Akaroa2
User's Manual

Gregory Ewing
Krzysztof Pawlikowski
Donald McNickle

Preface

The original version of AKAROA was designed at the Department of Computer Science, University of Canterbury in Christchurch, New Zealand, by Dr Krzysztof Pawlikowski and Victor Yau (Department of Computer Science) and Dr Donald McNickle (Department of Management). The project was partially sponsored by Telecom Australia Research Laboratories in Melbourne.

The current implementation (AKAROA2) has been re-designed and re-implemented by Dr. G. Ewing, Dr. K. Pawlikowski and Dr. D. McNickle.

Contributions from Peter Smith, Ruth Lee, Mirko Eickoff, Will Gittoes, Jin Hong, Mofassir Haque, Ludger Bischofs, Adam Freeth and Martin Brožovič are also acknowledged.

For more information, please contact
Prof. K. Pawlikowski, Department of Computer Science and Software Engineering,
University of Canterbury, Christchurch, New Zealand
Email: krys.pawlikowski@canterbury.ac.nz
WWW: http://www.cosc.canterbury.ac.nz/krys.pawlikowski/

Contents

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 therefor 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 acceptable error and confidence level for each parameter. When the global estimates of all parameters have reached the required error 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 or C++ to simulate the system you wish to study, using whatever techniques you would normally use. [1] 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 estimated error is within the specified limit, 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 any language capable of calling a library routine written in C. The modelling facilities described in chapter 5 are only available to C++ programs, however.

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:

```
/*
 *   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
Param    Estimate       Delta  Conf         Var   Count   Trans
    1    0.492028   0.0148889  0.95 3.96252e-05    1512     252
    2    0.322265   0.0159348  0.95 4.53877e-05    1554     259
```

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 `AkRandomReal` as a basic source of random numbers:

```
real AkRandomReal();
```

Each time AkRandomReal is called, it returns a random real number x such that $0 < x < 1$, drawn from a uniform distribution.

2.5.1 Algorithm used by AkRandomReal

AkRandomReal uses a Combined Multiple Recursive pseudorandom number generator (CMRG) with a period of approximately 2^{191}. This sequence is divided into blocks of 2^{128} and one block assigned to each simulation engine.

The particular generator used is the one called MRG32k3a in Pierre L'Ecuyer, "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators", *Operations Research*, vol. 47, no. 1, Jan-Feb 1999, pp. 159-164. For more information, see the on-line manual entry AkRandomReal(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. 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:

```
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.) For an example of a simulation which uses this routine, see Chapter 6.

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

$$\hat{\sigma}^2_{\bar{X}} = \frac{1}{N}\hat{\sigma}^2_{X_i} \tag{2.1}$$

where X_i is the ith data item and N is the number of independent data items, and

$$\hat{\sigma}^2_{X_i} = \frac{1}{N-1}\sum_{i=1}^{N}(X_i - \bar{X})^2 \tag{2.2}$$

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 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 *ssh*, 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 or ssh, you must make any necessary additions to your search path in your shell startup file, not just in the shell from which you issue the remote command.

Shared vs. non-shared file systems

The Akaroa system is easiest to use if your home directory is shared between all the hosts on which you will be running Akaroa processes. The following examples assume that this is the case.

If your home directory is not shared, Akaroa can still be used, but you will need to copy the files `.akmaster` and `.akauth` from your home directory on the host where akmaster is running to your home directories on all the other hosts. This must be done *after* starting akmaster (and will need to be repeated if you shut down and restart akmaster).

WARNING: Take care with the `.akauth` file. This file, and any copies of it, should not be readable by anyone other than its owner (i.e. its permissions should be set to `-rw-------`). This is important for the security of the Akaroa system: if any other user can read the contents of your `.akauth` file, that user could run arbitrary processes under your user ID.

Example: Starting up Akaroa via *ssh*

Here is an example of using ssh to start 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 shell startup file to include the directory where her simulation programs reside, and the directory where the akaroa programs reside.

Note: Depending on how your system is set up, you may be asked to enter your password after each ssh command. This is not shown in the following examples.

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

Once an akslave is up and running, it breaks its links with the ssh. So, if the ssh 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 *argument*s 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 ID = 17
Simulation engine started: host = pukeko, pid = 23672
Simulation engine started: host = purau, pid = 434
Param    Estimate       Delta  Conf         Var   Count   Trans
    1    0.503476   0.0157353  0.95 4.42582e-05    1530     255
whio%
```

3.3.1 Specifying acceptable error and confidence level

By default, Akaroa runs your simulation until all results have a relative error of $\pm 5\%$ or better, at a confidence level of 95%. These can be changed using the `-e`, `-a` and `-c` options to akrun.

Relative error

You can specify the maximum acceptable relative error using the `-e` option to akrun. For example,

```
akrun -n 2 -e 0.02 uni
```

specifies a relative error of $\pm 2\%$ or better.

Absolute error

You can specify the acceptable error in absolute instead of relative terms using the `-a` option. For example,

```
akrun -n 2 -a 0.005 uni
```

specifies an absolute error of ± 0.005 or better.

> *Important:* If you suspect that the true mean of the quantity you are estimating could be zero or nearly zero, you will have to specify an absolute error, otherwise the simulation may never stop.

Specifying both relative and absolute error

If you specify both a relative *and* an absolute error, the simulation will be stopped when *either* error criterion is satisfied. This can be useful if you are unsure of the magnitude of the estimate, and therefore want to specify the error in relative terms, but also want to guard against the estimate turning out to be zero.

For example,

```
akrun -n 2 -e 0.02 -a 0.005 uni
```

will stop when the estimate reaches an error of either $\pm 2\%$ or ± 0.005, whichever happens first.

Confidence level

The confidence level can be specified using the `-c` option. For example,

```
akrun -n 2 -c 0.9 uni
```

will test the error of the results at a confidence level of 90%.

3.3.2 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).

3.3.3 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.

3.3.4 Controlling the random number seed

Each time you invoke akrun to start a simulation, Akaroa begins allocating blocks of random numbers to the simulation engines starting from the same point in the random number sequence. 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 allocator begins at the point where it left off after the previous run.

To find out the state of the random number allocator 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
whio%
```

Note the *RandomNumberState* (`0:20000` in this example) written out before the report. This indicates the state of the random number allocator at the end of the last repetition. To run the simulation again, starting the random number from 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
whio%
```

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

3.3.5 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

When using the obsolete LCG generator, a warning is issued if the random number stream is exhausted before the simulation completes.

The CMRG generator used in Akaroa 2.6 does not check for random number sequence exhaustion. The length of the random number block allocated to each simulation engine is 2^{128}, and Akaroa can allocate 2^{32} such blocks, so it is extremely unlikely that exhaustion will ever be a problem.

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

`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

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

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:

```
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.

[1] 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; AkMessage should be used instead.

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.

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 relative error and confidence level (if they differ 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 akmaster process.

- *alive*: The engine is running and reporting estimates.
- *dead*: The engine has died unexpectedly.

3. The *Relative Error* box displays a bar graph for each parameter being estimated. The red bar shows the relative error of the current global estimate, and the black triangle shows the maximum error requested for that parameter.

4. The *Global Estimates* table shows the current global estimate of each parameter, its relative error, 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, error 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. There are various ways that values can be specified for Akaroa Environment variables. One way is to supply an *environment file* for your simulation that specifies these settings; another is to use command-line options to *akrun*.

> *Note: The Akaroa Environment has nothing to do with the Unix environment. You cannot change an Akaroa Environment variable using the shell commands which set Unix environment variables.*

4.1 Environment Files

There are two ways to specify an environment file for a simulation:

1. Place a file called `Akaroa` in the directory where *akrun* is to be executed. When akrun starts up, it looks for this file, and if it is present, reads environment settings from it.

   > *Note:* A simulation engine running stand-alone also looks for this file. Currently this is the *only* method of specifying environment settings for a stand-alone simulation engine.

2. The `-f` option to *akrun* can be used to specify an alternative environment file, for example,

   ```
   akrun -n 2 -f my_env_file mm1 0.1
   ```

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

```
RelError = 0.01
Confidence = 0.90
```

The `RelError` variable specifies the acceptable relative error, and the `Confidence` variable specifies the confidence level. This example specifies the relative error 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 relative error of parameter 2 to 0.02, and the relative error and confidence levels of all other parameters to 0.01 and 0.90.

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

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

The full syntax of the environment file is presented in section 4.3.

Command line environment options

The `-D` option to *akrun* provides an alternative means of supplying Akaroa Environment settings. One `-D` option is required for each environment variable to be set, for example,

```
akrun -n 2 -D AnalysisMethod=BatchMeans \
   -D MaxTransientObs=10000000 mm1 0.9
```

Currently, this method can only be used to specify values which apply to all parameters. To specify values for particular parameters, an environment file must be used.

4.2 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 the Transient Phase

`MaxTransientObs = 1 000 000`
: Maximum allowed number of observations in the transient phase. If more than this number of observations is collected without the transient detector determining that the transient period is over, the simulation will be aborted.

`MaxSchrubenHeuristicObs = 10 000`
: Maximum allowed number of observations in the heuristic phase of the Schruben test. If the Schruben transient detector fails to leave its heuristic phase before this number of observations is collected, the simulation will be aborted.

Variables pertaining to all analysis methods

`RelError = 0.05`
: Maximum acceptable relative error. If this is set to zero, no relative error criterion is tested against (AbsError must be given a non-zero value in this case).

`AbsError = 0.0`
: Maximum acceptable absolute error. If this is set to zero, no absolute error criterion is tested against (RelError must be given a non-zero value in this case).

`Confidence = 0.95`
Confidence level.

`TransientMethod = Schruben`
Method of finding the length of the transient period. In the current version of Akaroa, only one method is available, based on the the Schruben test [9].

`AnalysisMethod = Spectral`
Method of estimating variance. In the current version of Akaroa, two methods are available: **Spectral** and **BatchMeans** [9].

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.

Variables pertaining to Random Numbers

`RandomGenerator = CMRG`
Algorithm for generation of random numbers.

`CMRG`
Combined Multiple Recursive Generator (period 2^{191})

`LCG`
Linear Congruential Generator (obsolete) (period $100(2^{31} - 1)$)

Other Variables

`KillSignal = 15`
: The signal with which to terminate simulation engines when the simulation is over. Typically useful values are listed below; see your Unix system man pages for signal numbers corresponding to other signals.

2	SIGINT
9	SIGKILL (cannot be caught or ignored)
15	SIGTERM

4.3 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 `AkRandomReal` as a basic source of random numbers.

5.1.1 Synopsis

The following random number functions are defined:

```
#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 ≥ 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:

```
#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.

```
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. (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.)

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:

```
#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 *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.

5.6.1 Synopsis

Class *AkSimulation* is defined as follows:

```
#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);

};
```

5.6.2 Using AkSimulation

To use the AkSimulation class, you first create an instance of it, specifying the command name and arguments to use to start the simulation engines. The 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 allocator.

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:

```
struct AkResult {
  long count;       // Total number of observations made
  long trans;       // Total 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 allocator. 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 exhaustion of the random number stream is detected during the simulation. By default, this method returns the value `AkSimulation::Terminate` which causes the simulation to be terminated with an appropriate error. If you override this method to return `AkSimulation::Continue`, the simulation will be continued with the random number stream starting again from the beginning. (*Note: Detection of random overflow is not implemented for the CMRG generator (the default in Akaroa 2.6 and later). This is because the sequence is sufficiently long to make random overflow impossible for all practical purposes*.)

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.

```
/*
 *   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 time spent in the system by the customer 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, time_in_system;
  arrival_time = CurrentTime();
  server.Acquire(1);
  Hold(Exponential(1/service_rate));
  server.Release(1);
  time_in_system = CurrentTime() - arrival_time;
  AkObservation(time_in_system);
}

// 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.

```
/*
 *   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.
//

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

Adding Observation Analysis Methods to Akaroa

A.1 Introduction

Akaroa 2.7 is designed in a modular fashion which permits new methods of analysing observations to be easily added. Two kinds of observation analysis modules can be added, Transient Detection methods and Variance Analysis methods.

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

A.1.1 Observation analysis phases

Analysis of observations in Akaroa is carried out in two phases, the *transient phase* and the *steady state phase*.

During the transient phase, observations are passed to the selected Transient Detection module, as determined by the `TransientMethod` Akaroa environment variable. The Transient Detection module discards observations until it determines that the transient phase is over, and the simulation has reached steady state.

The steady state phase is then entered, and observations are passed to the selected Variance Analysis module, as determined by the `AnalysisMethod` Akaroa environment variable. The Variance Analysis module decides when checkpoints should be taken, estimates the mean and the variance of the mean, and passes the estimates on to Akaroa for further processing.

A.2 Copying the Akaroa sources

Adding new modules to Akaroa involves modifying some of the existing sources, so 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 not be preserved by `cp`.

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

A.3 Adding a Transient Detection method

Implementing a Transient Detection method and adding it to Akaroa requires the following steps:

1. Write a new subclass of class `TransientDetector` which implements your method.
2. Declare your method to Akaroa by including a call to the macro `DefineTransientDetectorType`.
3. Add the name of your method to the list of possible values for the `TransientMethod` variable in the Akaroa environment. Optionally, you can also add new Akaroa environment variables for controlling your 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.

A.3.1 Subclassing TransientDetector

A `TransientDetector` performs transient detection for a single parameter. Akaroa will create an instance of your transient detector for each parameter to be analysed.

You will need to include the following header files:

```
#include "transient_detector.H"
#include "environment.H"
```

The constructor of your TransientDetector subclass should have the following signature:

```
MyTransientDetector(Environment *env);
```

There are two alternative ways to implement a TransientDetector:

1. Override the `TestObservations` method.
2. Override the `ProcessObservations` method.

You should override *either* one *or* the other of these methods, not both.

Overriding TestObservations

The `TestObservations` method has the following signature:

```
long TestObservations(long nobs, real obs[]);
```

Each time Akaroa receives an observation, the `TestObservations` method is called with a buffer containing all the observations collected so far. The `TestObservations` method should analyse these observations and determine whether they encompass the entire transient period. If so, it should return the number of transient observations to be discarded; if not, it should return -1.

The number of transient observations returned may be less than the number of observations in the buffer. In that case, the remaining observations will be passed to the variance analysis module before resuming simulation. It is thus possible for the transient detector to "look ahead" in the observation stream if it wishes.

Overriding ProcessObservation

This method is provided as an alternative for transient detectors that do not need to look ahead in the observation stream, and do not need observations to be buffered or want to perform their own buffering.

The `ProcessObservation` method has the following signature:

```
enum TransientResult {stillInTransient, outOfTransient};
TransientResult ProcessObservation(real value);
```

The `ProcessObservation` method receives observations one at a time. As long as the transient phase is not yet over, it should return `stillInTransient`. When it determines that the transient phase is over, it should return `outOfTransient`.

A.3.2 Declaring your transient detector to Akaroa

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

`DefineTransientDetectorType("`*name*`",` *class*`)`

where *name* is the name by which your method is to be know to the user, and *class* is the name of the class implementing your method. For example,

```
DefineTransientDetectorType("MyTransientMethod", MyTransientDetector)
```

A.3.3 Adding a value for the TransientMethod variable

You also have to add `name` to the list of valid values for the TransientMethod 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:

```
"TransientMethod",     "e",     "Schruben",    "Schruben",
                                                ".Independent",
                                                0,
```

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

```
"TransientMethod",     "e",     "Schruben",    "Schruben",
                                                ".Independent",
                                                "MyTransientMethod",
                                                0,
```

A.3.4 Adding your code to the Makefile

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

```
AKANAL_OBJ = \
        $HOME/mystuff/my_transient_detector.o \
        ...
```

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.

A.3.5 Recompiling Akaroa

Finally, you will need to recompile the Akaraoa system, and any simulation engines which are to use your new transient detector. See section A.5 for details.

A.4 Adding a Variance Estimation method

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

A.4.1 Checkpoints

Although the estimation 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 estimation 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 estimation 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 estimation 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 section A.6).

It is also possible for the simulation program to give hints to the estimation method as to where checkpoints should occur, by calling `AkCheckpoint` during the simulation.

A.4.2 Steps to implementing an estimation method

Implementing a variance estimation method and adding it to Akaroa requires the following steps:

1. Write a new subclass of class `VarianceEstimator` which implements your method.
2. Declare your method to Akaroa by including a call to the macro `DefineVarianceEstimatorType`.
3. Add the name of your 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 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.

A.4.3 Subclassing VarianceEstimator

A `VarianceEstimator` performs variance estimation for a single parameter. Akaroa will create an instance of your estimator for each parameter to be analysed.

You will need to include the following header files:

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

The constructor of your VarianceEstimator subclass should have the following signature:

```
MyVarianceEstimator(Environment *env, long trans);
```

The *env* parameter recceives the Akaroa environment. The *trans* parameter receives the number of observations that were discarded during the transient phase. (The *trans* parameter is provided for informational purposes only; the transient observations have already been discarded by the time the variance estimator is called.)

Your estimator should implement the following three methods:

`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 after processing each observation, to find out whether your estimator 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 estimator determines that it has reached a checkpoint, it should return `true`, otherwise `false`.

`boolean GetCheckpoint(Checkpoint &cp)`

This method is called in two circumstances: when your `ReachedCheckpoint` returns `true`, or when the simulation calls `AkCheckpoint`. If possible, the estimator should calculate a checkpoint, fill in the `Checkpoint` structure as described below, and return `true`. If for some reason it is not possible to calculate a checkpoint, it should return `false`.

The following fields of the `Checkpoint` structure should be filled in:

`cp.mean`	Estimate of μ
`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.

The remaining fields of the `Checkpoint` structure should not be changed.

A.4.4 Declaring your variance estimator to Akaroa

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

`DefineVarianceEstimatorType("`*name*`",` *class*`)`

where *name* is the name by which your method is to be know to the user, and *class* is the name of the class implementing your method. For example,

```
DefineVarianceEstimatorType("MyAnalysisMethod", MyVarianceEstimator)
```

A.4.5 Adding a value for the AnalysisMethod variable

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 DefineVarianceEstimatorType call) to the list at the end, before the final zero. For example:

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

A.4.6 Adding your code to the Makefile

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

```
AKANAL_OBJ = \
        $HOME/mystuff/my_variance_estimator.o \
        ...
```

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.

A.5 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 modules. 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`).

A.6 Accessing the Akaroa Environment

If desired, your module 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.

A.6.1 Retrieving Akaroa environment variables

Values of Akaroa environment variables are retrieved using the `Environment *` pointer passed to the constructor of a TransientDetector or VarianceEstimator. This points to an `Environment` object which has the following methods:

```
int  GetInt(char *name);
real GetReal(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:

```
char *GetEnumString(char *name);
int  GetEnumInt(char *name);
```

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 variance estimator which retrieves the value of two existing Akaroa environment variables, `CPSpacingFactor` and `CPSpacingMethod`, and stores them for later use.

```
class MyVarianceEstimator : public VarianceEstimator {
public:
  MyVarianceEstimator(Environment *env, long trans);
  ...
private:
  real cpsf;
  int cpm;
  ...
};

MyVarianceEstimator::MyVarianceEstimator(Environment *env, long trans) {
  cpsf = env->GetReal("CPSpacingFactor");
  cpm = GetEnumInt("CPSpacingMethod"); // 0 = Linear, 1 = Geometric
  ...
}
```

A.6.2 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",
"MyReal",           "r",    "3.1415",
"MyString",         "s",    "strawberry",
"MyEnum",           "e",    "Honda",        "Honda", "Suzuki", "Yamaha", 0,
```

Appendix B

Obsolete Facilities

This chapter describes parts of Akaroa 2 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 2.

B.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.

B.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 `proc` to be called with the given `argument` at the current simulation time plus `delay`. For example,

```
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).

`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:[1]

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

`Time CurrentTime()`

Returns the current value of the simulation clock.

B.2 Linear Congruential Random Number Generator

This section describes the linear congruential random number generator (LCG) that was used in versions of Akaroa2 prior to 2.6. In version 2.6 and later, the generator defaults to the Combined Multiple Recursive generator (CMRG). The LCG can be selected by setting the Akaroa environment variable `RandomGenerator` to `LCG`.

The LCG uses 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_LCG(3).

[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.

[2] George S. Fishman and Louis R. Moore III. *An exhaustive analysis of multiplicative congruential random number generators with modulus* $2^{31} - 1$. SIAM J. Sci. Stat. Comput. Vol. 7, No. 1, January 1986, pp. 24-44

Bibliography

[1] K. Pawlikowski and V. Yau. "On Automatic Partitioning, Runtime Control and Output Analysis Methodology for Massively Parallel Simulations". Proc. European Simulation Symp. ESS '92 (Dresden, Germany, Nov. 1992), So. Computer Simulation, 1992, pp. 135-139

[2] V. Yau and K. Pawlikowski. "AKAROA: a Package for Automatic Generation and Process Control of Parallel Stochastic Simulation". Proc. of the 16th Australian Computer Science Conference, ACSC '93, Brisbane, Australia, Feb. 1993, vol. A, pp. 71-82

[3] K. Pawlikowski, V.Yau and D.McNickle. "Distributed Stochastic Discrete-Event Simulation in Parallel Times Streams". Proc. Winter Simulation Conf. WSC'94, IEEE Press, 1994, pp. 723-730

[4] G.Ewing, D.McNickle and K.Pawlikowski. "Credibility of the Final Results from Quantitative Stochastic Simulation".Proc. European Simulation Congress, ESC'95, Vienna (Austria), Sept.1995, Elsevier, 1995, pp. 189-194

[5] D.McNickle, K.Pawlikowski and G.Ewing. "Experimental Evaluation of Confidence Interval Procedures in Sequential Steady-State Simulation". Proc. Winter Simulation Conference, WSC'96, San Diego, Dec. 1996, pp. 382-389

[6] G.Ewing, D.McNickle and K.Pawlikowski. "Multiple Replications in Parallel: Distributed Generation of Data for Speeding Up Quantitative Stochastic Simulation". Proc. of IMACS'97 (15th Congress of Int. Association for Mathematics and Computers in Simulation, Berlin, Germany, August 1997), Wissenschaft und Technik Verlag, 1997, pp. 397-402

[7] K.Pawlikowski, G.Ewing and D.McNickle. "Coverage of Confidence Intervals in Sequential Steady-State Simulation". J. Simulation Practice and Theory, vol. 6, no. 3, 1998, pp. 255-267

[8] K.Pawlikowski, G.Ewing and D.McNickle. "Performance Evaluation of Industrial Processes in Computer Network Environments". Proc. ECEC'98 (1998 European Conference on Concurrent Engineering), Erlangen, Germany, April 1998. Int. Society for Computer Simulation, 1998, pp. 160-164

[9] K. Pawlikowski. "Steady-state simulation of queueing processes: Survey of problems and solutions", *ACM Computing Surveys*, June 1990, pp. 123-170

[10] J.-S. R. Lee, K. Pawlikowski and D. McNickle. "Do Not Trust Too Short Sequential Simulation". Proc. SCSC'99 (Summer Computer Simulation Conference), San Diego, July 1999. Int. Society for Computer Simulation, 1999, pp. 97-102