Preface

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 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 Observing more than one parameter ............................. 4  
2.2 Random Numbers ................................................ 5  
2.2.1 Algorithm used by AkRandom ................................. 5  
2.3 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 Controlling the random number seed ....................... 9  
3.3.3 Messages you may get from akrun ......................... 9  
3.4 Shutting down the Akaroa system ............................... 10  
3.5 Debugging a simulation ......................................... 10  
3.5.1 Sending diagnostic information .............................. 10  
3.5.2 Running a simulation engine under a debugger .......... 10  
3.5.3 Precautions against excessively short runs .............. 11

4 The Akaroa Environment .............................................. 13  
4.1 Environment Variables ......................................... 13  
4.2 Environment Syntax ............................................. 14

5 Akaroa Library Routines ............................................. 17  
5.1 Random Number Distributions ................................ 17  
5.1.1 Synopsis ....................................................... 17  
5.1.2 Descriptions .................................................. 17  
5.2 Queues .............................................................. 18  
5.2.1 Synopsis ....................................................... 18  
5.2.2 Using Queues ................................................ 19  
5.2.3 Methods ......................................................... 19  
5.3 Priority Queues .................................................. 19  
5.3.1 Synopsis ....................................................... 19  
5.3.2 Using PriorityQueues ........................................ 19  
5.4 Process Manager ................................................ 20  
5.4.1 Synopsis ....................................................... 20  
5.4.2 Creating a process .......................................... 20  
5.4.3 Stack size ...................................................... 21  
5.4.4 Scheduling ..................................................... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4.5</td>
<td>Other routines</td>
<td>21</td>
</tr>
<tr>
<td>5.5</td>
<td>Resources</td>
<td>22</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Synopsis</td>
<td>22</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Methods</td>
<td>22</td>
</tr>
<tr>
<td>5.6</td>
<td>AkSimulation</td>
<td>22</td>
</tr>
<tr>
<td>5.6.1</td>
<td>Synopsis</td>
<td>23</td>
</tr>
<tr>
<td>5.6.2</td>
<td>Using AkSimulation</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>Examples</td>
<td>27</td>
</tr>
<tr>
<td>6.1</td>
<td>An M/M/1 Queueing System</td>
<td>27</td>
</tr>
<tr>
<td>6.2</td>
<td>A Multiprocessing Computer System</td>
<td>28</td>
</tr>
<tr>
<td>6.3</td>
<td>A Terminating Simulation</td>
<td>29</td>
</tr>
<tr>
<td>A</td>
<td>Obsolete Facilities</td>
<td>33</td>
</tr>
<tr>
<td>A.1</td>
<td>Event Manager</td>
<td>33</td>
</tr>
<tr>
<td>A.1.1</td>
<td>Event Manager Routines</td>
<td>33</td>
</tr>
<tr>
<td>B</td>
<td>Adding an analysis method to Akaroa</td>
<td>35</td>
</tr>
<tr>
<td>B.1</td>
<td>Introduction</td>
<td>35</td>
</tr>
<tr>
<td>B.2</td>
<td>What an analysis method does</td>
<td>35</td>
</tr>
<tr>
<td>B.2.1</td>
<td>Checkpoints</td>
<td>35</td>
</tr>
<tr>
<td>B.3</td>
<td>Implementing an analysis method</td>
<td>36</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Steps to implementing an analysis method</td>
<td>36</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Class ParameterAnalyser</td>
<td>36</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Declaring your analyser to Akaroa</td>
<td>38</td>
</tr>
<tr>
<td>B.3.4</td>
<td>Adding a value for AnalysisMethod</td>
<td>38</td>
</tr>
<tr>
<td>B.3.5</td>
<td>Adding your code to the Makefile</td>
<td>38</td>
</tr>
<tr>
<td>B.3.6</td>
<td>Recompiling Akaroa</td>
<td>38</td>
</tr>
<tr>
<td>B.4</td>
<td>Accessing the Akaroa Environment</td>
<td>39</td>
</tr>
<tr>
<td>B.4.1</td>
<td>Retrieving Akaroa environment variables</td>
<td>39</td>
</tr>
<tr>
<td>B.4.2</td>
<td>Defining new Akaroa environment variables</td>
<td>39</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>41</td>
</tr>
</tbody>
</table>
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 [1-5]. 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 it 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.

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.

```c
/*
  * 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`, which 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 2.2.

The above program may be compiled using the following `csh` commands. You will need to find out from your site administrator the locations of the Akaroa header and library files and substitute them in the indicated places.

```
% setenv INCLUDE pathname of Akaroa include directory
```

---

1You 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.
% setenv LIB pathname of Akaroa library directory
% g++ uni.C -I$INCLUDE -L$LIB -lakaroa -lm -o uni

The same 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

<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>0.483686</td>
<td>0.0218746</td>
<td>0.95</td>
<td>8.55314e-05</td>
<td>756</td>
<td>252</td>
</tr>
</tbody>
</table>

*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.

### 2.1 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);
    double y = x * x;
    AkObservation(1, x);
    AkObservation(2, y);
  }
}
```

Running uni2 produces output similar to the following:

% uni2

<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>0.492028</td>
<td>0.0148899</td>
<td>0.95</td>
<td>3.96252e-05</td>
<td>1512</td>
<td>252</td>
</tr>
<tr>
<td>2</td>
<td>0.322265</td>
<td>0.0159348</td>
<td>0.95</td>
<td>4.53877e-05</td>
<td>1554</td>
<td>259</td>
</tr>
</tbody>
</table>
2.2 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 \( 1 \leq n < 2^{31} - 1 \).

2.2.1 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 5 multipliers are available, for a total sequence length of 10,737,418,230.

These multipliers are the 5 best 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).

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

```c
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

\[
\frac{\sigma^2}{\sigma^2_X} = \frac{1}{N} \frac{\sigma^2}{\sigma^2_X}, \tag{2.1}
\]

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

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

(2.2)
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 programs: akmaster, akslave, and akrun.

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 passed 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.

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:

```
whio% akrun -n 2 uni
Simulation engine 3910 started on purau
Simulation engine 29335 started on mohua
Param  Estimate  Delta  Conf  Var  Count  Trans
1  0.494646  0.0210355  0.95  7.9095e-05  1506  251
whio%
```
3.3. Running a Simulation

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.

In some cases, you may wish to control which hosts are used. For instance, you may want to run two Akaroa simulations at once, and spread the load over the available processors on your local area network.

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 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 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
Param Estimate Delta Conf Var Count Trans
1 0.502473 0.0251216 0.95 0.000163424 503 0
```

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
Param Estimate Delta Conf Var Count Trans
1 0.494674 0.0247054 0.95 0.000158099 535 0
```

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

3.3.3 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 re-used 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 $10^{10}$.

3.4 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.

3.5 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.

3.5.1 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 \texttt{AkMessage} routine:

\begin{verbatim}
AkMessage(format, arg1, arg2, ...);
\end{verbatim}

\texttt{AkMessage} formats its arguments like \texttt{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.\footnote{In some earlier versions of Akaroa, text written to the standard error of a simulation engine was reported by akrun. This is no longer supported; \texttt{AkMessage} should be used instead.}

3.5.2 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

\begin{verbatim}
akrun -n 1 xxgdb mysim
\end{verbatim}
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.

### 3.5.3 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.
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, *akrun* looks for a file called “Akaroa” in the current directory, and, if it is present, reads environment settings from it.

*Note: The Akaroa Environment is separate from the Unix environment. You cannot change an Akaroa Environment variable using setenv.*

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

```
Precision = 0.01
Confidence = 0.90
```

This specifies the precision of all parameters to be within ± 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.

```
Precision = 0.01
Confidence = 0.90
parameter 1 {
  Confidence = 0.97
}
parameter 2 {
  Precision = 0.02
}
```

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 `-f` option to *akrun*, for example:

```
whoa% akrun -f my_environment -n 2 uni
```

### 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.  

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.

---

environment → { setting | parameter }
setting → identifier ‘=’ value
value → integer | float | identifier | string
parameter → ‘parameter’ integer ‘{’ { setting } ‘}’
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)

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.

```cpp
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:

```cpp
#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 Priority Queues

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.

    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:

    #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.

---

1 An exception to this is the process executing the main program, which uses the initial Unix stack, and will therefore have its stack extended when necessary.
Process *CurrentProcess()
Returns a pointer to the Process whose LifeCycle is currently executing.

void DeleteProcesses()
Deallocation 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)
Released 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.

\subsection*{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};

private:

    // 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}

\subsection*{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
*

```c
#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.

/*
 * 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. Observations 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.

```c
/*
  * 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

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.
//
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.
//
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.

```c
#include <akaroa/events.H>
typedef real Time;

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

```c
Pentium *p = new Pentium;
Schedule(Explode, p, 42);
```

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

```c
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:

```c
while (!AkSimulationOver())
    NextEvent();
```

```c
Time CurrentTime()
```

Returns the current value of the simulation clock.

---

1 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.
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;
    }
```
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.

```cpp
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`.

```cpp
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:

\[
\text{DefineParameterAnalyserType("name", class)}
\]

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

\[
\text{DefineParameterAnalyserType("MyMethod", MyAnalyser)}
\]

B.3.4 Adding a value for AnalysisMethod

You also have to add \textit{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:

\[
"\text{AnalysisMethod", "e", "Spectral", "Spectral", "BatchMeans", \\n".Independent", 0, \\n"
\]

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

\[
"\text{AnalysisMethod", "e", "Spectral", "Spectral", "BatchMeans", \\n".Independent", "MyMethod", 0, \\n"
\]

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:

\[
\text{AKANAL.OBJ = \} \\
\text{\$HOME/mystuff/my_analyser.o \} \\
\text{...}
\]

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:

\[
\text{make system}
\]

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

\[
\text{make mm1}
\]

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:

\[
\begin{align*}
\text{int GetInt(char *name)} \\
\text{real GetReal(char *name)} \\
\text{char *GetString(char *name)}
\end{align*}
\]

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:

\[
\begin{align*}
\text{void GetEnum(char *name, char *&value)} \\
\text{void GetEnum(char *name, int &value)}
\end{align*}
\]

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:

```plaintext
/*Name*/ /*Type*/ /*Default*/ /*Values*/
"MyInteger", "i", "42",
```
"MyReal", "r", "3.1415",
"MyString", "s", "strawberry",
"MyEnum", "e", "Honda", "Honda", "Suzuki", "Yamaha", 0,
Bibliography


