 27.

akstat -E -s 27 -e 2
List engine 2 of simulation ID 27.

akstat -GL -s 27
List all global and local estimates of simulation ID 27.

akstat -L -e 2
List local estimates of all parameters for engine 2 of all simulations which have at least 2 engines.
```
3.5.2 Column headings

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST</td>
<td>Host name</td>
</tr>
<tr>
<td>PID</td>
<td>Process ID</td>
</tr>
<tr>
<td>ENGINES</td>
<td>Number of engines running on host</td>
</tr>
<tr>
<td>SID</td>
<td>Simulation ID</td>
</tr>
<tr>
<td>EID</td>
<td>Engine ID</td>
</tr>
<tr>
<td>PAR</td>
<td>Parameter number</td>
</tr>
<tr>
<td>PARMS</td>
<td>Number of parameters</td>
</tr>
<tr>
<td>ENGS</td>
<td>Number of engines belonging to simulation</td>
</tr>
<tr>
<td>RANDOM</td>
<td>State of random number generator</td>
</tr>
<tr>
<td>FLAGS</td>
<td>Internal state flags (see the akstat(1) man page)</td>
</tr>
<tr>
<td>COMMAND</td>
<td>Command and arguments</td>
</tr>
<tr>
<td>STATE</td>
<td>State of simulation engine</td>
</tr>
<tr>
<td>MEAN</td>
<td>Estimate of the mean</td>
</tr>
<tr>
<td>PREC</td>
<td>Relative precision of the estimate</td>
</tr>
<tr>
<td>VARIANCE</td>
<td>Variance of the estimate</td>
</tr>
<tr>
<td>OBS</td>
<td>Number of observations</td>
</tr>
<tr>
<td>TRANS</td>
<td>Number of transient-phase observations</td>
</tr>
<tr>
<td>CHKPTS</td>
<td>Number of checkpoints received</td>
</tr>
<tr>
<td>CP/MIN</td>
<td>Average number of checkpoints per minute received during the last 10 minutes</td>
</tr>
<tr>
<td>LAST CHKPT</td>
<td>Date and time at which the last checkpoint was received</td>
</tr>
</tbody>
</table>

For more detailed information, see the man page for akstat(1).

3.6 Shutting down the Akaroa system

To shut down the Akaroa system, simply kill the akmaster process. Any akslaves, akruns or simulation engines attached to it will automatically terminate.

You can remove a host from the pool available for running simulation engines, without shutting down the whole Akaroa system, by just killing the akslave on that host.

3.7 Debugging a simulation

Before you run your simulation under Akaroa, you should debug it as much as possible stand-alone. If you compile your simulation program with the -g option, you can run it under a source-level debugger and use all of the usual debugging techniques. Only when you are satisfied that your simulation program runs successfully on its own should you attempt to run it under Akaroa.

3.7.1 Sending diagnostic information

Usually, a simulation that runs correctly stand-alone will also run correctly under Akaroa. However, sometimes you may encounter a bug that only shows up under Akaroa. To help find such bugs, your simulation program can send diagnostic output using the AkMessage routine:

```c
AkMessage(format, arg1, arg2, ...);
```

AkMessage formats its arguments like printf and sends the result to the akrun process that started the simulation, which in turn writes it to standard error.
Note that the standard input, output and error of a simulation engine running under Akaroa are connected to /dev/null, so anything written to them will not be seen. ¹

3.7.2 Running a simulation engine under a debugger

As an alternative to producing diagnostic output, you can persuade Akaroa to run your simulation engine under a debugger by using a command such as

```
akrun -n 1 xxgdb mysim
```

You will need to supply any required arguments to your simulation engine in the run command to xxgdb. You will also need to ensure that the akslave is running in an environment where the DISPLAY variable is set correctly. The easiest way to ensure this is to start the akslave from an xterm on the relevant host.

3.7.3 Precautions against excessively short runs

In sequential stochastic simulation, sometimes the simulation stopping criteria are spuriously met, causing the run to be stopped too soon and producing results which are not reliable. If you are concerned about this possibility, you can guard against it by running the simulation more than once (with a different random number seed each time) and disregarding results from any runs which are much shorter than the others (i.e. produced much fewer observations).

To automate this process, akrun has a `-R n` option, which causes it to run the simulation `n` times with different random number sequences. For each parameter, the final result reported is the one from the run which submitted the greatest number of observations for that parameter.

Increasing the value of `n` will reduce the probability of a spurious final result being reported, but the simulation will take longer to complete.

The `-A` option may be used to obtain the results from all of the repetitions. Without this option, akrun only reports the final results chosen.

3.8 Graphical User Interface

The akgui program provides a graphical user interface to the Akaroa system as an alternative to the shell command interface provided by akrun, akadd and akstat.

*Note:* Akgui does not yet provide access to all the facilities of Akaroa. For some tasks you may need to use the shell command interface.

Before using akgui, you will need to start up the Akaroa system using the akmaster and akslave commands, as described in section 3.2.

3.8.1 The main akgui window

The main window of akgui displays two lists:

1. The *host list* shows the names of all hosts running akslave processes, their process IDs, and the number of simulation engines running on that host.

2. The *simulation list* shows information about the currently running simulations: the simulation ID, the number of parameters being estimated, the number of simulation engines, and the command name and arguments.

¹In some earlier versions of Akaroa, text written to the standard error of a simulation engine was reported by akrun. This is no longer supported; AkMessage should be used instead.
3.8.2 Starting a simulation

To start a simulation, click the New Simulation button in the main window. Enter the following information into the form which appears:

1. The simulation program name and arguments.
2. The required precision and confidence (if they are different from the default values initially displayed).
3. The number of simulation engines to launch. Alternatively, you may choose the Select Hosts option and select particular hosts on which to run engines.

You can optionally change the values of the following settings:

1. The analysis method (Spectral or BatchMeans).
2. The checkpoint spacing factor and method (see Chapter 4).

When you have filled out the form, click the Run button to begin the simulation. A simulation window appears as described in the next section.

3.8.3 Simulation window

The simulation window displays the status of a running simulation and provides means of adding engines or killing the simulation. There are four information display areas:

1. The box at the top of the window displays information identifying the simulation (command and arguments, and simulation ID) and the status of the simulation (Running, Finished or Failed).
2. The Simulation Engines table lists the host, process ID and state of each simulation engine belonging to the simulation. The possible states are:
   - launching: The engine has been launched but has not yet contacted the akmas-ter process.
   - alive: The engine is running and reporting estimates.
   - dead: The engine has died unexpectedly.
3. The Relative Precision box displays a bar graph for each parameter being estimated. The red bar shows the relative precision of the current global estimate, and the black triangle shows the relative precision requested for that parameter.
4. The Global Estimates table shows the current global estimate of each parameter, its relative precision, the total number of observations received for that parameter, and the number of observations discarded during the transient phase. It also shows the checkpoint arrival rate in checkpoints per minute (in total from all engines) and the date and time of arrival of the last checkpoint received.

To add more engines to the simulation, click the Add Engines button. A form appears similar to the one for selecting engines when the simulation was started.

When the simulation finishes, the simulation status changes to Finished. The engine table, precision bars and global estimate table are removed and replaced with a results table showing the final estimate of each parameter, the half-width of its confidence interval, and the total and transient observation counts. When you have finished examining the results, you can dismiss the window by clicking the Close Window button.

To kill the simulation prematurely, click the Kill Simulation button.
3.8.4 Examining an existing simulation

You can examine the status of any running simulation by double-clicking its entry in the main akgui window. If the simulation was started using akgui, this will bring its simulation window to the front. If it was started using akrun (or using a different instance of akgui), a simulation window will be created showing the status of the simulation.

The simulation window behaves slightly differently depending on whether the simulation was started by akgui or not. If the simulation was started by akgui, the simulation window must remain in existence until the simulation finishes – you cannot close the window without killing the simulation.

In contrast, if the simulation was not started by akgui, you can close the window at any time without affecting the simulation. Moreover, you cannot kill the simulation using akgui – to do that, you would have to find the akrun process which started the simulation and kill it.

In either case, the Add Engines button can be used to add engines to the simulation.

3.8.5 Quitting akgui

The Quit button in the main akgui window quits akgui and closes any existing simulation windows. The same thing will happen if you close the main akgui window using your window manager.

Warning: Quitting akgui will kill any simulations started by it!
Chapter 4

The Akaroa Environment

The Akaroa Environment is a collection of variables which control the operation of the Akaroa system. When you start a simulation, \textit{akrun} looks for a file called “Akaroa” in the current directory, and, if it is present, reads environment settings from it.

\textit{Note: The Akaroa Environment is separate from the Unix environment. You cannot change an Akaroa Environment variable using \textit{setenv}.}

Here is an example of an Akaroa environment file which sets the desired precision and confidence level for the results of the simulation.

\begin{verbatim}
Precision = 0.01
Confidence = 0.90
\end{verbatim}

This specifies the precision of all parameters to be within $\pm 1\%$ at a confidence level of $90\%$.

Variables may be set globally for all parameters, or locally for individual parameters. The following example sets the confidence level of parameter 1 to 0.97, the precision of parameter 2 to 0.02, and the precision and confidence levels of all other parameters to 0.01 and 0.90.

\begin{verbatim}
Precision = 0.01
Confidence = 0.90
parameter 1 {
  \hspace{1em}Confidence = 0.97
}
parameter 2 {
  \hspace{1em}Precision = 0.02
}
\end{verbatim}

Variables not mentioned at all in the Akaroa file take on default values supplied by the Akaroa system.

You can specify an alternative file from which to read the Akaroa environment using the \textit{-f} option to \textit{akrun}, for example:

\begin{verbatim}
whio% akrun -f my_environment -n 2 uni
\end{verbatim}

4.1 Environment Variables

Here is a list of the Akaroa Environment variables you are most likely to want to set. The values after “=” are the default values.
Variables pertaining to all analysis methods

Precision = 0.05
Relative precision.

Confidence = 0.95
Confidence level.

AnalysisMethod = Spectral
Method of estimating variance. In the current version of Akaroa, two methods are available: Spectral and BatchMeans. ¹

Variables pertaining to Spectral Analysis

CPSpacingMethod = Linear
Method used to determine spacing between checkpoints (local estimates sent to the akmaster process). One of:

Linear
Constant number of observations between checkpoints.

Geometric
Number of observations between checkpoints increase geometrically.

CPSpacingFactor = 1.5
For Linear spacing, distance between successive checkpoints, relative to the length of the transient period.
For Geometric spacing, factor by which checkpoint spacing increases after each checkpoint.

PeriodogramPoints = 25
Number of points of the periodogram used in spectral analysis.

PolynomialDegree = 2
Degree of the polynomial fitted to the periodogram in spectral analysis.

Variables pertaining to Batch Means

InitBatchSize = 50
Initial batch size. The final batch size chosen will be a multiple of this size.

AnalysedSeqLen = 100
Length of the sequence of batch means tested for autocorrelation during the batch size selection phase.

AutoCorrSignif = 0.1
Significance level at which the coefficients of autocorrelation of the batch means are tested when determining whether to accept a batch size.

4.2 Environment Syntax

The formal syntax of the Akaroa environment file is described by the following grammar. Items enclosed in curly braces {...} may be repeated zero or more times.

An identifier is a letter followed by zero or more letters or digits. An integer or float is an integral or floating point constant written in the usual way. A string is a sequence of characters enclosed in double quotes.

4.2. ENVIRONMENT SYNTAX

\[
\begin{align*}
\text{environment} & \rightarrow \{ \text{setting} \mid \text{parameter} \} \\
\text{setting} & \rightarrow \text{identifier} \ '=$' \ \text{value} \\
\text{value} & \rightarrow \text{integer} \mid \text{float} \mid \text{identifier} \mid \text{string} \\
\text{parameter} & \rightarrow \text{parameter' integer '}{' \{ \text{setting} \} '}{'}
\end{align*}
\]
Chapter 5

Akaroa Library Routines

Akaroa comes with a set of library routines and classes designed to help you write stochastic discrete-event simulations. Their use is optional – you may use them if they help, or you may use just the core Akaroa routines already described.

5.1 Random Number Distributions

Functions are available for providing random numbers drawn from a variety of commonly-used distributions. These functions all use AkRandom as a basic source of random numbers.

5.1.1 Synopsis

The following random number functions are defined:

```c
#include <akaroa/distribution.H>

real Uniform(real a, real b);
long UniformInt(long n0, long n1);
long Binomial(long n, real p);
real Exponential(real m);
real Erlang(real m, real s);
real HyperExponential(real m, real s);
real Normal(real m, real s);
real LogNormal(real m, real s);
long Geometric(real m);
real HyperGeometric(real m, real s);
long Poisson(real m);
real Weibull(real alpha, real beta);
```

5.1.2 Descriptions

- **real Uniform(real a, real b)**  
  Uniformly distributed reals in the range $a$ to $b$.

- **long UniformInt(long n0, long n1)**  
  Uniformly distributed integers in the range $n0$ to $n1$, inclusive.

- **long Binomial(long n, real p)**  
  Binomial distribution from $n$ items, each with a probability $p$ of being drawn.
real Normal(real m, real s)
Normal distribution with mean $m$ and standard deviation $s$.

real LogNormal(real m, real s)
Log-normal distribution with mean $m$ and standard deviation $s$.

real Exponential(real m)
Exponential distribution with mean $m$.

real HyperExponential(real m, real s)
HyperExponential distribution with mean $m$ and standard deviation $s$, $s > m$.

long Poisson(real m)
Poisson distribution with mean $m$, $m > 0$.

long Geometric0(real m)
long Geometric1(real m)
Geometric distributions with mean $m$, $m > 0$. Geometric0 returns integers $\geq 0$; Geometric1 returns integers $> 0$.

real HyperGeometric(real m, real s)
HyperGeometric distribution with mean $m$.

real Erlang(real m, real s)
Erlang distribution with mean $m$ and standard deviation $s$.

real Weibull(real alpha, real beta)
Weibull distribution with parameters $alpha$ and $beta$.

5.2 Queues

Class Queue implements a queue of objects of some specified type. Objects may be added to the tail of the queue and removed from the head. The queue may be tested for emptiness, and the number of objects in the queue may be determined. Objects may belong to more than one queue at a time, if desired.

5.2.1 Synopsis

Class Queue is defined as follows:

```cpp
#include <akaroa/queue.H>

template <class T>
class Queue {
public:
  Queue();
  virtual void Insert(T *item);
  virtual void Remove(T *item);
  virtual T *Next();
  virtual T *Head();
  virtual int Empty();
  virtual int Length();
};
```
5.2.2 Using Queues
When declaring a variable of type Queue, you need to specify the type of object the queue is to contain, e.g.

```
Queue<Customer> customersWaiting;
```

5.2.3 Methods

```
Queue::Insert(item)

Adds item to the tail of the queue.
```

```
Queue::Remove(item)

Removes item from the queue, if it is present (wherever it happens to be).
```

```
Queue::Next()

Removes one item from the head of the queue and returns a pointer to it. If the queue is empty, it returns null.
```

```
Queue::Head()

Returns a pointer to the head item of the queue, without removing it. If the queue is empty, it returns null.
```

```
Queue::Empty()

Returns true if there are no items in the queue, false otherwise.
```

```
Queue::Length()

Returns the number of items in the queue.
```

5.3 Priority Queues

PriorityQueue is a variant of class Queue which maintains its contents in order of priority. The priority of the elements is defined by a user-supplied method.

5.3.1 Synopsis

Class PriorityQueue is defined as follows:

```
#include <akaroa/priority_queue.H>

template <class T>
class PriorityQueue : public Queue<T> {
public:
    virtual void Insert(T *item);
    virtual void HigherPriority(T *item1, T *item2) = 0;
};
```

5.3.2 Using PriorityQueues

To use the PriorityQueue template to create a priority queue of a particular type, you have to implement a method called HigherPriority which takes pointers to two items of that type. The method should return true if the first one has higher priority than the second, false otherwise.
PriorityQueue::Insert(item) will then insert the given item in the appropriate place in the queue according to its priority in relation to the items already there. All other methods of PriorityQueue work the same as for Queue.

For example, here is a definition of a priority queue of objects of class Customer which the user has defined as having a height member. It arranges for taller customers to have priority over shorter ones.

```cpp
class MyPrioQ : public PriorityQueue<Customer> {
public:
    int HigherPriority(Customer *, Customer *);
};

int MyPrioQ::HigherPriority(Customer *c1, Customer *c2) {
    return c1->height > c2->height;
}
```

### 5.4 Process Manager

The Process Manager is provided to help you implement process-oriented discrete event simulations. It allows you to create multiple “lightweight processes”, or threads of execution, within the Unix process that is running your simulation. In this section, the term “process” refers to a lightweight process.

The Process Manager also maintains a *simulation clock*, and provides the means for processes to schedule themselves or other processes to execute at specified simulation times.

#### 5.4.1 Synopsis

The Process Manager defines the following types and functions:

```cpp
#include <akaroa/process.H>

typedef real Time;

class Process {
public:
    Process(long stackSize = 1024);
    void Schedule(Time delay);
protected:
    virtual void LifeCycle() = 0;
};

Time CurrentTime();
Process *CurrentProcess();
void Hold(Time delay);
void Hold();
void DeleteProcesses();
```

#### 5.4.2 Creating a process

Initially, there is one process executing the main program of your simulation. To create additional processes, you need to define a subclass of class Process, and give it a LifeCycle method. For example:
class Customer : public Process {
protected:
    void LifeCycle();
};

void Customer::LifeCycle() {
    EnterStore();
    WaitForServer();
    if (!AskFor(aRareItem))
        ComplainToManager();
    LeaveStore();
}

You could then create a new Customer process with:

    Customer *c = new Customer;

The newly created process is scheduled to execute at the current simulation time. When it gains control, it will execute its LifeCycle method.

Despite its name, the LifeCycle does not automatically cycle. If the LifeCycle method returns, the process’s thread will be terminated and the memory occupied by the Process object deallocated (i.e. the process will delete itself).

### 5.4.3 Stack size

By default, a new process is allocated 1024 bytes of stack space, plus some extra to allow for the requirements of the Process Manager. If this is not sufficient, you can specify a larger stack when you create a process:

    Customer *c = new Customer(5000);

It is important to give your processes enough stack space. Once created, a process’s stack cannot be extended; if the process runs out of stack space, your simulation will crash.

### 5.4.4 Scheduling

A process can be scheduled to execute at a specified simulation time. Process::Schedule(delay) will schedule the process to execute at the current simulation time plus delay; until then, the process will be blocked.

Hold(delay) blocks the current process until the simulation clock reaches the current time plus delay. It is equivalent to CurrentProcess() -> Schedule(delay).

Hold() with no arguments blocks the current process indefinitely. It will not run again until some other process schedules it.

Process scheduling is non-preemptive. Once a process is running, control is never transferred to another process until the current process either calls Hold or invokes Schedule on itself.

### 5.4.5 Other routines

CurrentTime()

Returns the current value of the simulation clock.
Process *CurrentProcess()

Returns a pointer to the Process whose LifeCycle is currently executing.

void DeleteProcesses()

Deallocates all instances of class Process in existence. This is useful if you have a terminating simulation and you want to return your system to an empty state before starting another repetition.

A process queued for a Resource will be removed from the queue before being deleted. However, any other pointers you have to it will be left dangling, so it is up to you to deal with those.

5.5 Resources

Class Resource is used to represent a finite resource which comes in discrete units, and to coordinate processes which are competing for access to the resource.

5.5.1 Synopsis

Class Resource is defined as follows:

```cpp
#include <akaroa/resource.H>

class Resource {
public:
    Resource(int capacity);
    void Acquire(int amount);
    void Release(int amount);
};
```

5.5.2 Methods

Resource::Resource(int capacity)

The capacity specifies how many units of the resource are initially available.

Resource::Acquire(int amount)

Allocates the specified number of units of the resource to the current process. If the requested amount is not available, the process is blocked until sufficient units become available. Processes waiting for units are allocated them on a first come, first served basis.

Resource::Release(int amount)

Releases the specified number of units of the resource and make them available for other processes.

5.6 AkSimulation: Running an Akaroa simulation from a program

The akrun command is designed primarily for launching an Akaroa simulation manually and visually examining the results. If you want to automate the running of one or more simulations, one way would be to write a shell script which invokes akrun. However, extracting the results from the textual output written by akrun can be tedious.
To make it easier to automatically run an Akaroa simulation and process the results, the class \textit{AkSimulation} is provided. This class allows a C++ program to directly initiate an Akaroa simulation. The results are returned in the form of a structure, which you can then process as desired.

\section*{5.6.1 Synopsis}

Class \textit{AkSimulation} is defined as follows:

\begin{verbatim}
#include <akaroa/simulation.H>

class AkSimulation {

public:

    // Creation and setting up
    AkSimulation(char *command);
    AkSimulation(int argc, char *argv[]);
    void UseHosts(int numHosts);
    void UseHost(char *hostName);
    void SetEnvironmentFile(char *path);
    void SetRandomState(AkRandomState);

    // Running the simulation
    int Run();

    // Getting the results
    int GetNumParams();
    int GetResult(int paramNum, AkResult &);
    AkRandomState GetRandomState();
    char *ErrorMessage();

    // A type used by the routines below
    enum Disposition {Continue, Terminate};

protected:

    // Callback routines
    virtual void EngineStarted(int pid, char *host);
    virtual Disposition RandomOverflow();
    virtual Disposition EngineLost(int pid, char *host);
    virtual Disposition EngineOutput
    (int pid, char *host, char *data, size_t data_length);
};
\end{verbatim}

\section*{5.6.2 Using \textit{AkSimulation}}

To use the \textit{AkSimulation} class, you first create an instance of it, specifying the command name and arguments to use to start the simulation engines. The \textit{AkSimulation} class provides two alternative constructors for this. One takes a single string containing a program name and arguments separated by spaces; the other takes an array of string pointers. If any of your argument strings contain spaces, you will have to use the second form of constructor, because the first one does not interpret quotes or any other special characters.
After creating the AkSimulation, you then specify either how many hosts to use with UseHosts, or particular hosts to use with UseHost. If you are specifying particular hosts, you should make one UseHost call for each host you want to use.

Optionally you may use SetEnvironmentFile or SetRandomState to specify the environment file to use or the initial state of the random number generator.

Then you call Run, which launches the simulation and waits for it to complete. If Run returns 0, the simulation has completed successfully. You can then call GetNumParams to find out how many results are available, and GetResult for each parameter to get the results themselves.

The results are returned in an AkResult structure:

```c
struct AkResult {
    long count; // Total number of observations made
    long trans; // Number of transient observations
    double mean; // Estimate of mean value of parameter
    double variance; // Variance of estimate of mean
    double delta; // Half-width of confidence interval
    double conf; // Confidence level
};
```

After the simulation has been run, you can use GetRandomState to get the final state of the random number generator. This value can be passed to SetRandomState method of the same or another instance of AkSimulation.

The Run method may be called repeatedly to run the simulation multiple times. If this is done, the random number state used for each run will be the one left by the previous run, so in that case it is not necessary to use GetRandomState and SetRandomState.

If Run returns -1, the simulation did not complete successfully for some reason. You can use ErrorMessage to obtain a string explaining the reason for failure. (This method returns a pointer to static storage, so you should copy the string if you’re not going to use it right away.)

The EngineStarted method is called by the system to acknowledge that a simulation engine has been launched. The default implementation of this method does nothing. If you want to take some action on receiving the acknowledgement, create a subclass of AkSimulation and override this method.

The RandomOverflow method is called if the stream of random numbers runs out during the simulation. By default, this method returns the value AkSimulation::Terminate which causes the simulation to be terminated with an appropriate error.

You can override RandomOverflow to perform whatever action you want. If you return AkSimulation::Continue, the simulation will be continued with the random number stream starting again from the beginning.

The EngineLost method is called if contact with a simulation engine is unexpectedly lost. The default method returns AkSimulation::Continue, which causes the simulation to be continued with the remaining engines. If you override this method to return AkSimulation::Terminate, the simulation will be terminated with an appropriate error.

The EngineOutput method is called whenever a simulation engine writes output to its standard error. The default method writes the data to the standard error of the process invoking the simulation (preceded by an identification of the host and process from which the data came) and returns AkSimulation::Continue, which causes the simulation to be continued. If you override this method to return AkSimulation::Terminate, the simulation will be terminated with an appropriate error.

Here is an example which illustrates the use of the AkSimulation class.

/*
5.6. AKSIMULATION

* run_uni2.C - Simple example illustrating the use of the
* ========== AkSimulation class
*
#include <stdio.h>
#include <akaroa.H>
#include <akaroa/simulation.H>

int main(int argc, char *argv[]) {
    AkSimulation *sim = new AkSimulation("uni2");
sim->UseHosts(3);
    if (sim->Run() == 0) {
        int n = sim->GetNumParams();
        for (int i = 1; i <= n; i++) {
            AkResult result;
            sim->GetResult(i, result);
            printf("Parameter %d: Mean = %lg +/- %lg\n", i, result.mean, result.delta);
        }
    } else
        printf("It didn’t work! %s\n", sim->ErrorMessage());
}
Chapter 6

Examples

This chapter contains some examples of complete simulation engines, illustrating the use of the core Akaroa routines and many of the library routines and classes.

6.1 An M/M/1 Queueing System

This example models a simple M/M/1 queueing system, illustrating the use of the Process Manager and the Resource class. You will see that it is just an ordinary simulation program, with the addition of a call to AkObservation at the point where the service time is calculated.

```cpp
/*
 * mm1.C - M/M/1 Queueing System
 * =====
 */

#include "akaroa.H"
#include "akaroa/distributions.H"
#include "akaroa/process.H"
#include "akaroa/resource.H"

double arrival_rate; // Rate at which customers arrive
double service_rate; // Rate at which customers are served

// There is one server, modelled here as a Resource
// with a capacity of 1 unit.

Resource server(1);

// Each customer is modelled as a process. A customer’s
// life consists of arriving, waiting for the server to become
// available, waiting to be served, and leaving.
// We calculate the time between entering and leaving,
// and hand it to Akaroa as an observation.
//
// This is not a very efficient implementation, but it serves
// to illustrate how to use Processes and Resources.

class Customer : public Process {
    public:
```
void LifeCycle();

};

void Customer::LifeCycle() {
    Time arrival_time, service_time;
    arrival_time = CurrentTime();
    server.Acquire(1);
    Hold(Exponential(1/service_rate));
    server.Release(1);
    service_time = CurrentTime() - arrival_time;
    AkObservation(service_time);
}

// The main program. After getting the load from the command
// line and calculating the arrival and service rates,
// we enter a loop generating new customers at the arrival
// rate.

int main(int argc, char *argv[]) {
    real load = atof(argv[1]);
    service_rate = 10.0;
    arrival_rate = load * service_rate;
    for (;;) {
        new Customer;
        Hold(Exponential(1/arrival_rate));
    }
}

6.2 A Multiprocessing Computer System

This example models a multiprocessing computer system consisting of one CPU, some
number of disks, and some number of terminals. It illustrates the use of the Process and
Resource classes, and how they can be used to model a closed system (one with no sources
or sinks).

At each terminal, a user interactively submits requests and waits for the results. Ob-
servations are made of the response times of the requests - i.e. the time between the user
making the request and the system finishing processing of the request.

Each user is modelled as a Process, and the CPU and disks are modelled as Resources.
The life cycle of a user consists of thinking for some random time and then making a
request. The request uses the CPU for a random time, then has some probability of either
using one of the disks for a random time and returning to use the CPU again, or of finishing.
The user then goes back to the think state and the life cycle repeats.

In this example, all of the random times are exponentially distributed.

/*
 * multi.C - Simulation of a timesharing computer system
 * ========
 */

#include "akaroa.H"
#include "akaroa/distributions.H"
#include "akaroa/process.H"
#include "akaroa/resource.H"

int num_users = 5; // Number of terminals/users
int num_disks = 1; // Number of disk drives
real mean_CPU_time = 20; // Mean burst of CPU usage
real mean_disk_time = 4; // Mean disk usage time
real mean_think_time = 100; // Mean time a user spends thinking
real use_disk_probability = 0.25; // Probability of using disk

class User : public Process {
  public:
    User() : Process(1024) {};
    virtual void LifeCycle();

  User **users;
  Resource *cpu;
  Resource **disks;

  void User::LifeCycle() {
    for (;;) {
      Time start = CurrentTime();
      cpu->Acquire(1);
      Hold(Exponential(mean_CPU_time));
      cpu->Release(1);
      if (Uniform(0, 1) <= use_disk_probability) {
        int i = UniformInt(0, num_disks - 1);
        disks[i]->Acquire(1);
        Hold(Exponential(mean_disk_time));
        disks[i]->Release(1);
      } else {
        AkObservation(CurrentTime() - start);
        Hold(Exponential(mean_think_time));
      }
    }
  }

  int main(int argc, char *argv[]) {
    users = new User*[num_users];
    for (int i = 0; i < num_users; i++)
      users[i] = new User();
    cpu = new Resource(1);
    disks = new Resource*[num_disks];
    for (i = 0; i < num_disks; i++)
      disks[i] = new Resource(1);
    Hold();
  }

6.3 A Terminating Simulation

This is an example of a simulation which produces independent observations. An M/M/1 queueing system is run for the first 25 customers and the mean delay of these customers is
submitted to Akaroa as an observation. The simulation is repeated to generate a series of observations, which are analysed using independent observation mode.

/*
 * mm1term.C - Terminating M/M/1 Simulation
 * =========
 * Example of a simulation which produces independent observations. Repeatedly runs an M/M/1 queueing system starting from empty and idle, and observes the mean delay of the first 25 customers.
 */

#include <stdlib.h>
#include <iostream.h>
#include "akaroa.H"
#include "akaroa/distributions.H"
#include "akaroa/process.H"
#include "akaroa/resource.H"

int customersRequired = 25;

double arrival_rate; // Rate at which customers arrive
double service_rate; // Rate at which customers are served

Resource *server; // The server

int customersServed; // For calculating mean real totalDelay; // delay of customers

//
// Process class modelling a customer
//

class Customer : public Process {
public:
    void LifeCycle();
};

void Customer::LifeCycle() {
    Time arrival_time, begin_service_time, delay;
    arrival_time = CurrentTime();
    server->Acquire(1);
    begin_service_time = CurrentTime();
    Time service_time = Exponential(1/service_rate);
    Hold(service_time);
    server->Release(1);
    delay = begin_service_time - arrival_time;
    ++customersServed;
    totalDelay += delay;
}

//
// Perform one repetition of the simulation.
Loop generating new customers until the required number of customers have been served. Then calculate the mean delay, give it to Akaroa as an observation, and clean out the system ready for the next repetition.

Note that we create a fresh server for each repetition to ensure that it starts out with the correct initial state.

```cpp
void RunOnce() {
    customersServed = 0;
    totalDelay = 0;
    server = new Resource(1);
    while (customersServed < customersRequired) {
        new Customer;
        Hold(Exponential(1/arrival_rate));
    }
    real meanDelay = totalDelay / customersServed;
    AkObservation(meanDelay);
    DeleteProcesses();
    delete server;
}
```

The main program. After getting the load from the command line and calculating the arrival and service rates, we inform Akaroa that the observations will be independent, then enter a loop repeating the simulation forever.

```cpp
int main(int argc, char *argv[]) {
    real load = atof(argv[1]);
    service_rate = 10.0;
    arrival_rate = load * service_rate;
    AkObservationType(AkIndependent);
    for (;;) {
        RunOnce();
    }
}
Appendix A

Obsolete Facilities

This chapter describes parts of Akaroa II and its libraries which are obsolete. They are provided only to support simulation programs written to run under previous versions of Akaroa. You should not use any of the facilities described here in new simulation programs, since they may disappear from future versions of Akaroa II.

A.1 Event Manager

The functions of the Event Manager have been taken over by the Process Manager. You should use either the Process Manager or the Event Manager, but not both.

The Event Manager maintains a queue of events, each of which is scheduled to occur at a specified simulation time. When an event occurs, it executes a piece of code which you supply. This code can perform whatever action you want, including scheduling further events.

To use the Event Manager, you write a procedure for each event which can occur in your simulation. Each event procedure should take one argument, which must be a pointer, although it can point to whatever type of data is appropriate, and different event procedures can take pointers of different types.

You start the simulation off by calling Schedule to schedule one or more events as described below. Then you enter a loop calling NextEvent repeatedly. Each time you call NextEvent, the earliest event in the event queue is extracted, the simulation clock is advanced to the time for which it is scheduled, and its associated procedure is called with the specified argument.

Typically, your action procedures will schedule further events, which will schedule further events again, and so forth, thus keeping the simulation going. You should also call AkSimulationOver periodically in your main loop, so that you can tell when to stop.

A.1.1 Event Manager Routines

The Event Manager defines the following types and routines.

```
#include <akaroa/events.H>

typedef real Time;

Values of type Time are used by the Event Manager to represent simulation times.
The unit in which simulation time is measured is up to the user's interpretation.

template <class T>
void Schedule(void (*proc)(T *), T *argument, Time delay);
```
Schedules the procedure \texttt{proc} to be called with the given argument at the current simulation time plus \texttt{delay}. For example,

\begin{verbatim}
Pentium *p = new Pentium;
Schedule(Explode, p, 42);
\end{verbatim}

schedules an event to occur 42 time units from now. When the simulation clock reaches that time, \texttt{Explode} will be called with \texttt{p} as argument (both of which the user has presumably defined in some appropriate way).

\begin{verbatim}
int NextEvent()
    If there are any events in the event queue, the one scheduled to occur next is removed from the queue, its action procedure is called with the argument specified when the event was scheduled, and true is returned. If the event queue is empty, false is returned.

    Typically, NextEvent will be called from the main loop of your simulation, which will look something like this:\footnote{This example assumes the simulation to be designed so that the event queue can never become empty. In a steady-state simulation, this will usually be the case. If there is a chance that the event queue could become empty, you should test the return value from NextEvent, and if it is false, do something that will schedule one or more events.}
\end{verbatim}

\begin{verbatim}
    while (!AkSimulationOver())
        NextEvent();
\end{verbatim}

\begin{verbatim}
Time CurrentTime()
    Returns the current value of the simulation clock.
\end{verbatim}
Appendix B

Adding an analysis method to Akaroa

B.1 Introduction

Akaroa comes with two methods for analysing observations: Batch Means and Spectral Analysis. If neither of these methods suits your needs, and you have another way in which you want to analyse your observations, you can implement your own analysis method and add it to the Akaroa library. This appendix describes how to do this.

Note: The information presented here depends on the internal structure of the Akaroa library, and is likely to change in future versions of Akaroa.

B.2 What an analysis method does

The job of an analysis method is to take a stream of observations and calculate two things from it: (1) an estimate $\hat{\mu}$ of the mean value $\mu$ of the parameter; (2) an estimate $\hat{\sigma}^2$ of the variance of $\hat{\mu}$.

Typically, an analysis method operates in two phases:

1. the transient phase, in which observations from the beginning of the simulation run are discarded, until the analysis method determines (by some means) that the simulation has reached steady state. The analysis method then enters:

2. the analysis phase, in which observations are collected and used to calculate values for $\hat{\mu}$ and $\hat{\sigma}^2$.

Not all analysis methods will have a transient phase; only those (such as Batch Means and Spectral Analysis) which require the system to be in steady state before beginning analysis. For example, analysis of independent observations in Akaroa is done using a third analysis method which does not have a transient phase.

B.2.1 Checkpoints

Although the analysis method could calculate a new estimate of $\hat{\mu}$ and $\hat{\sigma}^2$ after every observation, to do so would be very inefficient. Therefore, the analysis method will usually collect some number of observations before calculating a new set of estimates.

The point at which new estimates are calculated is called a checkpoint, and the spacing between checkpoints (the number of observations collected before a checkpoint is reached) is under the control of the analysis method. Some methods will have natural places to use
as checkpoints – in Batch Means, for instance, a checkpoint corresponds to a batch or some number of batches. In other methods – such as Spectral Analysis – checkpoint spacing can be arbitrary.

If your analysis method allows freedom in the spacing of checkpoints, you may wish to base it on the value of an Akaroa environment variable so that it is under the control of the user (see B.4).

### B.3 Implementing an analysis method

Before starting, you should make your own copy of the Akaroa source. The easiest way is to unpack the distributed .tar file in a directory of your own. In what follows, this directory will be referred to as $MYAK.

*Note:* Don’t use `cp` to copy the Akaroa source directory. It contains symbolic links, which will be messed up by `cp`.

You should update your `PATH` variable to look for the Akaroa binaries (`akmaster`, `akslave` and `akrun`) in `$MYAK/bin`.

#### B.3.1 Steps to implementing an analysis method

Implementing an analysis method and adding it to Akaroa requires the following steps:

1. Write a new subclass of class `ParameterAnalyser` which implements your analysis method.
2. Declare your analysis method to Akaroa by including a call to the macro `DefineParameterAnalyserType`.
3. Add the name of your analysis method to the list of possible values for the `AnalysisMethod` variable in the Akaroa environment. Optionally, you can also add new Akaroa environment variables for controlling your analysis method.
4. Add the name of your object file to the Akaroa Makefile and recompile Akaroa.

Each of these steps is described in detail below.

#### B.3.2 Class `ParameterAnalyser`

A `ParameterAnalyser` performs observation analysis for a single parameter. Akaroa will create an instance of your parameter analyser for each parameter to be analysed.

You will need to include the following header files:

```c
#include "parameter_analyser.H"
#include "environment.H"
#include "checkpoint.H"
```

The interface to class `ParameterAnalyser` is declared in `$MYAK/src/engine/parameter_analyser.H`. The relevant parts of this declaration are as follows.

```c
class ParameterAnalyser {
public:
    ParameterAnalyser(int paramNum, Environment *);
    virtual void ProcessObservation(real value) = 0;
    virtual boolean ReachedCheckpoint() = 0;
    virtual void GetCheckpoint(Checkpoint &cp) = 0;
}
```
B.3. IMPLEMENTING AN ANALYSIS METHOD

Your analyser class must have a constructor which takes the same arguments as the ParameterAnalyser constructor, and passes them on to that constructor. For instance, if your class is called MyAnalyser, your constructor should look like this:

```cpp
MyAnalyser::MyAnalyser(int n, Environment *e)
    : ParameterAnalyser(n, e)
{
    // initialise your analyser here
}
```

Your analyser should implement the following three methods:

```cpp
void ProcessObservation(real value)

Akaroa will call this method each time an observation for this parameter is submitted by the simulation engine.

boolean ReachedCheckpoint()

Akaroa will call this method periodically to find out whether your analyser has reached a checkpoint (i.e. it has collected enough observations since the last checkpoint to calculate an estimate of the mean and variance). If your analyser has reached a checkpoint, it should return true, otherwise false.

void GetCheckpoint(Checkpoint &cp)

When your ReachedCheckpoint method returns true, Akaroa will then call this method. Your analyser should calculate and fill in the following fields of the Checkpoint structure:

- `cp.count`
  Total number of observations submitted, in both the transient phase (if any) and the analysis phase.

- `cp.trans`
  Number of observations discarded during the transient phase, if any.

- `cp.mean`
  Estimate of $\mu$.

- `cp.variance`
  Estimate of $\sigma^2(\hat{\mu})$.

Optionally, you can set the value of `cp.df`. Akaroa sets this to zero before calling GetCheckpoint; if you leave it zero, Akaroa uses the normal distribution to calculate the confidence interval of $\hat{\mu}$ from $\hat{\sigma}^2(\hat{\mu})$. If you set `cp.df` to a non-zero value $n$, Akaroa uses a $t$-distribution with $n$ degrees of freedom.
B.3.3 Declaring your analyser to Akaroa

To make your analyser known to Akaroa, you must place a call to the following macro at the top of your source file:

```
DefineParameterAnalyserType("name", class)
```

where `name` is the name by which your analysis method is to be known to the user, and `class` is the name of the class implementing your method. For example,

```
DefineParameterAnalyserType("MyMethod", MyAnalyser)
```

B.3.4 Adding a value for AnalysisMethod

You also have to add `name` to the list of valid values for the AnalysisMethod variable (otherwise the user will get an error when he tries to use it). To do this, you need to edit the file `$MYAK/src/env/variables.C`. Find the part which contains:

```
"AnalysisMethod", "e", "Spectral", "Spectral", "BatchMeans", ".Independent", 0,
```

and add the name of your method (the `name` string that you used in the DefineParameterAnalyserType call) to the list at the end, before the final zero. For example:

```
```

B.3.5 Adding your code to the Makefile

Add the name of the object file (or files) implementing your analyser to the definition of AKANAL_OBJ in `$MYAK/src/Makefile.common`, for example:

```
AKANAL_OBJ = \n   $HOME/mystuff/my_analyser.o \n   ...
```

The pathname you use in the Makefile must either be a full pathname or relative to the `$MYAK/src` directory. The source file corresponding to the `.o` file should end in `.C` so that the Makefile will be able to find it.

B.3.6 Recompiling Akaroa

To recompile Akaroa, change directory to `$MYAK/src` and issue the following shell command:

```
make system
```

This will compile the Akaroa library and the programs `akmaster`, `akslave` and `akrun`, and make them available in the `$MYAK/lib` and `$MYAK/bin` directories.

You will also need to recompile any simulation engines that you want to use with the new method. To recompile one of the example simulations, e.g. `mm1`, use a command such as

```
make mm1
```

If you compile a simulation engine of your own, make sure that you link it with your new version of the Akaroa library (the one in `$MYAK/lib`).
B.4 Accessing the Akaroa Environment

If desired, your analyser can use the values of Akaroa environment variables. For example, you might want to use the value of the CPSpacingFactor variable as a basis for the checkpoint spacing. You can also define new environment variables of your own.

B.4.1 Retrieving Akaroa environment variables

Values of Akaroa environment variables are retrieved using the Environment * pointer passed to the constructor of the parameter analyser. This points to an Environment object which has the following methods:

- intGetInt(char *name)
- realGetReal(char *name)
- char *GetString(char *name)

These retrieve the values of integer, real and string valued variables, respectively.

There is also a fourth type of variable, enumerated, whose value is one of a set of named values (like the AnalysisMethod variable). There are two methods for retrieving the value of an enumerated variable:

- void GetEnum(char *name, char *&value)
- void GetEnum(char *name, int &value)

The first one returns the value as a string, and the second one returns it as an ordinal number (starting with 0).

Here is a partial example of a parameter analyser which retrieves the value of two existing Akaroa environment variables, CPSpacingFactor and CPSpacingMethod, and stores them for later use.

```cpp
class MyAnalyser : public ParameterAnalyser {
public:
    MyAnalyser(int n, Environment *env);
    ...
private:
    real cpsf;
    int cpm;
    ...
};

MyAnalyser::MyAnalyser(int n, Environment *env)
    : ParameterAnalyser(n, env)
{
    cpsf = env->GetReal("CPSpacingFactor");
    GetEnum("CPSpacingMethod", cpm); // 0 = Linear, 1 = Geometric
    ...
}
```

B.4.2 Defining new Akaroa environment variables

To add a new Akaroa environment variable, you need to add a row to the table in $MYAK/src/env/variables.C. The table has four columns: the name of the variable, its type, its default value, and (for enumerated variables only) a list of all the possible values.

Here are four example table entries, defining a variable of each of the four types:

```*/

/*Name*/ /*Type*/ /*Default*/ /*Values*/
"MyInteger", "i", "42",
```
<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>3.1415</th>
</tr>
</thead>
<tbody>
<tr>
<td>MyString</td>
<td>s</td>
<td>strawberry</td>
</tr>
<tr>
<td>MyEnum</td>
<td>e</td>
<td>Honda, Honda, Suzuki, Yamaha, 0,</td>
</tr>
</tbody>
</table>
Bibliography


NAME
akadd – add simulation engines to an Akaroa simulation

SYNOPSIS

akadd [-d] -s sid -n neng
akadd [-d] -s sid -H host...

OPTIONS
-d  Turn on debugging. With this option, akadd writes a trace of all messages sent to or received from the akmaster process.
-s sid  Specifies the simulation ID of the simulation to which engines are to be added.
-n neng Specifies the number of simulation engines to add.
-H host... Specifies a list of hosts on which to start new simulation engines.

DESCRIPTION
Akadd is used to add new simulation engines to an already running simulation. With -n, the given number of engines are started on hosts arbitrarily chosen from those available. With -H, one new engine is started on each of the specified hosts.

The -s flag identifies the simulation to which engines are to be added by means of the simulation ID reported by akrun when the simulation was started.

SEE ALSO
akrun(1)

Akaroa II User’s Manual

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
akmaster – Akaroa master process

SYNOPSIS

akmaster [-d]

OPTIONS
-d  Turn on debugging. With this option, akmaster will write a trace of all messages sent to or received from other processes, together with other information.

DESCRIPTION
Akmaster is the master process which coordinates all other processes in the Akaroa system. There must be one akmaster process running under your userid on some host before you can start any other Akaroa processes.

When you are finished with Akaroa, you should terminate the akmaster process using an interrupt or terminate signal (control-C, or kill with no arguments) so that it can clean up temporary files.

When an akmaster process terminates (either normally or abnormally), any connected akslave or akrun processes terminate immediately, and any connected simulation engines terminate at the next checkpoint.

FILES

`~/.akmaster` This file is created in the user's home directory to hold the host name and port number of the current akmaster process, so that other Akaroa processes can contact it. If an akmaster process terminates abnormally, you may need to remove this file before another akmaster can be started.

SEE ALSO
akslave(1), akrun(1)

_Akaroa II User’s Manual_

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
akrun – run an Akaroa simulation

SYNOPSIS

 [-n num-hosts] [-H host-name]...
 [-P precision] [-C confidence]
 [-D variable=value]...
 [--] command [arg...]

OPTIONS
-d
Turn on debugging. With this option, akrun writes a trace of all messages sent to or received from the akmaster process.

-f env-file
Obtain Akaroa Environment settings from the specified file. Without this option, the file Akaroa in the current directory is used if present, otherwise the built-in defaults are used.

-r
Report the state of the random number generator at the end of the run.

-s state
Initialise the random number generator to the given state before the run.

-R nreps
Repeat the simulation nreps times with different random number streams. For each parameter, report the result from the run which produced the greatest number of observations.

-A
With -R, report the results from all repetitions. Otherwise, only the final results chosen are reported.

-n num-hosts
Run simulation engines on the given number of hosts.

-H host-name
Run a simulation engine on the given host. One -H option must be given for each host to be used.

-P precision
Specifies the required relative precision of the results. The argument must be a number between 0 and 1.

-C confidence
Specifies the required confidence level of the results. The argument must be a number between 0 and 1.

-D variable=value
Specifies the value of an Akaroa environment variable. In the case of a string-valued variable, the value must be enclosed in double quotes (which must be protected from interpretation by the shell).

If the arguments to the simulation program contain options beginning with a hyphen, a double hyphen is required before the simulation command to separate the akrun options from the simulation program options.
**command arg...**

The command to execute in order to start each simulation engine. If the command is not a full pathname, it will be searched for by each akslave process along the search path that was in effect when the akslave was started.

**DESCRIPTION**

Akrun is the command by which the user runs a simulation under Akaroa. Before using akrun, an akmaster process must be running, together with an akslave process on each host where it is desired to run a simulation engine.

If the -n option is given with no -H options, the akmaster arbitrarily chooses the specified number of hosts from among those running akslaves, and launches a simulation engine on each one. If -H options are given, simulation engines are launched on the specified hosts. If both -n and -H options are given, the number of hosts specified with -n must equal the number of -H options.

Each simulation engine calculates a local estimate of each observed parameter, and periodically reports these estimates to the akmaster. The akmaster calculates a global estimate of each parameter and keeps track of its precision and confidence.

When all parameters have reached the required precision with the required confidence, the simulation engines are terminated.

To guard against unreliable results from spuriously short runs, the entire simulation may be repeated a number of times using the -R option. For each parameter, the result reported is the one from the run which produced the greatest number of observations for that parameter.

**Output**

The report consists of a headline followed by one line for each observed parameter. Each line contains the following fields:

- **Min, Max**: Lower and upper bounds of the estimate of the mean value of the parameter. The true mean lies between Min and Max with probability $p$, where $p$ is the confidence level specified for the parameter.
- **Variance**: Variance of the estimate of the mean.
- **Count**: The total number of observations made by all engines.
- **Trans**: The total number of observations discarded during the transient phases of all replications (before the system settled down into steady state).

**The Akaroa Environment**

The Akaroa Environment is a collection of variables which affect the operation of the Akaroa system. The settings of these variables are read from a file when a simulation is started. If a file is specified with the -f option, that file is used. If no -f option is specified and there is a file called Akaroa in the current directory, that file is used. Otherwise, built-in defaults are used for all settings.

(Note: The Akaroa Environment is separate from the Unix environment. You cannot set an Akaroa Environment variable using setenv).

The environment file has the following syntax:
Local settings apply only to the specified parameter. Global settings apply to all parameters which do not specify a local setting for that variable. If no local or global setting is specified for a variable, the built-in default is used.

Values for environment variables can also be specified using the -P, -C and -D command line options. Such values override any global settings for the same variables in an environment file, but not local settings.

The following variables may be set:

- **Precision**
  Relative precision required. Default is 0.05.

- **Confidence**
  Confidence level required. Default is 0.95 (i.e. 95 per cent).

- **AnalysisMethod**
  Method used to calculate the variance of the estimate in order to determine its precision and confidence. Valid values are: **Spectral** - use the method of spectral analysis [HEID81]. **BatchMeans** - use the method of batch means [PAWL90].

The following variables apply to the Spectral analysis method:

- **CPSpacingMethod**
  Method used to determine spacing between checkpoints. Valid values are **Linear** or **Geometric**. Default is **Linear**.

- **CPSpacingFactor**
  For **Linear** checkpoint spacing, this is the distance between successive checkpoints as a multiple of the length of the transient period. For **Geometric** checkpoint spacing, this is the factor by which the checkpoint spacing is increased after each checkpoint. Default is 1.5.

- **PeriodogramPoints**
  For the **Spectral** analysis method, this is the number of points used from the periodogram (see HEID81). Default is 25.

- **PolynomialDegree**
  For the **Spectral** analysis method, this is the degree of the polynomial fitted to the periodogram (see HEID81). Default is 2.

The following variables apply to the BatchMeans analysis method:

- **InitBatchSize**
  Initial batch size. The final batch size chosen will be a multiple of this size.

- **AnalysedSeqLen**
  Length of the sequence of batch means tested for autocorrelation during the batch size selection phase.

- **AutoCorrSignif**
  Significance level below which autocorrelation between batch means is considered small enough to accept the batch size being tested.

Version: 1
Last change: 14 February 1997
Random Numbers
Akaroa coordinates the random numbers received by each simulation engine to ensure that each replication of the simulation receives a different sequence of random numbers, independent of the sequences received by the other replications.

With the -r option, akrun reports the state of the random number generator at the end of the run with a line of the form:

\texttt{RandomState: \%d:%d}

where the first number specifies one of the coefficients listed in AkRandom and the second number specifies a position in the sequence generated by that coefficient.

The -s option may be used to set the initial state of the random number generator by giving it an argument of the form. By feeding the result of the -r option from one run to the -s option of a subsequent run, a simulation can be run multiple times with a different random number sequence each time.

FILES
Akaroa
Default file from which to obtain the Akaroa Environment settings.

SEE ALSO
akmaster(1), akslave(1), AkRandom(3)

Akaroa II User’s Manual


AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
    akslave - Akaroa simulation engine launcher

SYNOPSIS

    akslave [-d]

OPTIONS
    -d  Turn on debugging. With this option, akslave will write a trace of all messages sent to or
        received from the akmaster process.

DESCRIPTION
    Akslave is the part of the Akaroa system responsible for launching simulation engines on a particular
    host. To use a host to run simulation engines, there must be one akslave process running on that
    host. The akslave will automatically locate and contact the akmaster when started.

    The search path in effect when an akslave is started should include the directories containing any
    simulation engines that you will desire to run. Otherwise, it will be necessary to specify the full
    pathname of the simulation engine to akrun when initiating the simulation.

    Once an akslave process has started up properly, it closes its standard input, output and error. If
    the akslave is being started using rsh, this will cause the rsh command to terminate. Therefore,
    if the rsh command completes with no error messages, the akslave has started successfully and is
    running in the background on the remote host.

    As long as an akmaster process exists, an akslave can be started on a new host at any time, and
    that host will be available to any subsequently initiated simulations. An akslave process may be
    killed to make its host unavailable for subsequent simulations (but killing an akslave on a host where
    simulation engines are running is not recommended).

FILES
    ~/.akmaster
        Used by akslave to locate the user’s akmaster process.

SEE ALSO
    akmaster(1), akrun(1)

    Akaroa II User’s Manual

AUTHOR
    Gregory C. Ewing, University of Canterbury
NAME
akstat – enquire status of Akaroa system

SYNOPSIS

    akstat [-d] [-HSEGL] [-s sid] [-e eng] [-p param]

OPTIONS

    -d       Turn on debugging. With this option, akstat writes a trace of all messages sent to or received from the akmaster process.
    -H       Show list of hosts running akslave processes.
    -S       Show information about currently running simulations. With -s, shows the specified simulation, otherwise shows all simulations.
    -E       Show information about currently running simulation engines. With -s and/or -e, restrict the list to the specified simulation and/or engine ID.
    -G       Show information about global estimates. With -s and/or -p, show global estimates for the specified simulation and/or parameter only.
    -L       Show information about local estimates. With -s, -e and/or -p, show local estimates for the specified simulation, engine ID and/or parameter only.
    -s sid   Restrict listing to the specified simulation ID.
    -e eng   Restrict listing to the specified engine ID.
    -p param Restrict listing to the specified parameter number.

DESCRIPTION

The akstat command displays information about the status of the Akaroa system. There are two kinds of options to akstat: upper case options specify what kinds of information to display, and lower case options specify which simulation, engine or parameter to display information about.

Options may be combined in any meaningful way. Combining upper case options produces all of the requested listings. Combining lower case options produces the logical "and" of the individual restrictions.

Akstat with no arguments is equivalent to akstat -HS.

OUTPUT

Host List

The -H option produces a list with the following columns:

    SLAVE     Slave number.
    HOST      Name of the host on which the akslave process is running.
PID Process ID of the akslave process.

ENGINES Number of simulation engines running on the host on behalf of this akmaster process.

**Simulation List**
The -S option produces a list with the following columns:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>Simulation ID.</td>
</tr>
<tr>
<td>PARMS</td>
<td>Number of parameters being estimated.</td>
</tr>
<tr>
<td>ENGS</td>
<td>Number of simulation engines belonging to the simulation.</td>
</tr>
<tr>
<td>RANDOM</td>
<td>State of the random number generator.</td>
</tr>
<tr>
<td>FLAGS</td>
<td>An &quot;O&quot; in this column indicates that the simulation is over.</td>
</tr>
<tr>
<td>COMMAND</td>
<td>Command and arguments used to start the simulation engines.</td>
</tr>
</tbody>
</table>

**Engine List**
The -E option produces a list with the following columns:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>Simulation ID.</td>
</tr>
<tr>
<td>EID</td>
<td>Engine ID.</td>
</tr>
<tr>
<td>HOST</td>
<td>Name of the host on which the engine is running.</td>
</tr>
<tr>
<td>PID</td>
<td>Process ID of the simulation engine.</td>
</tr>
<tr>
<td>FLAGS</td>
<td>An &quot;R&quot; in this column indicates that the engine has reported a local estimate and is waiting for a reply from the akmaster process.</td>
</tr>
<tr>
<td>STATE</td>
<td>This column may contain one of: &quot;launching&quot; - a launch request has been sent to the akslave and the akmaster is waiting for the engine to connect; &quot;running&quot; - the engine is connected and running; &quot;dead&quot; - connection to the engine has been lost.</td>
</tr>
</tbody>
</table>

**Global Estimate List**
The -G option produces a list with the following columns:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SID</td>
<td>Simulation ID.</td>
</tr>
<tr>
<td>PAR</td>
<td>Parameter number.</td>
</tr>
<tr>
<td>MEAN</td>
<td>Current global estimate of the mean.</td>
</tr>
<tr>
<td>PREC</td>
<td>Relative precision of the global estimate.</td>
</tr>
<tr>
<td>VARIANCE</td>
<td>Variance of the global estimate.</td>
</tr>
<tr>
<td>OBS</td>
<td>Total number of observations made of this parameter by all engines.</td>
</tr>
<tr>
<td>TRANS</td>
<td>Number of observations made of this parameter during the transient phase by all engines.</td>
</tr>
</tbody>
</table>
Local Estimate List
The -L option produces a list with the following columns:

- **SID** Simulation ID.
- **EID** Engine ID.
- **PAR** Parameter number.
- **MEAN** Current local estimate of the mean.
- **VARIANCE** Variance of the local estimate.
- **OBS** Number of observations made of this parameter by this engine.
- **TRANS** Number of observations made of this parameter during the transient phase by this engine.
- **CHKPTS** Number of checkpoints (local estimates) received for this parameter from this engine.
- **CP/MIN** Average number of checkpoints per minute received for this parameter from this engine over the last 10 minutes.
- **LAST CHKPT** Time and date at which the last checkpoint was received for this parameter from this engine.

SEE ALSO
Akaroa II User’s Manual

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME

AkMessage – Send informative message from simulation engine to user

SYNOPSIS

```c
#include <akaroa/ak_message.h>

void AkMessage(char *format...);
```

DESCRIPTION

`AkMessage` can be used by a simulation program to report information to the user who initiated the simulation. It accepts `printf`-style arguments and formats them into a string.

If the simulation program is running stand-alone, the string is written to standard error followed by a newline.

If the simulation program is participating in a simulation started by `akrun`, the string is written to the standard error of the akrun process, preceded by an identification of the process from which it came.

If the simulation was started by an `AkSimulation` object in a user program, that object’s `Engine-Output` method is called with the string as an argument.

Note: Some earlier versions of Akaroa relayed text written to the standard error of a simulation engine back to the initiating process. This is no longer supported; the AkMessage routine should now be used instead.

SEE ALSO

`printf(3S), akrun(1), AkSimulation(3)`

`Akaroa II User’s Manual`

AUTHOR

Gregory C. Ewing, University of Canterbury
NAME
AkRandom – Akaroa random number source

SYNOPSIS

```
#include <akaroa.H>

unsigned long AkRandom();
```

DESCRIPTION

AkRandom is the basic source of random numbers used by simulation engines. The Akaroa system coordinates the random number streams received by different replications of a simulation being run in parallel, to ensure that each replication receives an independent random number sequence.

Simulation engines should always obtain random numbers using AkRandom or a routine based on it. All of the routines described in AkDistribution(3) are based on AkRandom.

RETURN VALUE

Each time it is called, AkRandom returns a random integer $n$ such that $1 \leq n < 2^{31} - 1$.

ALGORITHM

AkRandom uses a multiplicative linear congruential random number generator. The random number sequence consists of the concatenation of several sequences of length $2^{31} - 2$ with different multiplying coefficients. This sequence is partitioned among the different replications of a parallel run, so that the length of the sequence available to one replication is reduced in proportion to the number of replications.

The coefficients currently used by AkRandom are listed below. They are taken from a list of optimal full-period coefficients published by Fishman and Moore [FISH86]. The ones marked * have been recommended by those authors as being of particularly high quality.

<table>
<thead>
<tr>
<th>No.</th>
<th>Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>742938285*</td>
</tr>
<tr>
<td>1</td>
<td>950706376*</td>
</tr>
<tr>
<td>2</td>
<td>1226874159*</td>
</tr>
<tr>
<td>3</td>
<td>6208991*</td>
</tr>
<tr>
<td>4</td>
<td>1343714438*</td>
</tr>
<tr>
<td>5</td>
<td>2049513912</td>
</tr>
<tr>
<td>6</td>
<td>781259587</td>
</tr>
<tr>
<td>7</td>
<td>482920380</td>
</tr>
<tr>
<td>8</td>
<td>1810831696</td>
</tr>
<tr>
<td>9</td>
<td>502005751</td>
</tr>
<tr>
<td>10</td>
<td>464822633</td>
</tr>
<tr>
<td>11</td>
<td>1980989888</td>
</tr>
<tr>
<td>12</td>
<td>329440414</td>
</tr>
<tr>
<td>13</td>
<td>1930251322</td>
</tr>
<tr>
<td>14</td>
<td>800218253</td>
</tr>
<tr>
<td>15</td>
<td>1575965843</td>
</tr>
<tr>
<td>16</td>
<td>1100494401</td>
</tr>
<tr>
<td>17</td>
<td>1647274979</td>
</tr>
</tbody>
</table>

Version: 1                        Last change: 14 February 1997 1
The list of coefficients may be extended by editing the file `src/stats/random_generator.C` in the Akaroa source and recompiling the system. To ensure quality it is recommended that additional numbers also be taken from the list in [FISH86].

**FILES**

`<akaroa>/ src/stats/random_generator.C`

Source file containing the list of coefficients, for extending the random number sequence.

**SEE ALSO**

AkDdistribution(3)


Akaroa II User’s Manual
NAME
AkSimulation – Akaroa simulation launching class

SYNOPSIS

#include <akaroa/simulation.h>

class AkSimulation {
public:
    AkSimulation(char *command);
    AkSimulation(int argc, char *argv[]);
    void UseHosts(int n);
    void UseHost(char *host);
    void SetEnvironmentFile(char *path);
    void SetRandomState(AkRandomState state);
    void SetPrecision(float);
    void SetConfidence(float);
    int SetEnvInt(char *name, int value);
    int SetEnvReal(char *name, double value);
    int SetEnvString(char *name, char *value);
    int SetEnvEnum(char *name, char *value);
    int Run();
    int GetNumParams();
    int GetResult(int i, AkResult &result);
    AkRandomState GetRandomState();
    char *ErrorMessage();
    enum Disposition {Continue, Terminate};
protected:
    virtual void EngineStarted(int pid, char *host);
    virtual Disposition RandomOverflow();
    virtual Disposition EngineLost(int pid, char *host);
    virtual Disposition EngineOutput
        (char *host, int pid, char *data, size_t 0);
};

struct AkResult {
    long count;
    long trans;
    double mean;
    double variance;
    double delta;
};

DESCRIPTION
Class AkSimulation provides the means for a program to initiate a simulation under Akaroa, wait
for it to complete and collect the results.

The program to be run as a simulation engine and the arguments to be passed to it are specified
when constructing an instance of AkSimulation.
The first constructor accepts a string containing a program name and arguments separated by spaces. No interpretation of quotes or other special characters is done on this string, so the argument strings cannot contain embedded spaces.

The second constructor accepts a count argc and a vector of strings argv, where argv[0] is the program name and argv[1]... are the arguments. The value of argc should be the number of strings in argv. The strings may contain any characters, including spaces. These strings will appear unchanged as the argv parameters of the simulation program.

UseHosts specifies that simulation engines are to be run on n hosts arbitrarily chosen from those available. Alternatively, particular hosts may be specified by calling UseHost once for each host which is to be used.

SetEnvironmentFile specifies the file from which to obtain Akaroa Environment settings. Otherwise, the defaults used by akrun(1) apply.

SetRandomState sets the initial state of the random number generator before running the simulation. It takes the following structure as an argument:

```cpp
class AkRandomState {
    public:
        int sequence;
        unsigned long phase;
}
```

where sequence specifies one of the multiplying coefficients listed in AkRandom(3), numbered from 0, and phase specifies a position within the sequence generated by that coefficient, as an integer in the range $0 \ldots 2^{31}-2$.

SetPrecision and SetConfidence set the required precision and confidence. Calling these methods is equivalent to using SetEnvReal (see below) to set the values of the "Precision" and "Confidence" variables.

SetEnvInt, SetEnvReal, SetEnvString and SetEnvEnum set the values of integer, real, string and enumerated-type valued Akaroa environment variables. They return 0 on success, and -1 if the variable does not exist or is of the wrong type.

Note: If a variable is set both by one of these methods and in a file specified with SetEnvironmentFile, the results are undefined.

Run launches the simulation and waits for it to complete. If it completes successfully, true is returned. Otherwise, false is returned, and ErrorMessage may be used to obtain a string describing the reason for failure. ErrorMessage returns a pointer to static storage.

Run may be called repeatedly to execute the simulation multiple times. If this is done, the initial random number state of each run will be the final state from the previous run, so that each run will receive a unique stream of random numbers.

GetNumParams returns the number of parameters for which results are available. GetResult returns the result for parameter $i$ in the following structure:
struct AkResult {
    long count;
    long trans;
    double mean;
    double variance;
    double delta;
};

count  The total number of observations made by all engines.

trans  The total number of observations discarded during the transient phases (before the system settled down into steady state).

mean   Estimate of the mean value of the parameter.

variance  Variance of the estimate of the mean.

delta  Confidence interval of the estimate of the mean. The true mean lies between mean - delta and mean + delta with probability \( p \), where \( p \) is the confidence level specified for the parameter.

GetRandomState returns the final state of the random number generator after running the simulation.

Callback

The following methods are called by the system in response to certain events. They may be overridden to take user-defined action when these events occur.

EngineStarted is called to acknowledge the successful launching of a simulation engine. The default implementation does nothing.

RandomOverflow is called if all sequences of random numbers are exhausted (i.e., all multipliers have been used). If AkSimulation::Continue is returned, the simulation is continued, starting with the first multiplier once more. If AkSimulation::Terminate is returned, the simulation is terminated with an appropriate error. The default is to terminate the simulation.

EngineLost is called if contact with a simulation engine is unexpectedly lost. If AkSimulation::Continue is returned, the simulation is continued with the remaining engines. If AkSimulation::Terminate is returned, the simulation is terminated with an appropriate error. The default is to continue.

EngineOutput is called whenever a simulation engine sends a message using the AkMessage routine. If AkSimulation::Continue is returned, the simulation is continued. If AkSimulation::Terminate is returned, the simulation is terminated with an appropriate error. The default is to write the message to the standard error of the invoking process, preceded with an identification of the process from which it came, and continue the simulation.

SEE ALSO
akrun(1), AkRandom(3), AkMessage(3)

Akaroa II User’s Manual

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
CurrentTime – Akaroa simulation clock

SYNOPSIS

#include <akaroa/time.H>

typedef real Time;

Time CurrentTime();

DESCRIPTION
CurrentTime returns the current value of the simulation clock.

The simulation clock is maintained by either the Process Manager or the Event Manager, whichever is being used (they cannot both be used in the same simulation program). The Process Manager should be used by new simulations; the Event Manager is provided only for backward compatibility.

SEE ALSO
Process(3), events(3)

Akaroa II User’s Manual

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
PriorityQueue – Akaroa prioritised queue class

SYNOPSIS

#include <akaroa/priority_queue.h>

template <class T>
class PriorityQueue : public Queue<T> {
    public:
        virtual void Insert(T *item);
    protected:
        virtual int HigherPriority(T *item1, T *item2) = 0;
    }

template <class T>
int HigherPriority(T *item1, T *item2);

DESCRIPTION
PriorityQueue implements a queue of items which is ordered according to a user-specified priority rule.

Insert inserts item into the queue at a position which is determined by its priority relative to other items in the queue.

HigherPriority defines the relative priority of items in this queue. An implementation of this method must be supplied; it should return true if item1 has higher priority than item2, false otherwise.

SEE ALSO
Akaroa II User’s Manual

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
Process – Akaroa Process Manager

SYNOPSIS

```cpp
#include <akaroa/process.H>

typedef real Time;

class Process {
public:
    Process(long stackSize = 1024);
    void Schedule(Time delay);

protected:
    virtual void LifeCycle() = 0;
};

Process *CurrentProcess();
void Hold(Time delay);
void Hold();
void DeleteProcesses();
```
Process::Schedule blocks the process until the simulation clock reaches the current time plus delay.

Process::LifeCycle contains the code which is executed by the process’s thread. If it returns, the thread is terminated and the Process object is deallocated.

If a Process is deleted by another process while its life cycle is active, its thread is terminated immediately.

CurrentProcess() returns a pointer to the currently executing process.

Hold(delay) is equivalent to CurrentProcess() -> Schedule(delay).

Hold() blocks the current process indefinitely. It will not run again until it is re-scheduled (using Process::Schedule).

DeleteProcesses() deletes all instances of class Process in existence. If a Process is queued for a Resource (or any other subclass of Semaphore) it is removed from the queue before being deleted.

SEE ALSO
CurrentTime(3), Resource(3)

Akaroa II User’s Manual

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
Queue – Alaroa queue class

SYNOPSIS

```cpp
#include <akaroa/queue.H>

template <class T>
class Queue {
    public:
        Queue();
        virtual void Insert(T *item);
        virtual T *Next();
        virtual T *Head();
        virtual void Remove(T *item);
        virtual int Empty();
        virtual int Length();
};
```

DESCRIPTION
Class Queue implements a first-in first-out queue of pointers to objects of a given type.

*Insert* adds *item* to the tail of the queue.

*Next* removes an item from the head of the queue and returns a pointer to it. If the queue is empty, 0 is returned.

*Head* returns a pointer to the item at the head of the queue, without removing it. If the queue is empty, 0 is returned.

*Remove* removes *item* from the queue, if present, regardless of its position.

*Empty* returns true if the queue is empty, false otherwise.

*Length* returns the number of items in the queue.

SEE ALSO
*Alaroa II User’s Manual*

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
Resource – Akaroa Resource class

SYNOPSIS

#include <akaroa/resource.H>

class Resource {
public:
    Resource(int capacity);
    void Acquire(int amount);
    void Release(int amount);
};

DESCRIPTION
Class Resource represents a finite resource which comes in discrete units, and coordinates processes competing for access to the resource.

Resource::Resource initialises the resource to have capacity units available.

Resource::Acquire blocks the current process until amount units of the resource are available, then allocates that many units to the process. Processes waiting for units to become available are allocated them on a FIFO basis.

Resource::Release releases amount units of the resource, making them available to other processes.

SEE ALSO
Process(3)

Akaroa II User’s Manual

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
Uniform, UniformInt, Binomial, Normal, LogNormal, Exponential, HyperExponential, Poisson, Geometric0, Geometric1, HyperGeometric, Erlang, Weibull – Akaroa random number distributions

SYNOPSIS
#include <akaroa/distributions.H>

real Uniform(real a, real b);
long UniformInt(long m, long n);
long Binomial(long n, real p);
real Normal(real m, real s);
real LogNormal(real m, real s);
real HyperExponential(real m, real s);
real Exponential(real m);
real HyperGeometric(real m, real s);
long Poisson(real m);
long Geometric0(real m);
long Geometric1(real m);
real Weibull(real alpha, real beta);

DESCRIPTION
These routines return pseudorandom numbers drawn from various distributions. They all use AkRandom(3) as a basic source of random numbers.

Uniform returns uniformly distributed real numbers in the range $a$ to $b$.

UniformInt returns uniformly distributed integers in the range $m$ to $n$, inclusive.

Binomial returns numbers from a binomial distribution of $n$ items where each item has a probability $p$ of being drawn.

Normal returns numbers from a normal distribution with mean $m$ and standard deviation $s$.

LogNormal returns numbers from a log-normal distribution with mean $m$ and standard deviation $s$.

Exponential returns numbers from an exponential distribution with mean $m$.

HyperExponential returns numbers from a hyperexponential distribution with mean $m$ and standard deviation $s$.

Poisson returns numbers from a Poisson distribution with mean $m$.

Geometric0 and Geometric1 return numbers from geometric distributions with mean $m$. Geometric0 returns integers $\geq 0$, whereas Geometric1 returns integers $\geq 1$.

HyperGeometric returns numbers from a hypergeometric distribution with mean $m$ and standard deviation $s$. 
Erlang returns numbers from an Erlang distribution with mean \( m \) and standard deviation \( s \).

Weibull returns numbers from a Weibull distribution with parameters \( \alpha \) and \( \beta \).

SEE ALSO

AkRandom(3)

Akaroa II User’s Manual

AUTHOR

Gregory C. Ewing, University of Canterbury
NAME
Schedule, NextEvent – Akaroa event manager

SYNOPSIS

#include <akaroa/events.h>

typedef real Time;

template <class T>
void Schedule(void (*proc)(T *), T *arg, Time delay);

int NextEvent();

NOTE
The Event manager is obsolete and has been superseded by the Process Manager (see Process(3)).
The routines described here are provided for backward compatibility only.

DESCRIPTION
These routines implement an event management system for discrete event simulation, which may optionally be used by simulation engines.

The type Time is used to represent simulation times.

Schedule schedules an event to occur delay units of simulation time from the current time. When the simulation clock reaches the scheduled time, the procedure proc will be called with the argument arg. The procedure may do whatever is required, including scheduling further events.

NextEvent should be called repeatedly from the main loop of the simulation. It retrieves the next event from the event queue, advances the simulation clock to its scheduled time, and calls the associated procedure.

Normally, NextEvent returns true. If there are no events in the event queue, NextEvent returns false. In this case, the simulation engine should do something to schedule one or more events so that the simulation may continue.

BUGS
Due to limitations of g++, the Schedule function is implemented as a macro rather than a template, and therefore cannot be overloaded.

SEE ALSO
CurrentTime(3)

Akaroa II User’s Manual

AUTHOR
Gregory C. Ewing, University of Canterbury
NAME
AkObservationType, AkDeclareParameters, AkObservation, AkSimulationOver – Report observations made during simulation to Akaroa

SYNOPSIS

#include <akaroa.h>

enum AkObservationTypes {
  AkCorrelated, AkIndependent
};

void AkObservationType(AkObservationTypes);

void AkObservation(real x);

void AkDeclareParameters(int numParams);

void AkObservation(int paramNum, real x);

int AkSimulationOver();

DESCRIPTION
These routines are used by a simulation engine to report observations to the Akaroa system and determine when to cease the simulation.

AkObservationType is used to declare whether the observations will be correlated or independent. The default is AkCorrelated. If AkIndependent is to be specified by calling this routine, it must be done before calling AkDeclareParameters or AkObservation.

AkObservation is used to report an observation to Akaroa. If only one parameter is being observed, it is simply passed to AkObservation.

If more than one parameter is to be observed, AkDeclareParameters should be called once to inform Akaroa how many parameters to expect. Then AkObservation should be called with two arguments, the parameter number (parameters are numbered from 1 upwards), and the observation.

After processing an observation, AkObservation tests whether the estimates of the means all the observed parameters have reached the required precision, and if so, terminates the simulation engine.

AkSimulationOver is an obsolete routine which tests whether all parameters have reached the required precision.

SEE ALSO
Akaroa II User’s Manual

AUTHOR
Gregory C. Ewing, University of Canterbury