University of Edinburgh

School of Informatics

MAKING SIMJAVA COUNT

M.Sc. Project Report

Costas Simatos

September 12th, 2002

Abstract

SimJava is a discrete-event process-based simulation API. Being easy to use and flexible, it has found widespread use among simulation practitioners either as a simulation tool in itself or as the basis for other tools and extensions. However, SimJava's simplicity is also its major shortcoming, requiring the modeller to manually undertake a number of tedious and error-prone tasks. This project's aim is to enhance SimJava in several ways in order to provide a powerful simulation tool, free of such burdens. The sampling methods used will be improved, sophisticated statistical support will be provided, powerful transient and termination conditions will be made available, and finally, detailed graphical output analysis will be provided as an option for simulations. These enhancements will be made available in an easy to use and automated manner, providing the modeller with powerful functionality and allowing him to focus on the modelling aspects of experiments.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Jane Hillston, for her invaluable support throughout the course of this project. Being always available and eager to provide guidance, her expertise and experience presented a great asset to my work.

I would also like to thank my fellow student Kannan Ratnasingham who used my project for his own simulation study of Freenet. His work provided me with ideas concerning new functionality and served to test this project to otherwise impractical extents.

Finally, I would like to give special thanks to Giota for her patience and support.

Table of contents

List of figures

List of tables

1 Introduction

1.1 Project description

SimJava is a discrete-event, process-based simulation API developed in Java. When it was first built, SimJava provided simulation practitioners with a flexible set of building blocks with which systems could be abstracted into simulations and subsequently run in order to study their behaviour. Having been developed in Java, SimJava offered portability in the form of platform independence as well as the ability to build animated simulations as applets to be included in web pages. Animated simulations could be made easily accessible and provide a powerful tool for presentation and education purposes.

The simplicity of SimJava, which provided it with its high level of flexibility, also proved to be its main shortcoming. Its minimal kernel provided little in the way of automating commonplace simulation tasks and required the modeller to expend much effort in order to specify the simulation's parameters and maintain control over its execution. This added effort, required in all but the most trivial simulations, diverted the modeller's attention from the pure modelling requirements of the system under study, and led to additional coding which, apart from obscuring the system's model, could easily prove to be error-prone and time consuming.

The primary focus of this project was to augment SimJava's statistical capabilities, one of the areas where SimJava was seriously lacking. This task affected all stages of the simulation, starting from the definition of measures and distribution samplers upon initialisation, the collection of observations throughout the simulation run, and finally the application of output analysis in order to test the quality of the obtained results. Apart from providing statistical support to SimJava, other necessities arose as the project was being planned. To complement the enhanced statistical support, almost all aspects of the original SimJava would need to be enhanced. These ranged from improving the simulation's reporting facilities to providing better control over the simulation's run length. As a whole, this project was developed as an incremental process in which most simulation requirements were catered for and provided through SimJava's new API in an automated and easy to use manner.

1.2 Project goals

1.2.1 Improvement of sampling methods

All simulation packages and tools utilise random number generators in order to drive experiments. Generators are used to produce sample values for input and internal parameters present within simulations. These parameters are rarely set deterministically but are rather defined to follow appropriate distributions. Samples from these distributions are obtained by modifying a uniform sample from a random number generator through the process of random variate generation. Random samples are generated using a seed, which determines the sequence of samples generated. These sequences have a maximum cycle length before repeating, which is determined by the generator's type and parameters.

From the simulation's point of view the sample values generated for each distribution need to be independent and their cycles non-overlapping. Furthermore, in order to be able to repeat experiments, the exact sequences of samples generated must be reproducible by specifying the same seeds. The need to reproduce these sequences calls for a pseudorandom number generator (PRNG) rather than a truly random one. These generators generate samples in a deterministic manner but produce sequences that pass statistical tests for randomness.

The current version of SimJava is lacking concerning the definition of distributions and their sampling. One limitation is the small number of predefined distributions available to the modeller. Furthermore, the PRNG used by SimJava to implement these distributions provides correlated sample values. Additionally, the seeds for the distributions' underlying PRNGs currently need to be explicitly set by the modeller. This task is far from trivial since seeds to produce non-overlapping cycles can't easily be identified manually. This limitation leads to common simulation errors that are all based on the improper seeding of the PRNGs used.

The first goal for this project is to provide an efficient PRNG for SimJava that produces good sequences of random samples. Furthermore, the task of specifying seeds for the PRNGs will not be left in the care of the modeller but will be performed automatically. Given an initial root seed and an estimate of the maximum samples required from the PRNGs, the simulation's kernel will automatically produce suitable seed values that will guarantee non-overlapping sequences of random numbers for each distribution. The modeller however will still have the ability to control the generators' seeding by supplying explicit seeds to generators or determining the seed sequence and sample spacing produced by the kernel. Finally, additional distribution classes will be made available to the modeller, which will include all the commonly used distributions in simulation models.

1.2.2 Improvement of statistical support

In many cases, simulation models are built to study and predict the behaviour of systems that can't otherwise be efficiently measured. This behaviour is characterised by measurements concerning the system's entities e.g. the utilisation of a web server or the average service time of a disk. Simulation models need to provide the modeller with the ability to easily specify measures of interest such as the ones mentioned above. Furthermore, collection of observations for the calculation of these measurements should ideally be done automatically or at least with minimum effort.

In quantitative modelling, a simulation used to approximate the behaviour of a system is only as useful as the quality of the measurements produced. In order to make sound decisions based on simulation output, a modeller must also be presented with the level to which the produced measures represent the unknown true system behaviour. Finally, these estimates together with their respective measures need to be presented to the modeller by automatically generating a report containing a summary of the simulation's run information.

Currently SimJava provides very little to support such statistical analysis. Only a single class exists to enable the modeller to collect observations. Furthermore, the recording of observations must be done explicitly in every case, and reports of the results must be generated manually. Additionally, no functionality exists to enable the modeller to estimate the quality of the measurements obtained.

One of the main goals of this project is to provide sophisticated mechanisms for statistical analysis. These mechanisms will be largely automated, as will the final report generation. A set of predefined measures available for calculation will be made available to the modeller such as a disk's throughput or utilisation, providing access to measurements such as the measure's sample mean and variance. The process of collecting observations and performing calculations for such measures will be automated and transparent, requiring the modeller only to select the appropriate measures. In the case that these predefined measures do not cover all of the modeller's requirements, other measures may be defined as custom and updated with minimum effort. Once the simulation has completed, a report of these measurements will automatically be produced without any additional effort.

In addition to providing easy to use methods of specifying measures of interest, the modeller will be informed of the quality of the obtained results. In order to provide this essential information, output analysis methods will be implemented. As in the case of measurement calculations, the modeller will need only to specify which method (if any) will be used for output analysis and it will automatically be applied to the collected observations. These methods will calculate confidence intervals for the measurements produced, and will be able to extend the simulation in order to obtain results of a given quality.

1.2.3 Improvement of transient period and run length definition

In many cases, a simulation is constructed in order to study a system's steady state behaviour. This is the state in which the system has overcome the bias of its original, starting state. In such cases of steady state analysis a suitable warm-up or transient period must be specified which will be discarded upon calculating measurements of interest. It may of course be the case that a modeller is interested in transient analysis, or in other words, the system's behaviour from its starting state. In any case, a termination condition is required if the simulation is to eventually complete. This is essential if measurements are to be extracted from the experiment and their quality estimated before being presented to the modeller.

Currently in SimJava, the warm-up period and termination condition must be explicitly set by the modeller and checked manually during the simulation's progress. Moreover, concerning the termination of the run, the modeller must decide on an explicit time at which this will occur, without taking into account the quality of the measurements calculated so far. It is apparent that more sophisticated methods for ending simulations need to be provided.

An important goal of this project is to provide the modeller with the ability to easily specify conditions for the transient period and run length. These conditions will be automatically maintained and checked by the simulation kernel without any further effort. Furthermore, they will be centrally defined and as such, lead to the easy identification of an experiment's settings. Several types of conditions will be available permitting the modeller to specify the simulated time allowed to elapse or the amount of work introduced into the system. Finally, the kernel may be allowed to automatically identify a transient period or more importantly define the simulation's run length, based on the quality of the measurements obtained by using variance reduction techniques.

1.2.4 Provision of graphical output analysis

Even though a simulation may accurately calculate measurements of interest and perform output analysis, it is only useful if the results are well presented to the modeller. As previously mentioned, once the simulation has completed it will automatically produce a detailed report containing all the measurements obtained as well as their confidence intervals and additional simulation information. It would be highly desirable however for a form of graphical output to be produced that would enhance the modeller's understanding of the system.

The current version of SimJava provides the `simdiag` package that can be used to construct diagrams of the simulation's progress. However, these diagrams only display the observations collected as the simulation progresses and not the progress of the measurements of interest. Such a graphical overview, in order to be flexible and meaningful can only be conducted once the simulation run has completed. Furthermore, `simdiag` graphs are quite limited with respect to their applicability and are quite difficult to use, allowing access only to expert modellers.

An additional goal of this project is to provide the ability to automatically generate graphical output. Graphs will be generated for all measures of interest and will provide functionality such as zooming and annotating. Rather than presenting the observations gathered at each time, these graphs will display the actual measurements of interest and how they progressed throughout the simulation run. The graphical output will also be modified to present information regarding the output analysis method used in the experiment. In order to inspect, annotate and store graphs, a graph viewing utility will be built that will provide easy to use functionality for satisfying the modeller's graph viewing requirements.

1.3 Problems encountered

The implementation of the above goals as one complete simulation API was a difficult undertaking. The approach followed for each individual goal was based on well-known and established algorithms and simulation techniques. This fact was beneficial for providing concise implementation guidelines but at the same time provided challenges by setting the minimum quality standard for the project at a high level.

The main challenge for this project however, apart from the problems that arose from the implementation of each individual goal, was combining all the elements into one complete package. Since SimJava was designed as a tool for building general-purpose simulations the enhancements this project introduced would have to work for any simulation, ensuring the seamless and correct co-operation of all underlying elements. This task would be further complicated by the fact that automation and ease of use were considered to be high priorities. The original SimJava provided flexibility and wide applicability by using a very basic kernel and requiring the modeller to manually implement and carry out high-level simulation tasks. This project was conceived with the ambitious goal of providing the modeller with powerful but easy to use and fully automated functionality, maintaining flexibility and applicability but reducing the effort required and permitting the modeller to focus on the purely modelling aspects of experiments.

Additional problems arose by the fact that this project's intention was not to build a new simulation API but to enhance SimJava. The use of SimJava did of course provide a proven kernel that was simple enough to expand and integrate this project's enhancements. If a new simulation package was built, much time and effort would be spent on building up its kernel and its simulation building blocks, therefore restricting the actual enhancements that would be introduced. However, SimJava's simplicity did at times require significant re-implementation of the kernel as well as re-thinking concerning how individual components interacted. Furthermore, the large number of existing SimJava simulations would need to be taken into account and required modifications to these would have to be kept at a minimum. However different the internal workings of the new SimJava version would be compared to the original, the interface provided to simulation practitioners would have to be kept as similar as possible. This requirement proved to be quite demanding as the implementation progressed, requiring a great deal of additional effort that would otherwise be unnecessary.

1.4 Dissertation structure

The focus of this chapter was to provide an introduction to the project. A brief, general description was provided in Section §1.1, the main project goals in Section §1.2, and a description of the project's challenges and problems in Section §1.3. This Section serves as a map for the structure of the rest of the dissertation.

Chapter 2 focuses on background knowledge that is required in order to understand and appreciate the issues discussed in the remaining chapters. The concepts of modelling and simulation for performance analysis are discussed here as well as methods and techniques commonly used in simulation studies. The approach for simulating systems adopted by SimJava is then discussed as well as its benefits and shortcomings. The latter are examined in order to highlight the motivations that led to the conception of this project's goals. This chapter concludes with a brief discussion of related work in the field of performance analysis simulation.

Chapters 3 through 6 focus on the project's major goals. In Chapter 3 the goal of providing better sampling methods is discussed. Chapter 4 focuses on the goal of providing sophisticated statistical support to SimJava simulations, possibly the most important goal of this project. Chapter 5 proceeds to address the goal of improving the definition of a transient period and the simulation's termination condition. Chapter 6 focuses on the goal of supplying automated and detailed graphical output for SimJava simulations. Each of these chapters follows a similar structure of first presenting the issues involved with the discussed goal, then the alternatives for achieving it, and finally the approach adopted for the new version of SimJava. Wherever suitable, code fragments are supplied to highlight details of the implementation, as well as simulation examples that serve to illustrate the use of the new functionality in SimJava simulations.

Chapter 7 presents an evaluation of this project. In this evaluation, the new pseudorandom number generator is tested with a multitude of tests that prove its statistical correctness. Following this, the old and new versions of SimJava are tested to compare their efficiency. This comparison is on the basis of memory usage and the time required to complete. Finally, a discussion is made concerning the functionality and ease of use provided by the new version contrasting the effort that would be required from modellers attempting to recreate such behaviour using the original SimJava.

In the end of this dissertation, Chapter 8 draws the final conclusions from the project. The project's goals are again briefly highlighted and presented along with the achievements made. At this point several minor goals that were also achieved but not presented as the project's main goals are discussed to complete the list of work that was successfully completed. Finally, suggestions are made for future work that could complement the accomplishments of this project and further improve SimJava.

2 Project background

2.1 Introduction

This chapter focuses on issues that are required in order to appreciate the remainder of the project. First, the main approaches for performing performance analysis are discussed and contrasted. The chapter then proceeds to highlight concepts involved with simulation in particular, identifying commonly found simulation components.

Following this brief introduction to simulation issues, SimJava is presented and its approach to simulation is described and discussed. This discussion also provides the opportunity to highlight the motivations that led to the extension of certain aspects of SimJava. Finally, the chapter concludes with a discussion of related simulation tools and packages.

2.2 Performance analysis

Performance analysis is concerned with the study of systems in order to better understand their behaviour, with regard to their performance characteristics, and make informed decisions about future actions. The obvious approach for studying a system's behaviour is to experiment with the system itself and observe any results obtained. However this approach, although appearing to be simple, proves to be quite challenging to apply. It may be the case that the system under study may be too dangerous to directly experiment with, a prime example being the experimentation with a nuclear reactor's safety settings. However, even if the system under study is safe to experiment with it may be too costly or time consuming to observe and perform numerous tests. A heavily loaded file server will be too costly to be turned off, experimented with and observed to collect results.

Alternatively, even if a system is available for experimentation it may be the case that measurements are very hard to extract. A great deal of effort could be required to extract simple readings and observations from within a complex system. Furthermore, the level at which these observations are obtained may not be the one desired. Finally, the ultimate situation in which direct experimentation can't be applied is when the system itself does not exist. Clearly in this case an approach must be found which system designers can use to make informed decisions. The solution to the problems of direct experimentation comes in the form of modelling.

The approach adopted by analytical modelling is to select an appropriate paradigm and use it to build an abstract model of the system, which is subsequently solved to obtain results. At the heart of most of these modelling paradigms lies an interpretation of the model as a set of states in a Markov process. Every high level model is modified to produce the process's state transition diagram in the form of an *infinitesimal generator matrix*. This matrix is then solved as a set of *global balance equations* and the *steady state probability distribution* is obtained. This is a probability distribution listing the probability of the model being in each state once steady state is reached i.e. once the system's behaviour begins to exhibit stability and regularity. The probabilities of this distribution can then be used to obtain performance measurements, which are the true goal of the model. This is usually achieved by assigning rewards to certain states of interest and using the steady state probabilities to obtain measurements such as a disk's throughput or utilisation.

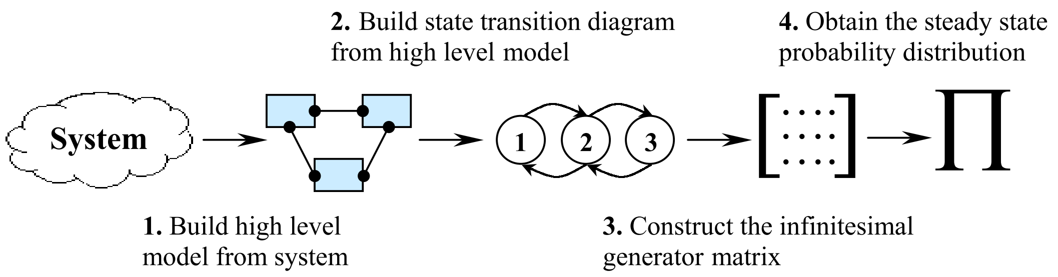

2. Build state transition diagram from high level model
4. Obtain the steady state probability distribution
System
1
2
3
Π
1. Build high level model from system
3. Construct the infinitesimal generator matrix

Figure 2.1: Typical steps for building and solving Markovian models

Simulation is the other major approach used to study the behaviour of systems. As in the case of Markovian modelling, an appropriate simulation API or tool is used and an abstracted model of the system is created. In this case however, the model isn't a set of states and transitions but a dynamic set of interacting processes or events. The simulation is run and the system's behaviour is observed by collecting observations of interest, rather than calculated in the form of a probability distribution. These observations can be easily used in calculations to produce the required measurements.

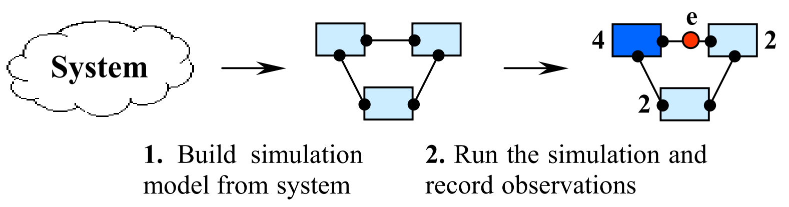

System
e
4
2
2
1. Build simulation model from system
2. Run the simulation and record observations

Figure 2.2: Typical steps for bullding and running simulation models

The main problem with simulation studies is the fact that several experiments need to be performed since each one represents only one sample path over the state space. On the other hand however, the benefits of studying systems through simulation are quite important. To begin with, simulations are not bound by the strict Markovian requirements that are present in analytical modelling techniques. This fact leads to great flexibility in capturing the behaviour of a system but also in the level of abstraction for the model. In addition, simulation permits transient analysis to be performed with the same ease as steady state analysis, which is quite difficult when using a Markovian modelling approach. Finally, since modelling techniques require the production of a generator matrix to capture the state space, certain systems may lead to excessively large models. This problem, known as *state space explosion*, seriously limits modelling techniques in their attempt to study large and complex systems. Although simulation does face problems of long runs for obtaining sufficient samples, state space explosion is not an issue since a system's behaviour is observed rather than calculated.

2.3 Simulation concepts

2.3.1 Event-based and process-based simulation

The most common approaches to discrete event simulation are *event-based* and *process-based* simulation. Event-based simulation focuses the modeller's attention to the specific events that occur within the system. The events currently present within the simulation are kept in a queue and are scheduled according to their designated time. Once an event is scheduled, its event routine, a set of predefined actions, is executed to modify the system's state.

Process-based simulation, although considerably slower in execution than event-based simulation, is much easier to use when modelling a system's behaviour. In this case, the system is considered as a dynamic set of processes that interact with each other by scheduling events. The events in this case serve to activate processes and determine their actions rather than drive the simulation itself.

2.3.2 Event scheduler

In both the major simulation paradigms discussed in Section §2.3.1 the concept of an *event scheduler* is central. The event scheduler's purpose is to maintain the queue of events waiting to be scheduled. It serves to advance the simulation time and activate appropriate events waiting in the queue. The event scheduler is one of the most frequently executed components of a simulation model. It is called whenever an event is generated or needs to be scheduled, and may be called several times during one event to generate new ones. It is therefore essential that it be implemented efficiently.

2.3.3 Simulation clock and time management

Every simulation model must maintain a central variable representing the simulated time, or in other words, the simulation's *clock*. This is maintained usually by the event scheduler that may advance the time either by one unit at a time or, more commonly, directly to the next event waiting in the queue. This latter approach is called *event-driven* time management.

2.3.4 Random number generation

Random numbers are essential in most discrete event simulations. They are used to generate delays for the simulation's internal and input parameters such as a disk's seek time or the packet inter-arrival time at a network buffer. These delays are rarely set deterministically but rather follow a certain appropriate distribution. Each distribution generates a sample by modifying a uniform sample between 0 and 1 obtained from a random number generator, through a process known as *random variate generation*.

The samples produced by the simulation's generators define the sample path the experiment will follow over the state space. The specific sample sequence produced by each generator is determined by a *seed* that needs to be well selected in order to avoid overlapping sequences and unwanted correlation. Using different seeds in the simulation's generators will produce a different sample path over the state space. More on the topic of random number generation will be discussed in Chapter 3.

2.3.5 Report generation

In order to study the performance measures derived from a simulation they must be collected and presented to the modeller in a suitable report file. Throughput the simulation, observations of interest are collected for the experiment's defined measures and are used to generate the desired measurements once the experiment has completed. In order to be of use, these results, along with other relevant information, need to be presented to the modeller. Most simulation packages and tools provide report generating facilities in order to structure and control the output of experiments.

2.3.6 Tracing

The trace of a simulation can be a very useful tool for debugging (also known as verification) and validation purposes. It is essentially a time-ordered listing of all the actions that took place within the simulation, providing the modeller with a complete account of the experiment's behaviour. Many simulation packages provide functionality with which trace messages can be generated or the simulation's default trace modified. Since trace generation is usually very inefficient it is generally only used in model development.

2.3.7 Transient and steady state analysis

The initial state assigned to a simulation model is called its *starting state*. It may be the case that experiments are conducted in order to study a system's behaviour starting from that initial state, or in other words, perform its *transient analysis*. An example of this could be an experiment to study the load on a file server in its first hour of operation. It is more common however that experiments are carried out to study the typical behaviour of a system, for example a typical hour of the file server's operation. In this case we are interested in performing the system's *steady state analysis*.

The system's steady state is the state in which the system is performing with a degree of regularity and can be considered as stable. In this state, the bias of the starting state has been overcome and the results obtained are representative of the system's actual, long-term behaviour. The time period from the beginning of the simulation up to point where it is considered to exhibit steady state behaviour is known as its *warm-up* or *transient period*. In the case of steady state analysis the observations obtained in this period need to be excluded from the final measurements.

2.3.8 Output analysis and variance reduction

As mentioned in Section §2.3.4, each run of a simulation represents a single path over the state space. This fact could mean that the measurements obtained from that run might differ considerably from the true behaviour of the system. In order to obtain a better estimation of a measure's true mean and estimate its quality, an *output analysis method* must be applied to the results. Commonly used output analysis methods attempt to obtain several independent estimates of the measure's true mean by performing additional runs with different generator seeds, or by segmenting a single run into independent sub-runs. These methods will be discussed in detail in Section §4.5.

Variance reduction techniques attempt to eliminate the bias of isolated runs by better estimating a measure's true mean. Such techniques can range from carefully selecting and modifying the stream of samples obtained from the simulation's random number generators, to attempting to initialise the simulation close to its steady state. Other techniques employ output analysis methods in order to drive experiments. In this case, output analysis methods are not used simply to estimate the quality of obtained results but to also dynamically extend the simulation if the obtained accuracy is not satisfactory. Output analysis methods in use as variance reduction techniques will be further discussed in Section §5.3.

2.4 SimJava

SimJava is a discrete event, process based simulation API, augmenting Java with building blocks for defining and running simulations. The approach adopted by SimJava is to regard systems as a set of interacting processes or *entities*. Entities communicate with each other by scheduling events through *ports*, which are unidirectional communication links with other entities. These events are maintained centrally in an event queue, which is managed along with the set of entities, by the simulation's kernel. The advancement of time is event-driven i.e. the simulation clock progresses on the basis of the event scheduling times.

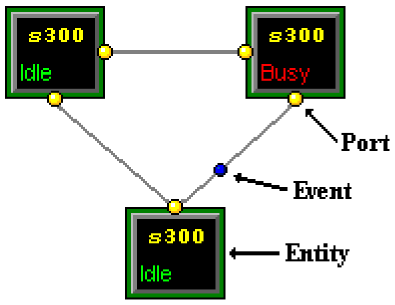


Figure 2.3: The basic components of a SimJava model

Entities in SimJava are subclasses of `Sim_entity`, an abstract class that encapsulates all the runtime functionality of entities. This class provides a `body` method that needs to be overridden to define the entity's behaviour. Ports are instances of `Sim_port` and are assigned to their entities with the `add_port` method. Events are created whenever a `sim_schedule` method is used or when an internal, control event occurs. `Sim_system`, the simulation's kernel, is responsible for maintaining all entities and events, and advancing the simulation's clock. `Sim_system` also provides methods for initialising the simulation, linking entities' ports, and ultimately running the experiment.

To receive events an empty `Sim_event` object needs to be created and used with a waiting method such as `sim_wait`. The event received will be passed into the empty `Sim_event` object, which will then provide access to the event's information such as its source, time and tag. The tag is a value that serves as an event type identifier and can be used in predicates to selectively wait for certain events. Events in SimJava are kept within two event queues. The *future event queue* is used to store all the events that have been generated but whose scheduling time has not yet been reached. The *deferred event queue* is used to store events that could not be received by entities either because they were not waiting for events at the scheduling time, or because the incoming event didn't match a specific predicate used by the entity.

Delays are experienced by entities using a hold method in which the delay time is specified. This delay time may either be specified deterministically, or sampled from an appropriate distribution. Entities may currently use the Exponential, Normal and Uniform distributions for sampling delay values.

In order to better understand the structure of SimJava simulations a simple example is provided. This example models a simple FIFO queue consisting of two entities: a source and a sink. The source generates events for the sink every 10 time units until a total of 1000 events have been generated. The sink receives incoming events and experiences for each arrival a delay of 8.5 time units. This is an extremely simple simulation in which delays are deterministically set and no measurements are produced. The example however serves to illustrate how SimJava's main components can be used to model a system's behaviour. The code for this simple simulation follows:

```
import eduni.simjava.*;

class Source extends Sim_entity {
  private Sim_port enqueue;

  Source(String name) {
    super(name);
    enqueue = new Sim_port("enqueue");
    add_port(enqueue);
  }
  public void body() {
    for (int i=0; i<1000; i++) {
      sim_schedule(enqueue, 0.0, 0);
      sim_hold(10.0);
    }
  }
}

class Sink extends Sim_entity {
  private Sim_port arrival;

  Sink(String name) {
    super(name);
    arrival = new Sim_port("arrival");
    add_port(arrival);
  }
  public void body() {
    Sim_event next = new Sim_event();
    int i = 0;
    while (true) {
      i++;
      if (i > 1000) break;
      sim_wait(next);
      sim_hold(8.5);
    }
  }
}

public class Queue {
  public static void main(String args[]) {
    Sim_system.initialise();
    Source source = new Source("source");
    Sink sink = new Sink("sink");
    Sim_system.link_ports("source", "enqueue", "sink", "arrival");
    Sim_system.run();
  }
}
```

Developed in Java, SimJava provides the benefits of platform independence as well as clean thread and memory management. Furthermore being a processed based simulation API it is quite easy to use in order to make abstract models of systems. This is aided by SimJava's simplicity, which allows great flexibility with respect to defining experiments. A major benefit of SimJava and one of its initial design goals, is the fact that experiments can be defined as animated applets as well as standalone applications. These applets can be embedded in web pages and used to greatly increase the accessibility of simulations. Furthermore, animated applets can be used as powerful presentation and education tools.

Several aspects of SimJava can, however, prove to be rather problematic under closer inspection. One first point concerns the use of distributions for generating delays to replace deterministically provided values. Each distribution is implemented using a random number generator upon which it relies to generate samples. However, the generator used by SimJava is Java's `Random` class, whose properties are withheld. This fact makes the task of identifying the generator's maximum cycle length, as well as seeds for well-spaced sample sequences, a difficult task. This is rather problematic since the modeller must explicitly set all the generators' seeds. With respect to distributions, SimJava is also quite limiting since classes are only provided for the Exponential, Normal and Uniform distributions. These sampling problems provided the motivation for implementing a new, proven random number generator, whose seeding would be automatically carried out by the kernel, as well as additional distribution classes to provide maximal flexibility to the modeller.

The simple example provided above does not produce any statistical output. SimJava's statistical support is provided through a single class, `Sim_accum`, which can be instantiated and used to store observations for certain types of measures. Measurements can be obtained from this class by relevant methods that calculate the observations' minimum, maximum and mean. In order to output these results, code must be written to collect all the calculated measurements and manually produce a report file. The approach usually adopted in SimJava is to define an entity for collecting all these measurements and using them to produce the experiment's report. This approach for obtaining output from a simulation is clearly limiting and error-prone and as such, provided the motivation for augmenting SimJava with sophisticated statistical support. Any type of measure should be catered for, output analysis methods should be applied on obtained results to estimate their quality, and a report file should be produced containing all the experiment's results. Furthermore, these tasks should all be carried out automatically or with minimum effort, without requiring any additional coding.

SimJava provides also little support for defining a simulation's transient period or run length. In the example provided, a simple termination condition is used based on the number of generated events. Usually, SimJava simulations use conditions based on the elapsed simulated time that need to be checked at each entity. This fact motivated SimJava's extension to provide a wider and more sophisticated range of transient and termination conditions that would be centrally defined and automatically checked.

2.5 Related work

This section serves as a brief description of several simulation tools and packages that influenced or where influenced by SimJava.

The original SimJava was itself an attempt to implement HASE++ in Java. HASE is a simulation environment built at the University of Edinburgh that focuses on the rapid development and execution of computer hardware simulations at multiple levels of abstraction [23]. HASE++ is a C++ simulation library built under the HASE project as an extension of SIM++, developed by Jade Simulations International. SIM++ has seen extensive use for computer architecture simulations and was extended by HASE++ in order to allow running simulations on architectures not supported by Jade. The development of SimJava as a Java-based simulation package based on HASE++ was motivated by the increased accessibility and platform independence that Java and in particular Java applets would provide [6]. Ongoing efforts are being made as part of the HASE project to build JavaHASE, a Java extension of HASE that uses the SimJava kernel to enable HASE simulations to be run as Java applets [22].

Distributed SimJava is an effort to extend SimJava by the MITRE Corporation. This uses SimJava's basic kernel and extends it in order to allow simulations to be executed in a distributed environment through Java's Remote Method Invocation. The goal of this project is to permit entities to run at remote hosts and share the simulation's workload [15].

SimProd is another SimJava extension built at the Asian Institute of Technology. This extension uses SimJava's kernel to provide a simulation package specifically tailored for modelling and experimenting with production systems [9].

Another SimJava extension was made by the Neko project of the Distributed Systems Lab of the Swiss Federal Institute of Technology in Lausanne. This project modified and used SimJava as a Java simulation framework for constructing and testing distributed algorithms on a variety of networks [16].

The final SimJava extension to note is GridSim, a grid simulation toolkit built at the University of Melbourne. It allows modelling and simulation of entities in parallel and distributed computing (PDC) systems for the design and evaluation of scheduling algorithms. GridSim is built upon SimJava's simple event handling mechanisms [2].

3 Improvement of sampling methods

3.1 Introduction

The previous two chapters provided an introduction to the project, covering the project's goals and necessary background. Sections §1.2.1 and §2.3.4 presented and initiated the discussion of random number generation and how SimJava's sampling methods could be improved. This chapter will continue this discussion and describe in detail the issues concerned with random number sampling.

Improving SimJava's sampling methods, the first goal of this project, may be considered as consisting of three steps:

- Provide a suitable random number generator
- Ensure that it is seeded properly
- Define a wide range of distributions in which to fit random samples.

3.2 Providing a new PRNG for SimJava

3.2.1 Desired properties of the PRNG

Design issues concerning SimJava's random number generator were focused on which specific generator type would be selected. Properties of PRNGs are investigated mainly with regard to their specific application. For instance, PRNGs are of common use in computer security in which case the main concern is high randomness to prevent forgeries and random number prediction by malicious parties.

In the case of a simulation package's generator, the first property of interest is the statistical randomness of the generator's output [8]. This is expressed mainly by the correlation between successive samples and their distribution's uniformity. Many statistical tests exist to test a sequence of samples for its randomness properties. The generators commonly used in simulation packages have been thoroughly tested and the results can be easily obtained.

Another property of interest is the generator's pseudorandomness, meaning that although appearing random, samples are deterministically produced [8]. A sequence of samples is produced by seeding the generator; the seed's selection determines which sequence will be produced. This property is essential if simulation runs are to be reproduced. Furthermore, effects of modifications to input parameters can be best measured when sampling variance is eliminated. PRNGs of simulation packages all have this property.

A third property that is vital for producing long runs is the generator's cycle length [8]. The cycle represents the maximum number of samples that the generator can produce given an initial seed, before the sequence loops and repeats itself. Generators with short cycles can seriously limit the effective length of simulation runs since a limited number of independent samples can be obtained in each one. A generator's cycle length depends on its type and its specific properties.

Finally, PRNGs used in simulation packages need to provide efficient sampling [8]. During a simulation run, PRNGs are used to generate samples for input and internal parameters, which are modified to fit their specific distributions. In most simulations a large number of such samples is required. As such the PRNG needs to be efficient, mainly with regard to its speed, in order not to present a substantial overhead for the simulation run.

3.2.2 Alternatives concerning PRNG selection

Most simulation literature agrees on some interesting issues concerning the selection of a PRNG for a simulation package. The first point made is that high complexity does not mean high randomness [1][8]. A testament to this fact is the wide use of generators based on simple linear conguential generators (LCGs), which produce samples through a single iterative calculation. The second point of advice is that simulation package developers should avoid providing their own PRNG implementations [8]. In contrast it is suggested to use a generator already in use which has been extensively tested and proven to be good.

The points stated above and the desired properties of PRNGs for simulation, provide a basis for evaluating available generators. The first decision made was to avoid implementing a new PRNG but instead use an existing one. In addition, the ability to reproduce simulation runs exists in all PRNGs currently is use; when seeded with the same root seed they all produce identical sequences of samples. As such, the main properties to be considered are cycle lengths, randomness properties and efficiency.

The first PRNG considered was the one used in JavaSim [17], a simulation package developed by the University of Newcastle upon Tyne. The obvious appeal of this generator was the fact that it provided a ready implementation in Java, which could easily be modified for use in SimJava. This generator is based on shuffling the output of an LCG with a multiplicative generator. However this generator provides a short cycle length (2^{24}), a fact that reduces the useful length of simulation runs.

After considering this limitation, a PRNG was sought that would not be restricted a short cycle. On this basis the Mersenne Twister generator (MT19937) [14] was examined as a candidate for SimJava. MT19937 offers a huge cycle of 2^{19937}-1 that would more than suffice for any simulation study. Furthermore, MT19937 has been thoroughly tested and has passed all tests concerning the randomness of its output [19]. Additionally, sample generation using MT19937 is very fast. With respect to these properties it would seem that MT19937 is an ideal choice for SimJava's PRNG. However, this generator was not selected because of a limitation concerning the goal of automatically providing seeds for non-overlapping sample sequences. To be precise, MT19937 needs to be seeded with 624 root seeds. Although these can be calculated based on a single seed, the process of automatic seed selection becomes rather complicated. This sub-goal is further discussed in Section §3.3.

Following the failings of the generators considered so far, a survey was made to examine the generators used in current simulation packages. The most commonly used types of generators are multiplicative linear conguential generators (MLCGs) [4]. These are an extension to basic LCGs that were found to have fairly poor randomness properties [1][8]. MLCGs are very simple and thus easy to implement. Samples are produced by calculating:

$$Y_i = A \cdot Y_{i-1} \bmod M$$

where Y_i is the next seed, Y_{i-1} the previous one, A the multiplier and M the modulus. For uniform samples between 0 and 1, Y_i is divided by the modulus. The choices of the multiplier and the modulus determine the generator's output randomness and cycle length. The most commonly used modulus is 2^{31}-1 [4][8], which can provide a full cycle of length 2^{31}-2, when A is selected to be a primitive root of M.

3.2.3 Selection of the PRNG for SimJava

An MLCG was chosen to be SimJava's PRNG. This decision was based on the simplicity of the generator and its high speed. The selected modulus was 2^{31}-1 since it is commonly used and provides a suitable cycle length. The choice of the multiplier required further inspection since failures of such generators are mostly attributed to improper multipliers. Primitive root multipliers currently used with the 2^{31}-1 modulus include:

- 16,807 used by the MLCG in SIMAN and SLAM
- 630,360,016 used by the MLCG in SIMSCRIPT II.5
- 742,938,285 used by the MLCG in GPSS/H

Statistical tests [3][4][5] proved that 16,807 does not provide very good randomness properties. Furthermore, these studies proved that MLCGs provide sample sequences with the best randomness properties when used with the 742,938,285 multiplier. As such, this was the multiplier selected for SimJava and is used in the generator as follows:

$$Y_i = 742{,}938{,}285 \cdot Y_{i-1} \bmod (2^{31} - 1)$$

Implementing this PRNG in Java was a straightforward task. The main point of concern when implementing any generator in a programming language is avoiding numeric overflows. Since calculations with very large numbers are frequent, it is quite often the case that the straightforward implementation of the above formula is not possible. Alternative methods for calculating the next seed must be then introduced, the concern in this case being that efficiency does not suffer too much from the added complexity. Luckily, Java's long arithmetic is sufficient for the calculations. The maximum possible result, obtained when Y_{i-1} is equal to 2^{31}-2 and is multiplied with the multiplier, is less than 2^{64}, the maximum long value. As such, the implementation of the PRNG in class `Sim_random_obj` was straightforward:

```
public class Sim_random_obj implements ContinuousGenerator {
  private final long a = 742938285;  // The multiplier
  private final long m = 2147483647; // The modulus
  private long x;                    // The last seed
  ...
  public double sample() {
    x = (a * x) % m;
    return (double)x / (double)m;
  }
  ...
}
```

This generator fulfils all the requirements for the PRNG of a simulation package. It is fast and simple, provides good output randomness properties, and has a sufficiently long cycle.

Furthermore, automatic seed selection for non-overlapping sample sequences can easily be provided. This is further discussed in the next Section.

3.3 Providing automatic seed selection

3.3.1 Issues concerning automatic seeding

The selection of seeds for the PRNGs used in a simulation is of high importance. Each distribution's PRNG needs to be seeded in such a way that the variates produced are based on non-overlapping sample sequences. However, providing different, random seeds for each distribution is not enough. Each PRNG must be given a specific seed based on the sample sequence spacing desired by the modeller [4][8].

Automating the process of seed selection for each distribution is one of the sub-goals concerned with improving SimJava's sampling methods. One of the main reasons for the automation of this process was to enhance ease of use. It is difficult for the modeller to be aware of what constitutes good and well-spaced seeds since this would require knowledge of the PRNG's internals and effort in identifying each seed. Automating this process allows the modeller to focus on true simulation aspects such as the entities involved and their behaviour. Furthermore, this automation reduces the scope for bad seed selection and ensures, at least at the level of sampling, the meaningfulness of the observations and measurements obtained. Finally, although the process of selecting seeds for each distribution should be transparent it should also be overridable to cater for experienced modellers.

3.3.2 Alternatives for implementing automatic seeding

Given an initial root seed and a desired spacing of the sample sequences, generating well-spaced seeds is quite simple. The process merely involves seeding the PRNG with the root seed and sampling it as many times as specified by the spacing [8]. The current seed of the generator and the initial seed produce non-overlapping sample sequences with distance equal to the spacing.

Several simulation handbooks [1][4][8] provide tables of seeds for specific generators and sample spacings. Such tables can be easily generated using the procedure described in the previous paragraph. A similar table could be constructed for SimJava's PRNG using an arbitrary initial seed and measuring seeds for small enough spacings to be able to match the modeller's specifications. This table could be stored in a file and examined to select a suitable seed for each distribution. This approach however obviously limits flexibility. All seed values are calculated based on a specific initial seed, which can't be specified by the modeller. Furthermore, the spacing desired by the modeller may well differ compared with the spacing used to calculate the seeds. Although this problem can be overcome by using a small spacing to generate the seed file, this would possibly lead to a very large file. In addition, the time saved by not calculating each seed must be considered along with the I/O overhead of opening and reading the seed file. Finally, when running as an applet, the seed file can't be used due to applets' sandboxing restrictions. Considering the fact that providing "live" simulations in web pages as applets was one of the original motivations of SimJava, this could be considered a serious problem.

Since the approach of storing seed values in a file is problematic, on the fly seed generation must be selected. Considering that the PRNG selected for SimJava is an MLCG, two alternatives exist. The first is to calculate the desired seed given a root seed and their required spacing [8]. Seed generation resorts to making the following single calculation given these parameters:

$$Y_n = \left(A^n \cdot Y_0\right) mod\, M$$

where Y_n is the resulting seed, Y_0 the root seed, A the multiplier, M the modulus, and n the desired spacing. However, although this might seem efficient, the calculation involved would require exponentiation of the multiplier to the spacing required. Apart from having to deal with almost guaranteed overflow problems, this calculation is extremely time consuming. It is therefore more efficient to simply produce a number of samples equal to the spacing and return the last seed [8]. This constitutes the second alternative in seed generation.

3.3.3 Selected implementation for automatic seeding

The approach selected in SimJava is the last one stated in the previous Section. The modeller is required to supply a desired spacing and a root seed that serves to initialise a PRNG that will be used to cycle through the seed values. For each distribution that requires seeding, this PRNG is sampled until it obtains the next seed to suit the specified spacing. If the modeller neglects to specify a root seed and a spacing, default values are used.

To illustrate the process of automatic seeding the constructor of `Sim_random_obj` is presented:

```
public Sim_random_obj(String name) {
  x = eduni.simjava.Sim_system.next_seed();
  this.name = name;
}
```

Note the use of `Sim_system`'s method `next_seed` to obtain the next well-spaced seed. The implementation of this method is presented next:

```
public class Sim_system {
  private static Sim_random_obj seed_source;
  ...
  public static long next_seed() {
    long new_seed = seed_source.get_seed();
    if ((new_seed == root_seed) && not_sampled) {
      not_sampled = false;
    } else {
      for (int i=0; i < seed_spacing; i++) {
        seed_source.sample();
      }
      new_seed = seed_source.get_seed();
    }
    return new_seed;
  }
  ...
}
```

Additional methods are also provided for the modeller to alter the root seed and spacing during the simulation run, if such an action is desired. Finally, for experienced modellers wishing to explicitly manage all aspects of the simulation, seeding of the PRNGs can be done manually. This is possible through additional constructors for the distribution classes in which the seed for the PRNG can be explicitly set.

3.4 Providing additional distributions classes

3.4.1 Issues concerning the provision of additional distributions

As previously mentioned, the current version of SimJava provides only a limited number of distributions: Exponential, Uniform and Normal. The implementation of additional distribution classes although important to facilitate the modeller, does not provide many noteworthy implementation choices. Random variate generation techniques are widely published [1][4][8] and easily implementable once a suitable PRNG is in place. The main approach used in simulation literature is the inverse transformation method.

Two issues regarding the implementation of the new distribution classes were considered. The first one falls under the general aim of minimising modifications required to existing SimJava simulations, in order to run under the new version. With this in mind, the way in which distribution classes are instantiated is kept identical to the way followed in the previous version. As an example the constructor of the Negative exponential distribution class, `Sim_negexp_obj` is presented:

```
public Sim_negexp_obj(String name, double mean) {
  ...
  source = new Sim_random_obj("Internal PRNG");
  this.mean = mean;
  this.name = name;
}
```

The `sample` method of each distribution class is used to generate the distribution's next sample. For `Sim_negexp_obj` this method is implemented as follows:

```
public double sample() {
  return -mean * Math.log(source.sample());
}
```

The second issue concerns providing a framework with which new distributions can use existing ones for variate generation. To be precise, each distribution instance holds a PRNG seeded with the value provided by the automatic seed selection method discussed previously. In many cases, variate generation for one distribution requires sampling another. It is desirable in this case to generate samples from the other distribution based on random samples from the initial distribution's PRNG. To serve this goal, a class method was provided for each distribution that generates samples using another's PRNG instance. For `Sim_negexp_obj` this method is implemented as follows:

```
static double sample(Sim_random_obj source, double mean) {
  return -mean * Math.log(source.sample());
}
```

3.4.2 Distributions provided by SimJava's new version

The distributions that were implemented and made available under the new version of SimJava are the following:

- Continuous distributions:
  - Beta Distribution (class `Sim_beta_obj`)
  - Beta Prime Distribution (class `Sim_betaprime_obj`)
  - Cauchy Distribution (class `Sim_cauchy_obj`)
  - Chi-Square Distribution (class `Sim_chisquare_obj`)
  - Erlang Distribution (class `Sim_erlang_obj`)
  - F-Distribution (class `Sim_f_obj`)
  - Gamma Distribution (class `Sim_gamma_obj`)
  - Inverted Gamma Distribution (class `Sim_invgamma_obj`)
  - Logistic Distribution (class `Sim_logistic_obj`)
  - LogNormal Distribution (class `Sim_lognormal_obj`)
  - Negative Exponential Distribution (class `Sim_negexp_obj`)
  - Normal Distribution (class `Sim_normal_obj`)
  - Pareto Distribution (class `Sim_pareto_obj`)
  - Student's t Distribution (class `Sim_tstudent_obj`)
  - Uniform Distribution (class `Sim_uniform_obj`)
  - Weibull Distribution (class `Sim_weibull_obj`)
- Discrete distributions:
  - Bernoulli Distribution (class `Sim_bernoulli_obj`)
  - Binomial Distribution (class `Sim_binomial_obj`)
  - Geometric Distribution (class `Sim_geometric_obj`)
  - Pascal Distribution (class `Sim_pascal_obj`)
  - Poisson Distribution (class `Sim_poisson_obj`)

4 Improvement of statistical support

4.1 Introduction

In the previous chapter the first goal of the project, improving SimJava's sampling methods, was covered. In this chapter the second major project goal will be discussed, the improvement of the statistical support provided to SimJava simulations.

One of the major limitations of the original SimJava is the minimal amount of statistical support it provides to the modeller. Only a simple class (`Sim_accum`) is provided which the modeller can use to calculate basic measurements. It is clear that more sophisticated and automated support is necessary if detailed and accurate measurements are to be made possible. The statistical support provided in SimJava's new version should include the following:

- Automatic calculation of default measures
- Easily updateable custom measures
- Output analysis methods
- Automatic report generation

4.2 Providing automatic calculation of default measures

4.2.1 Issues concerning default measures

One of the main motivations of simulation is to study a system's behaviour by calculating and examining measurements. Some of the measures required are found to be of interest in many situations and can be calculated regardless of the specifics of each simulation. Such measures can be considered as *default measures*.

Since default measures can be obtained for any simulation, collecting observations for these need not be a task under the modeller's care. In contrast, observations can be obtained automatically while the simulation progresses and handles situations such as the scheduling of events to entities. One point that requires caution however is the identification of different measure types. This is of substantial importance since a measure's type determines the way in which relevant observations are collected, and the way in which calculations are finally carried out [4]. Generally speaking, measures fall under three categories:

- *Rate based measures*. These measures are based on the occurrence of events of interest. Such measures are the arrival rate, where the event concerned is an event arrival, and the throughput, in which case the event of interest is the completion of service for a job.
- *State based measures*. These measures concern different states that the entity of interest might find itself in. Apart from recording the specific state at each time, it is essential to also record the time period for which the entity was in that state. Examples of state based measures are the entity's queue length and its utilisation.

- *Interval based measures*. This third type of measure does not concern the entity directly but the events that it processes. For each event a time interval is stored whose meaning depends on the specific measure. For example, interval based measures could be the events' waiting or residence time.

The measurements that can be obtained for each measure depend on the measure's type. For instance, no sample variance or standard deviation can be obtained for rate based measures since their observations consist of event occurrence times. On the other hand, event counts that are meaningful for rate based measures can not be obtained for state or interval based measures. This illustrates the need to know each measure's type before attempting to calculate measurements for it.

In Section §3.2.1 where the desirable qualities of simulation PRNGs were described, one point of concern was the sampling speed of the generator. Slow sampling seriously increases the time required for the simulation to complete. A similar issue must also be considered concerning the collection of observations and the calculation of measurements. Since many thousands of observations may need to be collected, the time each observation requires must be minimal. Furthermore, if measurements are not computed as the simulation progresses, but observations are stored for calculation at the simulation's end, an additional point of concern should be the storage requirements. Situations such as memory overflow need to be taken into account. However, since large quantities of memory can be cheaply obtained, the main concern remains the time required to collect observations.

An additional issue that concerns the collection of observations is how they are stored. The goals of output analysis and sophisticated transient and termination conditions require that additional data need to be stored with each observation. To be precise, it is essential to store timing information along with each observation in order to determine the exact simulation time that it was collected. Such information makes it possible to determine if the observation should be considered as part of a time period such as the transient period or a certain observation batch.

In many cases, the modeller is only interested in specific measures of specific entities. In such cases collecting observations for all possible measures is greatly inefficient. Such an issue makes it important to provide the modeller with a means of specifying which measures are of interest for his experiment. Furthermore, a choice should be provided concerning the event types for which measures are to be calculated. This becomes an issue since events with specific tags are commonly used as control messages between entities. It is clear that such events should not be considered when calculating measurements. Moreover, even without such control messages, the modeller should be provided with the freedom to exclude certain event types from the measurements. The ability to make such selections needs to be provided and the selections themselves must require minimum effort.

4.2.2 Alternative approaches for providing default measures

The alternatives for supplying default measures mainly concern the way in which measures are selected and how calculations take place. Since automation is of great importance, observation collection and measurement calculation needs to be abstracted away from the modeller in the form of a separate class. One first question would be how to make this class accessible to the modeller.

One approach would be to provide a statistics-managing object to each entity by default. This could be a part of the entity's superclass and made accessible to the entity. Another approach would be to enable the modeller to instantiate such an object where needed. This second method seems more appealing since if follows the approach currently used in SimJava, in which each entity defines in its constructor the ports and sample generators it will use. It seems fitting that an entity would define in a similar manner its statistics-managing object.

Another issue is the way in which measures are selected for each entity. Since default measures are commonly found in most experiments, one approach could be to calculate all possible measures by default. However, this could mean that some of the measures will never actually be used and as such, their computation would be unnecessary. Concerns of efficiency, namely memory load and time required, rule out this approach and make the selection of measures of interest the best solution. One way of providing such a selection would be to allow the modeller to instantiate a separate statistics-managing object for each measure. This approach however would complicate the collection of observations, since for each event occurrence all the statistics-managers would need to be probed in order to find the one of interest. Moreover, it is often the case that a single event generates observations for a number of measures. An example of this is the completion of service for an event, in which case, the entity's throughput, service time and residence time would need to be updated. Such situations make the central control of all defined measurements more appealing. In this approach, one statistics-manager is responsible for all the entity's measures. Apart from simplifying the observation collection process, this approach makes the entity's measurements more easily accessible. Access to the entity's measurements is required since actions such as output analysis and report generation are based on these measurements.

One final consideration is how the measurements will actually be obtained from the observations. One approach would be to recalculate the measurements upon each related observation by use of recursive techniques [4]. Such an approach would minimise the memory requirements for storing observations since each observation wouldn't actually be stored, but rather used to re-compute the measurement of interest. Although providing very attractive memory efficiency, this method would prove problematic for sophisticated output analysis and variance reduction techniques. Several of these techniques require all the observations obtained to be available. An example of this is when the batch means method is used for variance reduction, in which case all the observations are needed in order to estimate the serial correlation between batches [1]. Such requirements make it necessary to store each observation. Finally, as mentioned in the previous section, timing information needs to be stored with each observation in order to know the exact simulation time that each one occurred. In order to achieve this, observations for rate based measures take the form of the occurrence time, and for state and interval based measures, the time intervals are represented by storing the intervals' start and end time.

4.2.3 Approach chosen for providing default measures

For the new version of SimJava, the statistical support for entities is encapsulated within a single class (`Sim_stat`). An instance of this class is made for each entity that wishes to calculate measures. The default measures of interest are selected by calling a relevant method of the statistics-managing object with each desired measure as a parameter. This is the only task required from the modeller in order to collect default measurements from entities. Observation collection and measurement calculation is all handled transparently by the statistics-manager. The default measures provided for entities are:

- Rate based measures:
  - Arrival rate
  - Throughput
- State based measures:
  - Queue length
  - Utilisation
- Interval based measures:
  - Waiting time
  - Service time
  - Residence time

The measurements that can be obtained from these measures are:

- The sample mean (for all measures)
- The sample variance (for state and interval based measures, except utilisation)
- The sample standard deviation (for state and interval based measures, except utilisation)
- The maximum observation (for state and interval based measures, except utilisation)
- The minimum observation (for state and interval based measures, except utilisation)
- The event occurrences' count (for rate based measures)
- Exceedence proportions (for state and interval based measures, except utilisation)

In order to illustrate the process of defining measures of interest a simple example is presented. This example is based on the FIFO queue simulation that was presented in Section §2.4. The simulation was modified to use sample generators to provide the entity delays instead of providing them deterministically. The Sink entity was further modified in order to provide it with a `Sim_stat` object and measure the entity's throughput, event service time and utilisation:

```
...
class Sink extends Sim_entity {
  ...
  private Sim_stat stat;

  Sink(String name) {
    ...
    stat = new Sim_stat();
    stat.add_measure(Sim_stat.THROUGHPUT);
    stat.add_measure(Sim_stat.SERVICE_TIME);
    stat.add_measure(Sim_stat.UTILISATION);
    set_stat(stat);
  }
  ...
}
...
```

These modifications to the Sink's constructor are all that are required in order to calculate the specified measures' measurements. These will be included in the simulation's report once the simulation has completed.

Apart from defining the measures of interest for an entity, the modeller may select the events of interest and the exceedence proportion levels. In the case of state based measures the exceedence proportions represent the proportion of time that the entity spent above each level. For interval based measures, they represent the proportion of events that experienced intervals above each level. Utilisation poses some limitations compared to the measurements that can be obtained from other state based measures because SimJava considers entities to have a single server service type. If a multiple server service type is required it could be defined as a custom measure. Custom measures are covered in Section §4.3.

Selecting event tags of interest and specifying exceedence proportions are carried out with the `measure_for` and `calc_proportions` methods respectively. As an example of their use, the example presented previously is further modified:

```
...
class Sink extends Sim_entity {
  ...
  private Sim_stat stat;

  Sink(String name) {
    ...
    stat = new Sim_stat();
    stat.add_measure(Sim_stat.THROUGHPUT);
    stat.add_measure(Sim_stat.SERVICE_TIME);
    stat.calc_proportions(Sim_stat.SERVICE_TIME, new double[] {75, 125});
    stat.add_measure(Sim_stat.UTILISATION);
    stat.measure_for(new int[] {0});
    set_stat(stat);
  }
  ...
}
...
```

These modifications specify that the proportion of events whose service time was above 75 and 125 time units will be calculated, and also that observations will only be collected for events with tag 0.

In order to calculate the above measures, each statistics-gatherer stores all the observations of interest. Timing information, as previously described, is also stored with each observation in order to facilitate, among others, output analysis and variance reduction techniques. When a certain action occurs, such as an arrival or departure from an entity's queue, the relevant statistics-gatherer is notified. Depending on the measures and event tags specified by the modeller, observations are made for each measure affected.

In order to illustrate the process of observation collection, an `update` method of `Sim_stat` is presented. This method is present in several variations to accommodate different event types and is called automatically from within the kernel when an observation needs to be recorded. The `update` method presented here is called when an event arrives at an entity and makes observations for the entity's arrival rate and queue length:

```
void update(int type, int tag, double time_occurred) {
  ...
  if (is_selected(measure_names[ARRIVAL_RATE])) {
    ...
    observations[ARRIVAL_RATE].add(new Double(time_occurred));
    ...
  }
  if (is_selected(measure_names[QUEUE_LENGTH])) {
    ...
    if (!((time_occurred == 0.0) && (prev_time_queue == 0.0))) {
      observations[QUEUE_LENGTH].add(new double[] {queue_length,
                                                   prev_time_queue,
                                                   time_occurred});
      queue_length++;
      prev_time_queue = time_occurred;
      ...
    }
    ...
  }
  ...
}
```

The only action that can't automatically be determined by the simulation is the completion of an event's service. Such an action is specific to the entity's behaviour, as defined by the modeller. As such, when an event is considered to have completed all service at an entity, the modeller notifies the simulation by calling the `sim_completed` method. The modified Sink entity's `body` method is presented as an example:

```
...
class Sink extends Sim_entity {
  ...
  public void body() {
    Sim_event next = new Sim_event();
    int i = 0;
    while (true) {
      i++;
      if (i > 1000) break;
      sim_wait(next);
      sim_pause(delay);
      sim_completed(next);
    }
  }
}
...
```

The `sim_completed` method is used to update the entity's `Sim_stat` object and also to signal an event completion to `Sim_system`. For `Sim_system`, such information is required in order to determine a transient period or the simulation's run length if these are defined on the basis of event completions. This issue will be addressed in Chapter 5.

All measurements are obtained by calling relevant `Sim_stat` methods. The modeller need not call these explicitly since these are automatically used to produce the simulation's report. These methods can however also be used at runtime if measurements are required during the simulation's run. Calculation of each measurement proceeds according to the measure's type and depending on the start and end time provided to the calculation method. The `average` method of `Sim_stat` is presented in brief to illustrate the process of calculating a measure's sample average:

```
public double average(String measure, double start_time, double end_time) {
  int id = get_id(measure);
  int type = get_type(id);
  ...
  double result;
  if (type == RATE_BASED) {
    int[] indices = observation_count(id, start_time, end_time);
    ...
    if ((end_time - start_time) == 0.0) {
      result = 0.0;
    } else {
      result = ((indices[1]-indices[0])+1)/(end_time-start_time);
    }
  } else if (type == STATE_BASED) {
    ...
  } else {
    ...
  }
  return result;
}
```

4.3 Providing easily updateable custom measures

4.3.1 Issues concerning custom measures

In the same way that certain measures can be considered in all simulations, other measures apply only to specific experiments. These measures can be considered as *custom measures*, their meaning being defined by the modeller with respect to the experiment's details. An example of such a custom measure is an entity's loss rate.

Although custom measures are defined by the modeller, they can still be classified as rate, state or interval based. The only difference in this case is with state based measures. Default state based measures, namely queue length and utilisation, can be considered as *continuous state based measures* since the entity moves from state to state continuously. One difference that needs to be considered with custom state based measures is the fact that the entity does not need to be constantly in some state. These measures can be considered as being *non-continuous state based measures*. With regard to measure types this is the only difference that needs to be considered.

It is apparent that since custom measures are defined by the modeller, the times when observations for these measures occur must also be specified by the modeller. This naturally leads to some loss of automation with respect to observation collection. A key issue however is to make the process of collecting observations require minimum effort. Once the custom measures have been defined and their observations have been collected, measurement calculations can proceed automatically, as with default measures.

The issues that need to be considered for custom measures are how they will be specified and in what form they will be made available to the simulation.

4.3.2 Alternative approaches for providing custom measures

The statistical support currently present in SimJava could be considered for providing custom measures. The `Sim_accum` class can be used to collect observations and calculate measurements for non-continuous state based measures. This is however rather limiting since the modeller should be able to define any type of measure. An alternative approach for specifying custom measures is to provide an abstract class that could be subclassed, provided with the desired functionality, and used in the same way as the `Sim_stat` class described in the previous section. However, this approach would require a substantial amount of coding and effort from the modeller, which could prove to be as much error-prone as tedious. Alternatively, ready-made custom measure classes, depending on the measure's type, can be instantiated for each custom measure required. This could be a viable approach, but would hinder the accessibility of each entity's measurements in the case that numerous custom and default measures are defined.

The key point that needs to be identified concerning custom measures is that apart from needing to update them, issues concerning them are identical to those for default measures. Since the calculation of measurements is common for measures of the same type, custom or default, the main difference between the two cases is simply providing new methods for defining and updating custom measures. Therefore, the `Sim_stat` class defined for default measures can also be used for custom measures by providing new definition and update methods for each measure type. This appears to be by far the best approach, since code is re-used, and all measurements calculated by an entity can be made easily accessible to e.g. output analysis methods, in the form of a single object.

4.3.3 Approach chosen for providing custom measures

This last mentioned approach was the one pursued for SimJava. The `Sim_stat` class was extended to cater for custom measures by defining new measure-definition and observation-collection methods. In order to define a custom measure, the modeller needs to provide the name of the measure and its type. Three new observation-collection methods are also provided, one for each measure type. The form of the observations stored is identical to the form of default measure observations, a fact that makes the measurement calculation methods generally applicable. Furthermore, the same measurements that could be obtained for default measures are available for custom measures. This uniformity, apart from simplifying calculations, allows transient and termination conditions, as well as output analysis and variance reduction techniques, to be based on custom as well as default measures.

The example of Section §4.2.3 will be modified to illustrate how custom measures can be defined and updated. The Sink entity will be considered as having two processors that it may use to process incoming events. A single processor will be used for 40% of the events and both processors for the rest. A custom measure called "Processor use" will be defined to monitor the usage of the processors:

```
...
class Sink extends Sim_entity {
  Sim_random_obj prob;
  ...
  Sink(String name) {
    ...
    prob = new Sim_random_obj("Probability");
    add_generator(prob);
    stat = new Sim_stat();
    ...
    stat.add_measure("Processor use", Sim_stat.STATE_BASED, false);
    set_stat(stat);
  }
  public void body() {
    ...
    sim_wait(next);
    double before = Sim_system.sim_clock();
    sim_pause(delay);
    if (prob.sample() < 0.4) {
      stat.update("Processor use", 1, before, Sim_system.sim_clock());
    } else {
      stat.update("Processor use", 2, before, Sim_system.sim_clock());
    }
    ...
  }
}
...
```

4.4 Efficiency over detail

4.4.1 Efficiency issues concerning observation collection

In Sections §4.2 and §4.3 the concepts of default and custom measures were discussed. The last paragraph of Section §4.2.2 pointed out that the process of observation collection could be quite memory consuming. For long running and large simulations it may be the case that the memory requirements of the vast number of observations can not be accommodated by the system. On the other hand, it was also pointed out that since memory can be cheaply purchased in large quantities, the main motivation should be maximising the information gain from the simulation rather than minimising the memory consumption.

In light of these two issues, the optimum approach would be to provide both the options of efficiency and detail, allowing the modeller to decide which approach is more suitable. Furthermore it would be desirable to have the ability to mix efficiency with detail since the measurements obtained often have different levels of importance. Measures that form the primary interest of the simulation study can be set as fully detailed, while other, secondary measures, can be set as non-detailed but efficient.

As mentioned above, the main source of memory consumption is storing all the measures' observations. As such, efficiency can be enhanced by not storing observations but rather using them to recalculate affected measurements. This approach also improves the speed of simulations since the process of appending records to large data structures can be quite time consuming. The absence of the measures' observations however does come with a price since certain functionality such as graph generation will not be available.

4.4.2 Providing efficient measures for SimJava

In order to specify a measure as efficient the `set_efficient` method was implemented. This method is used on a per-measure basis, permitting the coexistence of efficient and detailed measures, and informs the entity's `Sim_stat` object that observations must not be stored for a certain measure. To illustrate the process of specifying a measure as efficient the Sink entity of our previous example is modified. All measures except its service time are set as efficient:

```
...
class Sink extends Sim_entity {
...
  Sink(String name) {
    ...
    stat = new Sim_stat();
    stat.add_measure(Sim_stat.THROUGHPUT);
    stat.add_measure(Sim_stat.SERVICE_TIME);
    stat.add_measure(Sim_stat.UTILISATION);
    stat.add_measure("Processor use", Sim_stat.STATE_BASED, false);
    stat.set_efficient(Sim_stat.THROUGHPUT);
    stat.set_efficient(Sim_stat.UTILISATION);
    stat.set_efficient("Processor use");
    set_stat(stat);
  }
  ...
}
...
```

Each efficient measure, depending on its type, is provided with a set of variables that hold the data required for calculating measurements. For example, if a rate based measure is set as efficient it will be provided with a simple counter that is incremented whenever an observation occurs. State and interval based measures require more data variables to store among others, exceedence proportion counters, minimum and maximum values as well as interval and state sums. Additional data is also stored to make sure that transient period data is correctly excluded from calculations and that the results obtained are identical to the ones that would be obtained if the measure was detailed. As a simple example an `update` method of `Sim_stat` is presented to illustrate the process of updating an entity's arrival rate:

```
void update(int type, int tag, double time_occurred) {
  ...
  if (is_selected(measure_names[ARRIVAL_RATE])) {
    if (is_efficient(measure_names[ARRIVAL_RATE])) {
      Long counter = (Long)data.get(measure_names[ARRIVAL_RATE]);
      counter = new Long(counter.longValue()+1);
      data.put(measure_names[ARRIVAL_RATE], counter);
    } else {
      ...
    }
    ...
  }
  ...
}
```

As mentioned in Section §4.4.1, the increased memory and speed efficiency comes at the price of sacrificing certain functionality. This is the functionality that requires the presence of all the measures' observations. The restrictions when using efficient measures are:

- Measurements for arbitrary time periods are not available.
- Exceedence proportions for levels other than the ones provided with the `calc_proportions` method can't be measured.
- No sample variance or standard deviation can be calculated.

- The batch means method of output analysis can't be used (see Section §4.5.1).
- A termination condition based on the confidence interval accuracy of a measure can't be used if batch means is used as a variance reduction technique (see Sections §4.5.1 and §5.3).
- The minimum-maximum method for transient period identification can't be used if the measure it is based on is efficient (see Section §5.2).
- No graphs can be produced for efficient measures (see Chapter 6)

4.5 Providing output analysis methods

4.5.1 Issues concerning output analysis

Output analysis methods are used to determine the quality of the obtained measurements. For each measure, a confidence interval of a specific level is calculated, which gives an estimate of the measurements' quality and accuracy.

The three most common output analysis techniques are independent replications, batch means and regeneration [8]. Of these, regeneration, although being able to provide the best results, will not be considered since it is not generally applicable and is very difficult to implement. The method of independent replications [8][13] repeats the simulation for a number of times, each time using different seeds in the entities' PRNGs. If only steady state analysis is desired, the transient period is estimated for each replication and is discarded. The remaining observations are used to form each replication's mean, and the means are subsequently used to calculate the total mean and mean variance. From these and the Student's t-quantile, based on the significance level and the degrees of freedom, the confidence interval half-width is obtained:

$$\overline{X} = \frac{\sum_{i=1}^{n} X_i}{n} \qquad S^2 = \frac{\sum_{i=1}^{n} \left(X_i - \overline{X}\right)^2}{n-1}$$

$$Var(\overline{X}) \approx \frac{S^2}{n} \qquad \hat{X} = t_{1-a/2,n-1} \cdot \sqrt{Var(\overline{X})}$$

where:

X_i	: The mean obtained from the i_{th} replication,
n	: The number of replications,
$\overline{X}$	: The total mean,
S^2	: The sample variance of the replication means,
$Var(\overline{X})$	: The mean variance,
$t_{1-a/2,n-1}$	: The Student's-t quantile for a significance level of *a* and *n*-1 degrees of freedom,
$\hat{X}$	: The confidence interval half-width

In the case of batch means [1][4][8][13], one very long simulation run is conducted and its observations are separated into batches. The mean for each batch is calculated and output analysis proceeds as in the case of independent replications. In this case, only one initial set of observations needs to be discarded and as such, batch means is a more efficient output analysis method compared to independent replications. However, successive batch means will tend to be correlated and as such experimentation with different batch sizes needs to be made in order to achieve the least correlation [1][13].

As in the case of measure definition and observation collection, the guidelines of ease of use and automation need to be upheld. This means that the modeller must be able to simply define an output analysis method and then allow the simulation to automatically apply it to the experiment's data. Parameters concerning the application of these methods, such as the number of replications or batches, need to be provided by default but also be overridable in order to accommodate experienced modellers.

Certain aspects of each output analysis method present challenges for their implementation. For independent replications, each replication must be carried out as a new simulation run. However, the observations obtained from previous replications need to be stored for later use. Furthermore, each entity needs to be restored to its original state for each replication. This task is far from trivial since the exact state of each entity is defined by the modeller and is not directly accessible to the simulation. Furthermore, each entity's PRNG needs to be re-seeded in order to produce different random sample sequences. For the batch means method, the problems of re-seeding generators, storing observations and re-initialising entities do not apply. However, in this case, an experimentation method needs to be specified and applied with which the optimum number of batches is selected.

Finally, additional functionality needs to be introduced in order to either compute or lookup the Student's-t quantiles for estimating the confidence interval half-width, and default values need to be selected for the parameters of each output analysis method.

4.5.2 Alternative approaches for performing output analysis

With respect to the time at which the output analysis method will be applied, there can be no alternatives. These methods may only be applied at the very end of the simulation run, once all observations have been collected. This limits the possible alternatives to the specific way in which these methods are applied.

In the case of independent replications, issues arise concerning the way in which the simulation's entities will be re-initialised. Basically, the best way to completely reset each entity would be to terminate the simulation and restart it. This however would require a large amount of data to be passed from one replication to another, in the form of collected observations and seed values used in generators, complicating the initialisation process. Furthermore, this would break the continuity of the simulation and the appearance of being a single unit of work, hindering at the same time other aspects of the simulation such as its animation. Another alternative would be to redefine the way in which simulations are defined, in such a way that a new instance of each entity would be made for each replication. This approach however would require changes in all current SimJava simulations.

A third, more appealing solution to this problem is to use Java's cloning facilities. When cloning an object, an exact replica is produced. Care is required however with mutable (updateable) objects since these will also need to be cloned. By following this approach, copies of all the simulation's entities are made before the first replication begins that are re-copied before each subsequent replication. Using this approach, exact copies of the entities are stored for each independent replication and most objects may be commonly used between the copies such as ports and generators.

An additional issue concerning independent replications is the re-seeding of the entities' generators. Jain in [8] suggests using the seeds that were last produced by each generator as the seeds for the next replication. This approach is by far the simplest since generators are common to cloned entities, and as such, no effort would be required to re-seed them. However, this would require the generators' seeds to be well-spaced enough to cater not only for a single run but also for several replications. Leaving the initial seeding up to the modeller could prove to be erroneous. Furthermore, automatically calculating the number of replications and thus the required seed spacing is some times impossible. This situation arises when independent replications are used as a variance reduction technique, in which case the number of replications to be performed is dynamically estimated based on the collected observations [13]. This issue is described in Section §5.3.

Another alternative for re-seeding the generators is to automatically have `Sim_system` re-seed them in each replication. This approach can be easily applied since the implementation of SimJava's new PRNG allows re-seeding. Moreover, the need to "reset" the simulation after each replication provides the opportunity to re-seed each entity's generators. Based on the root seed and seed spacing used for the original replication, new seed values would be produced and passed to the simulation's generators. This approach however, would require modifying the generator setup process in each entity. To be precise, each generator, after being created, would have to be added to the entity, in the same way that ports are currently added. This modification would require changes in existing SimJava simulations.

In the case of batch means, the main issue is the method for selecting the optimum number of batches. Two main alternatives exist in this case, the first one being to select the number of batches that gives the best confidence interval accuracy for the majority of measures, and the second one being to select the number of batches that minimises the serial correlation between successive batch means [1]. If the first approach if followed, the resulting confidence intervals could be under-estimated if successive batches are correlated. As such, determining the number of batches based on serial correlation appears to be the best approach.

A final point of consideration should be the method of calculating the t-quantiles necessary for estimating the confidence intervals. One alternative would be to have a pre-computed table of values, covering several confidence levels and batch or replication counts. A more flexible approach would be to calculate t-quantiles dynamically, a method that would cater for any confidence level and replication or batch count.

4.5.3 Approach chosen for performing output analysis

In the case of independent replications, the cloning approach was used to re-initialise the simulation's entities. Upon initialisation, a copy of all the entities is made and for each subsequent replication, the entities are copied back from their original copies. After each replication is complete, if the required number of replications hasn't been reached, the entities' `Sim_stat` objects are copied, their generators are re-seeded, and the simulation's counters and flags are reset. If all have completed, the collected `Sim_stat` objects from each replication are used to estimate confidence intervals for each specified measure.

In order to re-seed the entities' generators `Sim_system` generates a new seed for each one before each subsequent replication begins. The process of re-seeding is similar to the original generator seeding described in Section §3.3.3. In order to access the entities' generators, the modeller needs to add each one to the entity upon initialisation using the `add_generator` method as seen in the example of Section §4.3.3. To prevent incompatibility with existing simulations, if the generators are not added to their entities, they are simply not re-seeded. In this case, they are considered to be originally seeded to produce well-spaced sample sequences to cover all the replications.

In the case of batch means, the number of batches selected is the one that provides the least serial correlation for the majority of measures. The serial correlation is estimated using the method of jack-knifing [1]. In either case of output analysis, default values are specified for the number of replications (10) and the range of batches to be tested (5 to 20), as well as the confidence level to be used for estimating the measures' confidence intervals (90%).

In order to illustrate the use of output analysis methods, the FIFO queue example will be modified accordingly. The simulation is setup to use the method of independent replications, performing 5 replications and calculating confidence intervals of a 95% confidence level:

```
...
public class Queue {
  public static void main(String args[]) {
    Sim_system.initialise();
    Source source = new Source("source");
    Sink sink = new Sink("sink");
    Sim_system.link_ports("source", "enqueue", "sink", "arrival");
    Sim_system.set_output_analysis(Sim_system.IND_REPLICATIONS, 5, 0.95);
    Sim_system.run();
  }
}
```

The code that applies the selected output analysis method is too long and complicated to include at this point. However, the steps to apply each method can be clearly identified and easily understood. These steps are presented in table 4.1.

Independent replications	Batch means
1. Perform all replications. For each replication store the `Sim_stat` objects of all entities, reset the entities to their stored copies, re-seed their added PRNGs, and reset the simulation's flags and counters.	**1.** Perform simulation run.
2. Calculate replication means.	**2.** Calculate batch means for the set of batch counts provided and calculate their serial correlation.
3. Calculate total mean from replication means.	**3.** For the batch count that gives the least serial correlation recalculate the batch means.
4. Calculate confidence intervals using total mean, total standard deviation and appropriate t-quantile.	**4.** Calculate total mean from batch means.
5. Calculate other total measurements.	**5.** Calculate confidence intervals using total mean, total standard deviation and appropriate t-quantile.
6. Group simulation data.	**6.** Calculate other total measurements.
	7. Group simulation data.

Table 4.1: Steps to apply output analysis

Finally, each output analysis method, apart from estimating confidence intervals, is also responsible for grouping the simulation's data in a uniform and easily accessible manner. Such a grouping also occurs when output analysis has not been specified for the simulation. This task is performed at this point in order to facilitate future requirements such as report and graph generation.

4.6 Providing automatic report generation

4.6.1 Issues concerning report generation

Calculating measurements of interest is of no use if the results are not made available to the modeller. This is the reason why most simulation packages and tools generate reports to summarise the simulation's run and display the obtained measurements. Such a report needs to be also provided by SimJava. Furthermore, the process of generating this report should be automated without requiring any modeller effort. The only point where the modeller would be expected to provide information is in defining the level of detail required from the report.

The report generated by the simulation needs to contain all the information that could be desired by the modeller. This includes first and foremost the total measurements obtained. If an output analysis method is used, then the confidence intervals obtained should also be provided. Furthermore, in the case of independent replications or batch means, optional information could be the measurements obtained over each replication or over each batch. Finally, additional information that is commonly found in simulation reports are the seeds used in the simulation's generators. These seeds are required in order to recreate experiments.

Concerning SimJava in particular, animated versions of simulations are unable to store textual report files on disk due to applet sandboxing restrictions. However, it would be a limitation not being able to provide a report for animated simulations, even though report generation would not be a key issue for animated versions. The choice of having a report or not in these situations should be provided to the modeller. As such, a way of presenting the simulation's report needs to be found in order to enable simulation applets with the same reporting capabilities that are available to standalone versions.

4.6.2 Alternative approaches for report generation

No major design alternatives exist in the case of report generation. Reports may only be generated once the simulation has completed, and contain all the measurements and information that the modeller requested. The only issue that could be discussed is the way in which control over the level of detail is accomplished and how reporting would work for animated simulations.

Concerning the level of detail, three types of information may be included in a report. The first one is the set of total measurements obtained and the confidence intervals if an output analysis method has been used. The second one applies if independent replications or batch means have been selected as an output analysis method and consists of displaying measurements from each individual replication or batch. The third type of information is the set of seeds used to initialise the entities' sample generators. These three types of information, with the first one being the default, should be provided as options for the modeller. The approach of optionally including additional information is preferable to providing a report with every possible detail since overwhelming information can be as bad as no information at all.

Concerning the provision of reporting for animated simulations, the main issue is how to make a report available. Again the best approach seems to be to allow the modeller to decide whether a report is required or not. This is the case since many animated simulations are constructed only for educational, presentation, and debugging purposes, and as such, do not require the generation of a report.

4.6.3 Approach chosen for report generation

In the new version of SimJava, a report is generated automatically at the end of the simulation's run. This report contains as default the total measurements and confidence intervals calculated for all the specified measures. Further options are, as previously described, each replication's or batch's measurements, if applicable, and the seeding information of the experiment. These options can be easily specified by the modeller upon simulation setup. As an example, the FIFO queue simulation is presented with a definition of the report's detail. The flags used in the `set_report_detail` method specify that the report should not contain sample measurements from each replication, and that the seeds used to initialise the entities' PRNGs should be included:

```
...
public class Queue {
  public static void main(String args[]) {
    Sim_system.initialise();
    Source source = new Source("source");
    Sink sink = new Sink("sink");
    Sim_system.link_ports("source", "enqueue", "sink", "arrival");
    Sim_system.set_output_analysis(Sim_system.IND_REPLICATIONS, 5, 0.95);
    Sim_system.set_report_detail(false, true);
    Sim_system.run();
  }
}
```

Apart from the total measurements, sample measurements and generator seeds, the report is enriched with general information regarding the simulation. This information consists of the following:

- The version of SimJava used.
- The date the simulation was run.
- The start and end time of the simulation.
- The total simulated time.
- The total transient time.
- The total steady state time.
- The transient condition used.
- The termination condition used.
- The output analysis method used. Furthermore if one was used:

- The confidence level used.
- The number of replications performed (for independent replications).
- The number of batches used (for batch means).

An extract of a simulation's report file is presented as an example:

```
############################################################
#                                                          #
#                   SIMULATION REPORT                      #
#                                                          #
############################################################

Version: SimJava 2.0

Simulation date:         September 3, 2002
Simulation start time:   12:00:30 PM BST
Simulation end time:     12:00:46 PM BST

############################################################
#            Overall simulation run information            #
############################################################

Total simulated time:    1996905.7098756502
Total transient time:    500346.66026383184
Total steady state time: 1496559.0496118185
Transient condition:     100000.0 units of elapsed simulated time
Termination condition:   2000 event completions at Processor
Output analysis method:  Independent replications
Confidence level:        0.9
Replications performed:  5

############################################################
#       Total measurements and confidence intervals        #
############################################################

------------------------ Processor -------------------------

- Service time

Total mean:          110.51708278810608
Interval low bound:  110.26798013047872
Interval high bound: 110.76618544573344
Interval half width: 0.24910265762736117
Accuracy ratio:      0.0022539742394843913
Mean variance:       0.013653522115406224
Mean std deviation:  0.11684828674570383
Total maximum:       147.16099034261424
Total minimum:       72.48955239466159
Total average exceedence proportions:
    0.0 < Service time <= 50.0 : 0.0
    50.0 < Service time <= 100.0 : 0.1315
    100.0 < Service time <= 150.0 : 0.8685
    150.0 < Service time <= 200.0 : 0.0
    200.0 < Service time : 0.0

...
```

For animated versions, the modeller has the ability to specify whether or not a report should be added to the animation's applet. This is done by overriding the `anim_output` method in which the modeller may decide to include the simulation's report and/or the simulation's messages. If the modeller requires a report to be produced the applet's GUI is appropriately extended. A text area is added beneath the animation panel in which the report is placed once it has been generated.

5 Transient period and run length definition

5.1 Introduction

So far we have covered the first two goals of this project. Chapter 3 focused on the improvement of SimJava's sampling methods and Chapter 4 discussed how sophisticated statistical support could be provided to simulations. This chapter will focus on the goal of improving the way in which the transient period and run length of simulations are defined.

Conditions for determining a transient period and the simulation's run length currently need to be manually checked by the modeller. Moreover, these conditions are limited to checking whether a certain time period has elapsed. It is safe to assume that in many situations it would be more natural to define such periods based on other conditions such as the number of events completing service.

5.2 Providing improved transient period definition

5.2.1 Issues concerning transient period definition

Specifying a transient period is important in situations where the modeller is interested in steady state analysis. In these cases, the system needs to be allowed to progress for a certain time before steady state is assumed to have been reached. The transient period may be explicitly specified, either by determining a certain time period or by specifying a certain number of event completions, or put under the simulation's care and estimated automatically. Both of these options need to be provided in the new version of SimJava.

As mentioned previously, SimJava currently specifies the transient period on the basis of a certain time period having elapsed. Additional conditions that could be provided are those based on certain numbers of event completions, and the automatic truncation of initial data. When defining a transient condition based on event completions, the simulation needs to be somehow notified when events complete service. On the other hand, the method of truncating initial data can be automatically applied by using the *minimum-maximum method* [8] on the collected observations. Using this approach, observations are discarded until the first observation that is neither the minimum nor the maximum of the remaining ones is encountered.

As mentioned previously, simulations may be focused on transient as well as steady state analysis. Furthermore, certain systems never actually exhibit a steady state and as such, no specific transient period can be defined. In such situations, the modeller must have the ability to define simulations without any transient period. The modeller must therefore be provided with an easy and clean way of defining or not a transient period condition and, in the case that one is specified, with a way of easily defining its parameters. Following the definition of such a condition, the kernel should automatically undertake the process of checking it, acting accordingly if it is satisfied. This should be done without any additional coding or effort on behalf of the modeller.

5.2.2 Alternative approaches for providing transient period definition

The methods of specifying a transient condition are the ones highlighted in the previous section. As such, the major issue that remains is how will the conditions be applied. The effect that a transient period has on a simulation is basically to omit a certain number of initial observations from the steady state measurements' calculation. Several approaches for this task can be identified.

One approach to achieve this would be to start collecting observations only once the transient period has elapsed. Alternatively, if measurements are recalculated with each observation, the measurements and their related counters could be reset once the transient condition has been satisfied. This approach seems preferable to the first one since it would make sample measurements available to simulations even before steady state has been reached. Bearing this in mind, the latter approach was selected for any measures that have been defined as efficient. Recall that these measures focus on efficiency by avoiding the storage of all observations but rather using each one to recalculate affected measurements. Efficient measures were discussed in Section §4.4.

A third approach would be to collect all observations, and in the end select only those that are contained within the steady state period. This approach although increasing storage requirements maximises the information obtained from the simulation. As discussed in Section §4.4, a wide range of functionality requires the presence of all the simulation's observations. This includes batch means as an output analysis method, using the minimum-maximum method for transient period estimation, and the generation of detailed graphs to study the progress of the simulation's measurements. Furthermore, this approach simplifies the process of collecting observations by the simulation's `Sim_stat` instances. For these reasons, this was the approach selected for detailed measures.

5.2.3 Approach chosen for transient period definition

As mentioned in the previous section, the effect of the transient period depends on whether or not each measure has been defined as efficient or detailed. In both cases all observations are processed, either being used to recalculate measurements, or stored for later calculations. For efficient measures, once the transient condition has been satisfied all measurements and related counters are reset to values that will ensure correct, steady state results. For detailed measures the transient period is considered at the end of the simulation, when the methods for calculating steady state measurements are invoked with the steady state start and end times. Identifying observations that are contained within this period is made possible by the fact that all observations are stored along with their timing information.

The transient condition types provided are:

- A certain time period having elapsed. In this case, observations within this time period are not considered in calculations. To apply this condition, the modeller specifies a certain time at which the system is considered to have entered steady state.
- A certain number of event completions. With this condition, the time of the last event to be considered as being in the transient period is recorded and observations up to that time are disregarded as in the previous method. To select this method, the modeller needs to specify an entity of interest and an event type for which the condition will apply. Event

completions are signalled to `Sim_system` by the `sim_completed` method as seen in the examples of Section §4.2.3.

- Automatic truncation of initial data based on the minimum-maximum method. To use this method, the modeller needs to specify an entity's measure upon the observations of which the method will be applied. The measure specified must be state or interval based in order to be able to calculate minimum and maximum values, and also be defined as detailed to have all its observations available. This method is quite crude and may often produce underestimated transient periods [8], but nevertheless remains the only implemented method that does not require the definition of an explicit transient period.
- None. In this case no transient condition is specified and all the observations are used to perform calculations.

Defining the transient period of a simulation is achieved using the `set_transient_period` method. Several variations of this method are available to cater for each condition's parameters. To illustrate this method's use, the example we have used up to this point will be further modified to include a transient period condition. This condition is based on the elapsed simulated time specifying that steady state will be reached after 50,000 time units:

```
...
public class Queue {
  public static void main(String args[]) {
    Sim_system.initialise();
    Source source = new Source("source");
    Sink sink = new Sink("sink");
    Sim_system.link_ports("source", "enqueue", "sink", "arrival");
    Sim_system.set_output_analysis(Sim_system.IND_REPLICATIONS, 5, 0.95);
    Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, 50000);
    Sim_system.run();
  }
}
```

As mentioned previously, the transient condition is checked automatically by the kernel at each clock tick. This is achieved with the `check_conditions` method that is also used to check the simulation's termination condition, covered in Section §5.3. For our current simulation the relevant part of `check_conditions` is as follows:

```
public static boolean check_conditions() {
  ...
  switch (term_condition) {
    ...
    case NONE:
      switch (trans_condition) {
        ...
        case TIME_ELAPSED:
          if (clock >= initial_trans_time) {
            in_steady_state = true;
            ...
          }
          break;
        ...
      }
      break;
  }
  ...
}
```

Note that the termination condition is considered to be `NONE` because no explicit termination condition was specified. How a termination condition is defined is covered in Section §5.3.3.

5.3 Providing improved run length definition

5.3.1 Issues concerning run length definition

Many of the issues concerning termination conditions are similar to those for transient conditions. The range of termination conditions currently offered needed to be extended and their testing needed to be performed automatically, without modeller effort.

Termination conditions defined in terms of time periods or event completions are similar to the corresponding ones used for estimating a transient period. However, in the case of defining a suitable termination condition, the quality of the measurements obtained may be of prime importance. The modeller may not want to explicitly terminate a simulation run, but rather be interested in obtaining a good enough estimation of some measurement. In this case, output analysis methods are used as variance reduction techniques and prolong the simulation's run length as they see appropriate. The simulation terminates once an accurate enough confidence interval of a given level is obtained for a measure.

To apply variance reduction, sequential methods are proposed based on independent replications [13] or batch means [1]. In the case of independent replications, a set number of initial replications is performed and the number of additional ones required is estimated based on the obtained accuracy. In the case that batch means is used as the variance reduction technique, an initial set of observations is obtained and then batched. If the serial correlation of the batch means is too high, additional observations are collected. If correlation is low but the confidence interval accuracy obtained is not sufficient, additional observations are collected as in the previous case. Regardless of the variance reduction technique used, the confidence interval accuracy is defined as the ratio of the interval's half-width to the total measurement's mean.

5.3.2 Alternative approaches for run length definition

As in the case of transient conditions, the termination condition specified should be automatically tested without modeller effort. One issue that needs to be addressed however is the fate of events still remaining in the event queue. Currently in SimJava, the simulation only terminates when no more events are present in the event queue. Satisfying the termination condition essentially causes the entities to stop producing events. However, many events may still remain in the event queue, especially when the simulation contains many entities and has been running for a long time. As a result the simulation could continue for a considerable period after the termination condition has been satisfied. In order to avoid this situation, the best approach would be to discard the events remaining in the event queue rather than allowing them to be processed.

Another issue that needs to be addressed is how to notify entities that the simulation has completed, and how they should subsequently react. Entities that are continuously active within the simulation should check with `Sim_system` to see if they should continue. Since all entities will be inactive at the time the condition is satisfied, either holding or waiting for events, they need to be reactivated. Caution is required at this point however since entities must be handled in a different way depending on whether the simulation is completing, in the case of a single run, being extended, in the case of batch means performing variance reduction, or being reset, in the case of independent replications.

If the simulation run is to be extended, the entities must continue from the exact point where they became inactive. However, if independent replications are used, the entities must terminate and be reset. One alternative here is to activate inactive entities and allow them to proceed normally until they check with `Sim_system` and exit. This approach however could generate additional events and possibly cause unwanted observations to be collected. Furthermore, these entities may again become inactive, waiting for events that will never arrive. In light of these facts, the only viable approach is to reactivate inactive entities, allowing them to proceed until they check with `Sim_system`, but at the same time preventing their runtime methods of having any effect.

One final issue is to consider when the termination condition starts to apply. It may be considered from the beginning of the simulation, or only after steady state has been reached. The latter approach would require that the transient condition be satisfied before the termination condition starts to apply. This issue may therefore be considered as a need to define the relationship between the two conditions. Since both alternatives could be called for, the best approach is to allow the modeller to specify the desired relationship between the transient and termination conditions.

5.3.3 Approach chosen for run length definition

As with the transient condition, the termination condition is automatically checked by `Sim_system` at each clock tick. Once the condition is satisfied, the selected output analysis method is applied and the simulation's future is determined. If independent replications are used and additional replications are required, the entities are allowed to terminate, their `Sim_stat` objects are stored, and they are subsequently reset along with the simulation. Otherwise confidence intervals are generated for the entities' measures. Alternatively, if batch means is used for variance reduction and the simulation is to be extended, the entities are simply restarted from the position at which they became inactive. If the obtained accuracy and serial correlation of the batch means are acceptable, the simulation completes by generating confidence intervals for the simulation's measures.

The termination conditions that are available to the modeller are:

- A certain time period having elapsed. As in the case of the corresponding transient condition, once the specified time period has elapsed, the current simulation run is terminated. To use this method the modeller needs to specify a certain time at which the simulation will complete.
- A certain number of event completions. Similarly, in this case, the simulation will run until a certain number of event completions have been observed at an entity. In order to do this the modeller must specify an entity and a specific event type. `Sim_system` is notified of an event completing service in the same way that it would for the corresponding transient condition.
- A certain level of accuracy obtained for a measure. This is the most sophisticated termination condition. In this case, the simulation will terminate only once an accurate

enough confidence interval has been obtained for some measure. Accuracy, as previously mentioned, is defined as the ratio of the interval's half-width to the total mean. In order to use this termination condition, the modeller needs to specify an entity's measure for which the interval will be estimated, the confidence level to be used, the minimum desired accuracy, and an output analysis method to be used for variance reduction. In this case, no additional output analysis method may be defined and confidence intervals for all measures are produced based on the parameters of the specified variance reduction method.

- None. Simulations that run indefinitely would normally constitute a modelling error. However, in the case of SimJava it could be possible to consider an animated simulation that runs until stopped, for presentation purposes. Furthermore, this condition type is required for compatibility with existing SimJava simulations.

A termination condition is specified using the `set_termination_condition` method. To illustrate its use, the FIFO queue simulation will be modified to define a termination condition. Recall how checking the condition was performed manually in the Source and Sink entities. These manual checks are substituted by a call to the `running` method of Sim_system that checks to see whether the termination condition has been satisfied. The termination condition itself is centrally defined along with the transient condition:

```
class Source extends Sim_entity {
  ...
  public void body() {
    while (Sim_system.running()) {
      ...
    }
  }
}
class Sink extends Sim_entity {
  ...
  public void body() {
    while (Sim_system.running()) {
      ...
    }
  }
}
public class Queue {
  public static void main(String args[]) {
    ...
    Sim_system.set_transient_condition(Sim_system.TIME_ELAPSED, 50000);
    Sim_system.set_termination_condition(Sim_system.EVENT_COMPLETIONS,
                                     "sink", 0, 1000, false);
    Sim_system.run();
  }
}
```

Note that the boolean flag of `set_termination_condition` is set to false. This dictates that the termination condition will apply only after steady state has been reached. The code of `check_conditions` that applies to this example follows:

```
public static boolean check_conditions() {
  ...
  switch (term_condition) {
    case EVENTS_COMPLETED:
      switch (trans_condition) {
        ...
        case TIME_ELAPSED:
          if (include_transient) {
            ...
          } else {
            if (in_steady_state) {
              if (term_event_counter >= term_count) {
                simulation_completed = true;
              }
            } else {
              if (clock >= initial_trans_time) {
                in_steady_state = true;
                ...
              }
            }
          }
          break;
        ...
      }
      break;
    ...
  }
  ...
}
```

The case of animated simulations also provides an additional issue concerning the termination condition. The modeller may decide to terminate a simulation by stopping it before its termination condition has been reached. When such an action takes place the simulation is considered to have completed normally and the report generation proceeds as usual. However, caution needs to be applied since the simulation may have been stopped too prematurely to apply the selected output analysis method. In such cases, no output analysis is performed. A similar situation also occurs when the transient condition has not been satisfied at the point where the simulation was stopped. In this case, no transient period is considered for the remaining calculations and report generation.

As a final note, the events still remaining in the event queue once the termination condition has been satisfied are discarded. The motivations for this approach are highlighted in the final paragraph of Section §5.3.2.

6 Provision of graphical output analysis

6.1 Introduction

In the previous three chapters we discussed the first three major goals of this project. Chapter 3 covered the goal of improving SimJava's sampling methods, Chapter 4 continued to discuss how SimJava's statistical support was augmented, and finally, Chapter 5 discussed how SimJava's ability to define transient and termination conditions was enhanced. This chapter focuses on the last major goal of this project, providing sophisticated and detailed graphical output analysis.

As previously mentioned, simulations are built to study the behaviour of systems. This is accomplished by obtaining and studying the results produced by the simulation run. These results consist of general information concerning the definition of the experiment and the measurements that were calculated for the simulation's entities.

Most simulation packages make the experiment's results available to the modeller in a textual form: the simulation's report file. Automatically producing a detailed and informative report file was one of this project's sub-goals and was discussed in Section §4.6. It is often the case however that a graphical presentation of the simulation's results is desired and considered more useful than a textual presentation of obtained measurements. Such a graphical presentation of results provides the modeller with a user-friendly, flexible and detailed account of the simulation's results.

6.2 Issues concerning graphical output analysis

One of the major goals of this project was to enhance SimJava's statistical support. As mentioned in Sections §4.2.1 and §4.2.2, one of the issues concerning this goal was the form in which observations were collected for the simulation's measures. The approach selected was to enable the modeller with the ability to define both efficient and detailed measures. Efficient measures recalculate measurements with each observation, whereas detailed measures store all the observations for later calculations. Although detailed measures present much greater memory requirements than efficient ones, the motivation for providing them was that they maximised the information obtained from the simulation. One of the main aspects of this information gain was to enable graphical output analysis. As such, the graphs generated from simulations need to be as flexible and informative as possible in order to justify the measures' increased overhead.

The presentation of results in graphical form presents the potential of greatly increasing SimJava's reporting capabilities. In order to fully take advantage of this, the graphical output must contain at least all the information that is made available through the experiment's textual report. General information, such as the simulation's run length, must be presented along with detailed information concerning the entities' defined measures. In any case, the information gain must be at a maximum.

The main issue concerning the production of graphical output is the actual form this output will take. General simulation information can be easily presented in a manner similar to the one used in the report file. The main aspect however of the graphical output will be information regarding the entities' defined measures. For each measure, a graph must be produced that will inform the modeller of the measure's progress over the simulation's run length. Sample and total measurements must be easily accessible and, if possible, presented graphically. The graphs must also be easily modified to present additional information such as confidence intervals, the transient period, and the steady state behaviour of the measure. Finally, the ability to make and store annotations should be present since comments made on graphs by expert modellers are often as useful as the data itself. Any form of graphical output would be incomplete if it lacked an easily accessible annotation facility.

One of the major motivations of this project was to free the modeller from tedious and often error-prone tasks such as observation collection. The way in which this was achieved was by maximising automation wherever possible and providing powerful but easy to use functionality, allowing the modeller to concentrate on the modelling aspects of the simulation. With this in mind, it is clear that the generation of graphical output must not hinder the modeller from more important tasks. Generation of the graphs must be done automatically and without additional modeller effort. This is one of the major problems with `simdiag`, an existing SimJava package whose purpose is to provide graphical output for simulations. The usefulness of `simdiag` is limited partly because it may be used only in certain simulations, but mainly because of the complexity involved when using its graphs. The use of `simdiag` was therefore reserved mainly for SimJava experts, a requirement that could be considered quite limiting. The graphical output produced by the new version of SimJava must be applicable to any simulation and be easily available to all modellers regardless of their skill level and programming proficiency.

One final point of concern is the storage requirements and time overhead that the graph generation will incur. To maintain a maximal level of flexibility all the measures' observations need to be available to the graphs. If these graphs are to be stored on disk it is clear that a method of limiting the storage requirements of the observations must be applied. Finally, the time overhead that will be introduced to the simulation must be limited. It is apparent that in the case of long running simulations with many entities, measures and observations, the time required to produce the graphs could be considerable. It is clear that the delay introduced by the generation of graphs should be minimised and not present a significant overhead for the simulation's completion.

6.3 Alternative approaches for graph support

The first important issue is the type of information that the graphs will display. As previously mentioned each graph will correspond to an entity's measure. Furthermore, the graphs will be plotted on the basis of time. One alternative for the content of the graph is to display the observations as they were collected over time. Such graphs can be useful in identifying transient periods but are otherwise limited with regard to the information that the modeller can obtain from them. A more useful alternative for the graphs' content is to display the measures' sample average at each point in time based on the relevant observations. Such graphs can be used to study trends and to examine the measures' progress as the simulation advanced. Furthermore, by displaying the sample average at each point the modeller is informed of what the simulation's results would be for a wide range of run lengths. This greatly increases the level of information that is available compared to the textual report that presents results only for the entire (steady state) run length. Clearly, displaying each measure's sample average in the graphs is the best approach considering that information maximisation is the main objective.

Another issue that presents several alternatives is the time at which the graphs will be generated. One option is to calculate the graphs' plot as the simulation progresses, by storing the sample averages at certain points in time. The benefit of this approach is that the graphs will be instantly available once the simulation completes. However, several problems would arise if this option were to be adopted. First of all, since this method would require performing numerous calculations at many points in time, a significant overhead would be incurred especially if the graphs' granularity is high. Furthermore, added complexity would be introduced to the simulation's kernel in order to calculate the graphs' plots. The main problem with this method however is the lack of flexibility that the graphs would be burdened with. Since each plot would be pre-computed, no modifications such as zooming would be possible. These limitations require that the graphs be produced after the simulation has completed. Such an approach would minimise the overhead to the run length and would allow maximal flexibility.

The next issue that arises is how to present these graphs to the modeller. One option would be to build and present these graphs automatically upon simulation termination. This approach however could considerably extend the simulation's run length. The best approach would be to separate graph generation from the actual experiment allowing inspection if desired, but otherwise minimising the introduced overhead. To achieve this, the measures' observations need to be stored on disk and made accessible to the modeller by means of a separate utility that would access them to generate the graphs. Since graph annotations will be possible, storing the graphs will eventually be a necessity. As such, the option of storing the graph data on disk seems to be the most appealing.

To greatly simplify the graph data generation and storage, Java serialisation could be used. All the required information from the simulation can be gathered in one object and serialised to disk. Furthermore, to reduce storage requirements, Java's compression capabilities could be used on the serialised data, therefore preventing the possibly serious problem of high storage requirements. The serialised and compressed data can be read with the same ease by the graph viewing utility, which will subsequently generate each graph.

The approach of storing the graph data on disk does however present one problem. Animated versions of simulations will be unable to produce graphical output due to applet sandboxing restrictions. This problem can be considered minor if the purpose of animated simulations is taken into account. Animated simulations are not built to extensively study a system but rather for debugging and presentation purposes. If a modeller were interested in studying a system in depth and running long experiments, he would define the simulation as a standalone application, free of animation overheads. With this in mind and the fact that graph generation and examination is used for the detailed analysis of systems, it can safely be assumed that when graphs are desired, the simulation will be free of sandboxing restrictions i.e. run as an application. This limitation can be therefore considered as insignificant.

6.4 Approach chosen for graph support in SimJava

The approach used to provide graph support is the one highlighted in the previous section. The relevant data is collected at the end of the simulation run, compressed, and stored on disk. Subsequently a separate graph viewing utility is used to open the graph data and generate the graphs. The graphs display the sample average for each measure and may be modified to include additional information. Finally, graphs may be annotated, stored and opened at a later time.

In order to generate graphical output from a simulation only a single line of code needs to be introduced to an experiment. The FIFO queue example will be extended to invoke the `generate_graphs` method of `Sim_system` that informs the kernel that graphs are to be produced:

```
public class Queue {
  public static void main(String args[]) {
    ...
    Sim_system.generate_graphs(true);
    Sim_system.run();
  }
}
```

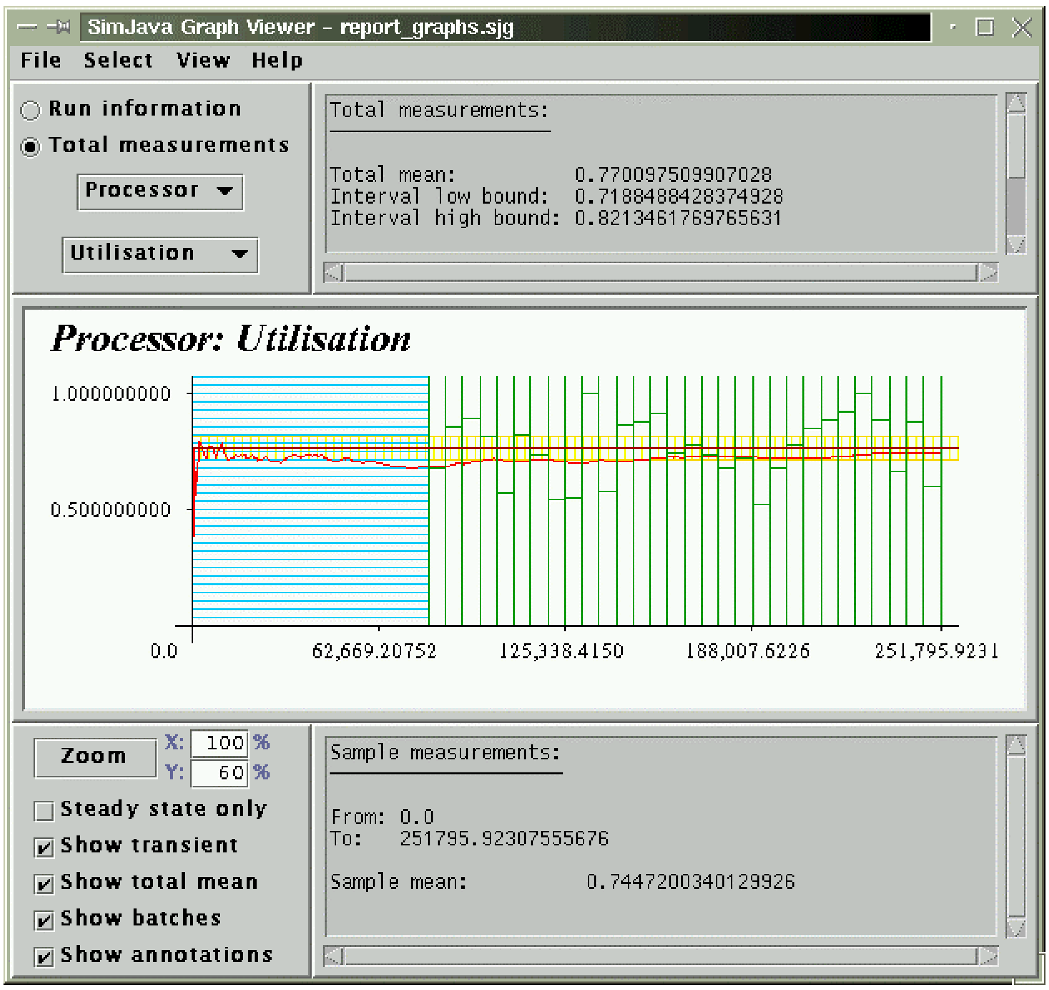
To produce the graphs' data, the measures' observations need to be collected and stored on disk. To achieve this, each entity is accessed to obtain its `Sim_stat` instance. This object contains not only the observations for each of the entity's measures but also all the functionality required for producing relevant sample measurements. These `Sim_stat` instances are collected along with general run information such as the total run time, the transient time, the output analysis method used, the total measurements etc.

The way in which this data is collected varies depending on the method of output analysis that is used in the simulation. In the case of batch means, additional information in the form of the number and size of the batches is also collected. For independent replications, apart from the total measurements, information is collected for each individual replication. This information consists of each replication's total and transient time as well as copies of the entities' `Sim_stat` instances that resulted from each replication. This data is not collected at the very end of the simulation, but as the run progresses. It is used to perform output analysis and to generate the simulation's report file. Finally, it is stored in order to be accessed by the graph viewing utility.

At the end of the simulation, the graph data is available in a single object. This is stored on disk using Java's serialisation, as previously discussed. Before actually storing the serialised object on disk, compression is also introduced. At this point Java's compression capabilities are utilised and the serialised object is compressed using GZIP. The final result is a ".sjg" (**S**im**J**ava **G**raphs) file that contains the compressed, serialised data that is required by the graph viewing utility to produce the graphs. This process is illustrated next by presenting the `generate_graphs` method of `Sim_system` that is internally invoked to store the graph data:

```
private static void generate_graphs() {
  ...
  ObjectOutputStream output = null;
  ...
  output = new ObjectOutputStream(new GZIPOutputStream(
                                  new FileOutputStream(graph_file)));
  System.out.print("Generating graph data...");
  output.writeObject(run_data);
  output.flush();
  output.close();
  System.out.println("finished.");
  ...
}
```

The **S**im**J**ava **G**raph **V**iewer (SJGV) is a utility used to open ".sjg" files and produce the simulation's graphs. This utility is used to display graphs along with controls that specify their detail or additional information to be displayed. The options available vary depending on the output analysis method that was used to produce the results. The following screenshot displays the viewer's GUI for a simulation using batch means for output analysis:



```
Total measurements:

Total mean:           0.770097509907028
Interval low bound:   0.7188488428374928
Interval high bound:  0.8213461769765631
```

```
Sample measurements:

From: 0.0
To:   251795.92307555676

Sample mean:          0.7447200340129926
```

Figure 6.1: The SimJava Graph Viewer's interface for an experiment using batch means

The viewer's GUI consists of five major components:

- The *selection panel* (top-left). This panel presents the user with options regarding the selection of a measure to display and the contents of the general information panel. The simulation's entities are made available to the modeller as well as the currently selected entity's measures. By selecting a measure from the measure list, the corresponding graph is generated and displayed. Similarly, when an entity is selected from the entity list, the measure list is updated and the graph is set to display the selected entity's first measure. In the case that an output analysis method was used, this panel also provides the ability to select the information displayed in the general information panel. The first option is to display the simulation's general information. This consists among others of the date and time the simulation was executed, the total simulated and transient time, the transient and termination conditions used, and information specific to the selected output analysis method. The second option is to view the currently selected measure's total measurements such as the total mean and its confidence interval.

- The *general information panel* (top-right). This panel is used to present general information concerning the experiment, or the total measurements of the currently selected measure. As mentioned above, which information is to be displayed is managed by the selection panel. The default is to present the simulation's general run information. In the case of no output analysis method having been used, this is the only option.

- The *graph panel* (centre). This panel is the main element of the viewer. It displays the currently selected measure's sample average as the simulation progressed. The x-axis represents the simulated time and the y-axis, the value of the sample average. The information displayed on the graph may be modified by the graph controls' panel.

- The *graph controls panel* (bottom-left). This panel contains options concerning additional information displayed on the graph as well as the graphs' level of detail. The options available for all simulations are the zoom control, which enables zooming on either axis, displaying only the steady state, displaying the transient period, displaying the total mean and its confidence interval, and turning the annotations on or off. In the case that batch means was used as an output analysis method, an additional option is provided which the modeller may select to view the batches and their means. If independent replications were used, two additional options are available. The first one is to select a specific replication and the second one to display the plots of all the replications on one graph. In this final case, the sample information panel is modified to display a legend for the plot. Moreover, several options are made unavailable such as displaying annotations or the transient period since these have replication-specific meaning.

- The *sample information panel* (bottom-right). This panel presents the currently selected measure's sample measurements. By default, the measurements displayed correspond to the entire run length of the simulation including transient period observations. If the option to display only the steady state is selected from the graph controls panel, the sample information panel is updated to display the steady state sample measurements. The measurements displayed in each case depend on the measure's type and whether or not the modeller has specified exceedence proportions to be calculated. In addition, this panel is used to display sample measurements up to a certain point selected by the modeller. This is done by clicking on the graph, upon which time this panel is updated to present the sample measurements up to the clicked point in time. The graph is also suitably updated to display the clicked point in time and the corresponding sample average. The selection is cleared when the modeller clicks outside the graph's bounds.

As a sample of the graphs that the viewer can produce four graphs are presented in figure 6.2. These graphs correspond to a simulation that was run with different output analysis methods to produce varying graphs. In all cases the utilisation of a "Processor" entity is displayed. For the top-left graph no output analysis method was used. For the top-right graph a termination condition was used that performed variance reduction using batch means. For the bottom-left graph independent replications were used and the results of the first replication are presented. Finally, for the bottom-right graph the data from the same simulation was used but the combined results from all the replications are displayed.

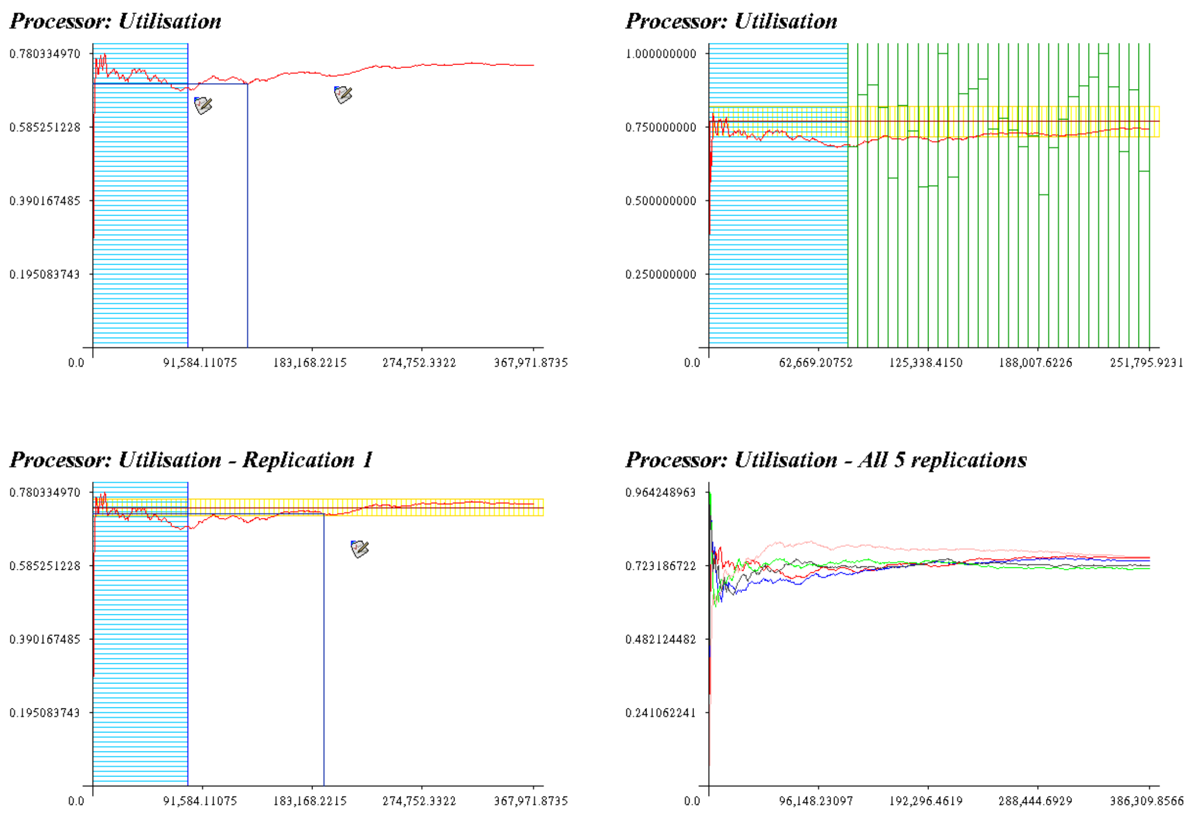


Figure 6.2: Sample graphs produced by the SimJava Graph Viewer

The main challenge with the graph viewing utility was the production of each graph's plot. The utility would have to produce graphs for any possible simulation but at the same time provide the modeller with commonly shared functionality. In order to produce the currently selected measure's plot, the relevant entity's `Sim_stat` instance is used. The measure's observations are accessed to obtain the minimum and maximum sample averages. These form the bounds for the y-axis. The bounds for the x-axis can be easily defined by the total run length and the start time which could either be time 0.0 or the transient period end. Using these bounds and the relevant graph pixel bounds, each time value and sample average could be scaled and accurately represented on the graph. This is possible by means of *normalisation*, with which a scaled pixel value (*valPixel*) can be obtained from a value (*val*) as such:

$$valPixel = \frac{val - min\,Val}{max\,Val - min\,Val} \cdot (max\,Pixel - min\,Pixel) + min\,Pixel$$

By inverting this formula, the value of a specific pixel can be retrieved. In the case that additional information was displayed on the graph, the minimum and maximum values could be modified. The time bounds could be modified when only steady state was displayed or when all the simulation's replications were presented in one graph. The sample average bounds could be affected by batch means, confidence interval bounds, and also other replications' minimum and maximum values. In any case, all values that could possibly affect the graph's bounds are inspected to detect the absolute minimum and maximum for each axis. Once the bounds have been identified, the above normalisation formula can be used to scale sample average[1] and time values.

To obtain the plot, the set of time values that corresponds to each x-axis pixel is identified. This set is used to obtain the sample averages and subsequently produce the graph (after scaling has taken place). However, the current calculation method used by `Sim_stat` to produce the sample average would be very inefficient for producing all the required averages since no previous computation is taken into account. In this case, the generality of the calculation method could be dropped in favour of increased efficiency. As such, a recursive calculation method was added to the `Sim_stat`, which produces the set of sample averages for a given set of time values. To further increase efficiency, this method was also used to obtain the minimum and maximum sample averages, to save going through the collected averages. Having obtained a set of (scaled) time values and their corresponding (scaled) sample averages, the plot could be easily produced by connecting each point. In the case that additional information is selected for presentation on the graph, relevant flags are set and relevant time or sample values are passed to the graph and used in its generation.

The final issue remaining is to paint the axes and relevant labels. The y-axis position is always fixed since no negative time values are possible. However, since negative sample averages are possible, the x-axis has to be positioned depending on the 0.0 point. The normalisation formula is also used for this purpose. Finally, to draw labels along the axes, the inverse normalisation formula was used to generate the values from a set of well-spaced pixels along each axis. The larger the graph's zoom level is, the more value labels will be placed on the graph's axes.

As mentioned previously, the graph viewer needed to cater for annotations. To make an annotation, the modeller is required to right click on the graph. At this point a dialogue is opened that allows the annotation to be made. Annotations are stored on the basis of the clicked time and value and as such can adapt to follow the graph when a different zoom level is specified for either axis, or whether or not steady state is displayed. To view annotations, the relevant option must be selected from the controls' panel and then the desired annotation is left clicked to display its text. At this point the modeller is also provided with the ability to update or delete the selected annotation.

To better illustrate the process of drawing a graph, the sequence of actions is presented in table 6.1. These actions correspond to the steps followed in `drawPlot`, the method of class

[1] Scaling of sample averages requires an additional slight modification to achieve the desired y-axis orientation of bottom to top rather than top to bottom used by Java. The inverse calculation of a sample average from a pixel was similarly modified.

`Graph` that is responsible for drawing each graph. This class is used by the graph viewer to represent each graph.

Steps to draw a graph	
1. Get the X-axis times	**6.** Draw the batches and their means*
2. Get the Y-axis values - Obtain minimum and maximum values	**7.** Draw the total mean*
3. Draw the confidence interval*	**8.** Draw the axes
4. Draw the transient period*	**9.** Draw the axes' labels
5. Draw the sample average plot	**10.** If the graph is clicked draw the clicked point
	11. Draw the annotations*
*: *If applicable and selected*	

Table 6.1: Steps to draw a graph

Without the ability to make annotations, the need to store graphs wouldn't exist since the graphs' data may not be altered by the viewer. However since the modeller may add annotations to graphs it is certainly the case that he, or some other interested party, will want to review them at a later time. For this reason the functionality of "Save" and "Save as" options was also added. Storing the graphs uses the exact same method of serialisation and compression used by `Sim_system` upon graph data generation. Along with these options, the ability to load at runtime a stored graph file was also introduced. Again, the same series of actions is followed, with additional correctness checks and the dynamic re-initialisation of the viewer's GUI, in the case that the opened data corresponds to a simulation that used a different output analysis method.

Before compression had been considered for storing the graph data, an additional option was also being considered. This option would enable the modeller to "finalise" a graph, thus discarding its observations and storing it as a static (Java) image. This would minimise the storage requirements of the graph, but also minimise at the same time its available flexibility and functionality. Since the introduction of compression practically solved the storage problems, no such option was required. However, in the place of this option alternative functionality was introduced. This functionality provides the modeller with the option to save the graph currently under display as a GIF image. As an image, a graph can be used in documentation and web publications of simulation results. The graphs presented in figure 6.2 were saved as images using this functionality.

The final point of functionality provided was an HTML help file that may be accessed by the viewer. This help facility contains basic instructions and information about the layout of the viewer and the list of possible options.

7 Project evaluation

7.1 Introduction

In the previous chapters the project's goals were presented and discussed. Chapters 3 through 6 focused on the each of the main goals presenting issues of concern, design alternatives and chosen implementation approaches. Having concluded the discussion for the project's design and implementation, this chapter continues by presenting an evaluation study. This study will not repeat the proven benefits of SimJava as a simulation tool but will address aspects introduced by this project.

The evaluation study will begin by establishing the new PRNG's suitability for SimJava through a series of PRNG tests. Following this, a comparison study between the new and the original SimJava will be presented. A series of tests are first carried out to measure and compare both versions' efficiency. Following this, additional differences of extensibility and modelling flexibility will be discussed. Finally, this chapter will conclude with a discussion of steps that were taken to ensure the new version's correctness.

7.2 Random number generator tests

7.2.1 Introduction

Random number generation could be considered as the driving force of most discrete event simulation tools since their samples define the path an experiment follows over the state space. The goal of improving SimJava's sampling methods was built upon providing a new PRNG suitable for simulation studies. It is therefore essential that the selected PRNG be put to the test to prove its worth.

Passing a single PRNG test can't be considered as a guarantee of randomness. The reason for this is that a PRNG that passes one test may fail to pass another, or may even fail the same test if a different set of samples is used. As a result of this, several tests need to be performed on a PRNG in order to establish its suitability for simulation [1][8]. Sections §7.2.2 through §7.2.5 discuss how several general tests were performed on the new PRNG. These tests, although being simple, represent a minimum quality standard that any generator must meet. Section §7.2.6 discusses the Spectral test, a test that is specifically suited for the type of PRNG used for SimJava [3][4][5].

7.2.2 Chi-square test

This is the most commonly used test to determine if an observed data set satisfies a specified distribution. When used to test the samples of a PRNG, the (0,1) interval is segmented into a number of equally sized cells and a set of samples is generated. Following this, the observed frequencies of samples for each cell are compared to the expected frequencies, and the following quantity is calculated:

$$D = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i}$$

where k is the number of cells, o_i the observed frequency for cell i, and e_i the expected frequency for cell i. For an exact fit, D should be zero but due to randomness it is non-zero. It can be shown that D has a chi-square distribution with k-1 degrees of freedom. The null hypothesis, that the observations come from the specified distribution, can't be rejected at a level of significance a if the computed D is less than the $X^2_{[1-a;\ k-1]}$ i.e. the Chi-square quantile for a significance level a and k-1 degrees of freedom [8].

To test the new PRNG, 1000 samples are extracted and the test is performed for 10 cells. The seed used to obtain the samples for this test and all subsequent tests is 4851: the default seed used by `Sim_system` to automatically initialise the simulation's generators. The results are as follows:

Cell	**Observed**	**Expected**	$\frac{(\textbf{Observed} - \textbf{Expected})^2}{\textbf{Expected}}$
1	100	100	0.0
2	103	100	0.09
3	123	100	5.29
4	81	100	3.61
5	95	100	0.25
6	94	100	0.36
7	107	100	0.49
8	95	100	0.25
9	112	100	1.44
10	90	100	1.0
Total	1000	1000	**12.78**

Table 7.1: Results of the Chi-square test

For a significance level of $a = 0.1$, $X^2_{[0.9;\ 9]}$ is 14.68, and the observed difference 12.78 is less than theoretically allowed. Therefore, at the 0.10 significance level, the PRNG is accepted as a good source of uniformly distributed random numbers over (0,1).

7.2.3 Kolmogorov-Smirnov test

This test is similar to the Chi-square test with respect to the fact that it allows one to test if a given set of samples is from a specified continuous distribution. It is based on the observation that the difference between the observed CDF (Cumulative Distribution Function) $F_o(x)$ and the expected CDF $F_e(x)$ should be small. The symbols K^+ and K^- are used to denote the maximum observed deviations above and below the expected CDF in a sample of size n:

$$K^+ = \sqrt{n} \cdot \max_{x} \left[F_o(x) - F_e(x) \right]$$

$$K^- = \sqrt{n} \cdot \max_{x} \left[F_e(x) - F_o(x) \right]$$

K^+ measures the maximum deviation when the observed CDF is above the expected CDF, and K^- measures the maximum deviation when the observed CDF is below the expected CDF. If these values are smaller than $K_{[1-a;\ n]}$ i.e. the K-S distribution quantile for significance level a and n degrees of freedom, the samples are said to come from the specified distribution at the a level of significance.

For random numbers distributed uniformly between 0 and 1, the expected CDF is $F_e(x) = x$, and if x is greater than j-1 other observations in a set of n observations, then the observed CDF if $F_o(x) = j/n$. Therefore, to test whether a set of n random samples is from a uniform distribution, the first step is to sort them in ascending order $\{x_1, x_2, \dots, x_n\}$ such that $x_{n-1} \le x_n$. Then K^+ and K^- are computed as follows [8]:

$$K^+ = \sqrt{n} \cdot \max_j \left(\frac{j}{n} - x_j \right)$$

$$K^- = \sqrt{n} \cdot \max_j \left(x_j - \frac{j-1}{n} \right)$$

Performing this test for the new PRNG, 30 samples were generated and the K quantities were calculated as follows:

$$K^+ = 0.9963103416718281$$
$$K^- = 0.5583253108940137$$

For n = 30 and a = 0.1, $K_{[0.9;\ 30]}$ is 1.0424. Since both K^+ and K^- are less than 1.0424 the sample sequence passes the test at the 0.1 level of significance.

7.2.4 Serial-correlation test

Another method to test the dependence of two random variables is to see if their covariance is non-zero. If it is non-zero, the variables are dependent. The inverse however is not true since if the covariance is zero the variables may still be dependent.

Given a sequence of random numbers $\{U_1, U_2, \dots, U_n\}$, one can compute the covariance between numbers that are k values apart. This is called *autocovariance at lag k*. Denoting this by R_k, it may be computed as follows:

$$R_k = \frac{1}{n-k} \cdot \sum_{i=1}^{n-k} (U_i - 0.5)(U_{i+k} - 0.5)$$

For large n, R_k is normally distributed with a mean of zero and a variance of $1/[144(n-k)]$. The 100(1-a)% confidence interval for the autocovariance is:

$$R_k \pm \frac{z_{1-a/2}}{12 \cdot \sqrt{n-k}}$$

where $z_{1-a/2}$ is the Normal distribution quantile for a significance level of a. If this interval does not include zero, it can be said that the sample sequence has a significant correlation [8]. Performing this test for the new PRNG with a significance level of a = 0.1 and for lag 1 to 10 we obtain the following intervals:

Lag	Lower bound	Upper bound
1	-0.0004214568	0.0023203469
2	-0.0001665742	0.0025753666
3	-0.0011628685	0.0015792094
4	-0.0026571635	0.0000850516
5	-0.0020544608	0.0006878915
6	-0.0022860358	0.0004564536
7	-0.0019661775	0.0007764492
8	-0.0025059013	0.0002368626
9	-0.0011328845	0.0016100167
10	-0.0009133553	0.0018296831

Table 7.2: Results of the Serial-correlation test

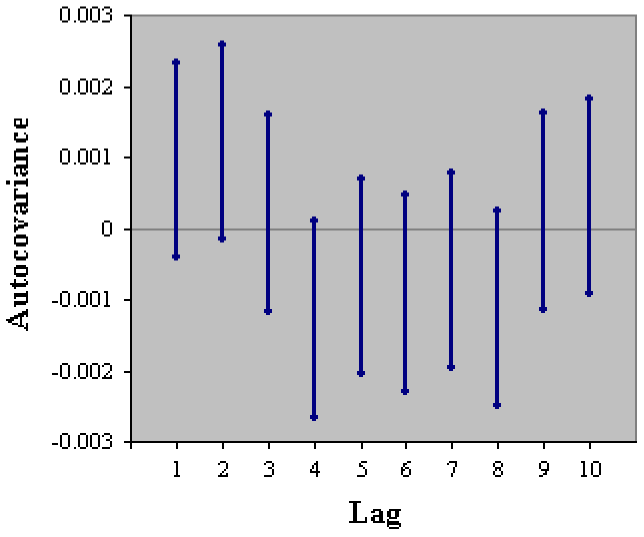


Figure 7.1: Confidence intervals of the autocovariances

Since all confidence intervals contain zero, it can be assumed that all autocovariances are statistically insignificant at a confidence level of 0.1.

7.2.5 Serial test

The serial test is similar in nature to the Chi-square test. It is used to test for uniformity in two dimensions or higher. Each dimension is segmented into a number of cells and a set of sample points is obtained. The observed point frequencies of the cells are compared to their expected frequencies and a Chi-square test is applied to measure the deviation.

For two dimensions, the space between (0,0) and (1,1) is divided into K^2 cells of equal area. A set of n samples $\{x_1, x_2, \dots , x_n\}$ is obtained and used to form *n/2 non-overlapping* pairs (x_1, x_2), (x_3, x_4), … , (x_{n-1}, x_n). These two-dimensional points are then counted to measure how many fall in each of the K^2 cells. Ideally, one would expect $n/(2K^2)$ points in each cell. The degrees of freedom for the Chi-square test in this case are K^2-1. This test can easily be extended to k dimensions by forming non-overlapping k-tuples, counting the observed frequencies of the K^k cells and comparing them to the expected frequency of $n/(kK^k)$, and finally performing the Chi-square test with K^k-1 degrees of freedom [8].

For SimJava's new PRNG, the serial test was applied in two, three, four and five dimensions using a significance level of a = 0.1. In order to have large enough observed and expected frequencies, additional sample points were required as the test was applied to higher dimensions. This fact makes the serial test quite time consuming for high dimensions.

For two and three dimensions, the sample points can be graphically presented in two and three-dimensional space. These graphical presentations can themselves serve as an initial simple test of uniformity. Note that in figure 7.3, the sample points were reduced in order to increase clarity. The test results for all dimensions can be found in table 7.3.

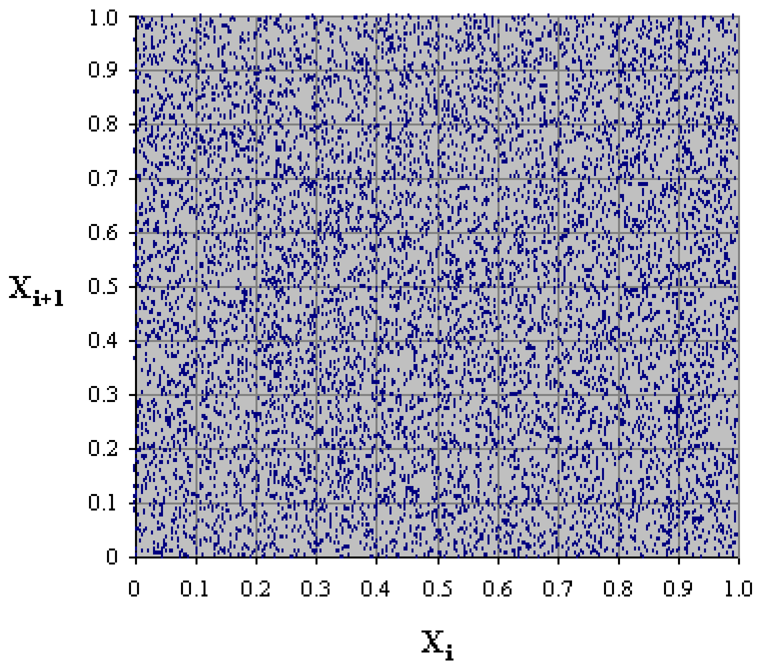


Figure 7.2: 10,000 sample points in two-dimensional space

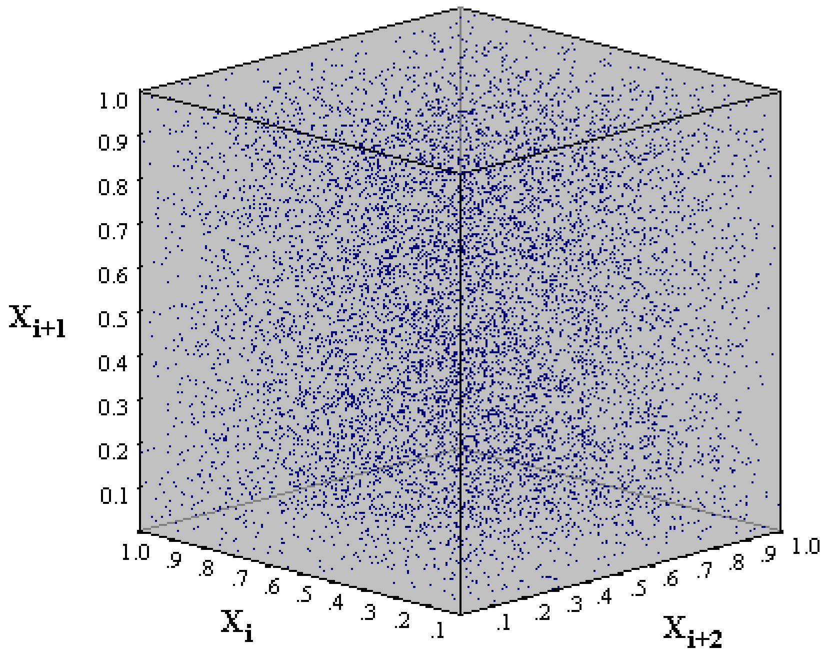


Figure 7.3: 10,000 sample points in three-dimensional space

Dimensions	Sample points	Cells	Test result
2	10,000	100	83.14
3	100,000	1,000	999.14
4	1,000,000	10,000	10,069.88
5	10,000,000	100,000	99,799.32

Table 7.3: Serial test results

For each dimension the result obtained from the Serial test was below the appropriate Chi-square value. Therefore, it can be said that the new PRNG successfully passes the Serial test for up to 5 dimensions, at a significance level of 0.1.

7.2.6 Spectral test

We will conclude the tests on the new PRNG with the Spectral test. This test is used to determine how densely the *k*-tuples $\{x_1, x_2, \dots, x_k\}$ can fill up the *k*-dimensional hyperspace. The *k*-tuples from a linear congruential generator (LCG) fall on a finite number of parallel hyperplanes. The Spectral test determines the maximum distance between adjacent hyperplanes. The larger this distance, the worse the generator. For generators with a small cycle this distance can be identified with a complete enumeration of the seed values it produces. In other cases however this test can be quite resource demanding [8].

The fact that distinguishes this test from the previous ones is that the parameters used in the generator are put to the test, rather than a set of its samples. This can be considered as the strength of this test since sets of samples depend on the selection of the PRNG's seed. For LCGs, and therefore MLCGs such as the PRNG selected for SimJava, this test can prove quite useful for selecting appropriate PRNG parameters, and has been used for this purpose in several relevant studies [3][4][5].

Performing the Spectral test for a single generator is not useful since there is no basis for comparison. This test is therefore used to compare different generators of the same type in order to select the one that gives the best results. Recall that MLCGs are generators of the form:

$$Y_i = A \cdot Y_{i-1} \bmod M$$

where Y_i is the next seed, Y_{i-1} the previous one, A the multiplier and M the modulus. Fishman and Moore in [5] performed an exhaustive study of such generators with modulus $M = 2^{31}-1$, testing 534,600,000 primitive roots of M in search of the best multiplier. The best multipliers this study produced are presented in table 7.4.

The basis upon which the generators are compared is the quantity $S_k(A,M)$. This is the ratio of the minimal achievable distance between successive hyperplanes, to the worst-case distance for each multiplier. The ratio depends on the number of dimensions k, the generator's multiplier A, and its modulus M. The higher the ratio for each dimension, the better the multiplier [4]:

Multiplier	Dimension				
	2	3	4	5	6
742,938,285[a]	0.8673	0.8607	0.8627	0.8320	0.8342
950,706,376[a]	0.8574	0.8985	0.8692	0.8337	0.8274
1,226,874,159[a]	0.8411	0.8787	0.8255	0.8378	0.8441
62,089,911[a]	0.8930	0.8903	0.8575	0.8630	0.8249
1,343,714,438[a]	0.8237	0.8324	0.8245	0.8262	0.8255
630,360,016[b]	0.8212	0.4317	0.7832	0.8021	0.5700
16,807[c]	0.3375	0.4412	0.5752	0.7361	0.6454

a. *The top five multipliers*
b. *The multiplier used in SIMSCRIPT II.5*
c. *The multiplier used in SIMAN and SLAM (entries are $S_k(A,M) \times 1000$)*

Table 7.4: Distance between parallel hyperplanes for $Y_i = A \cdot Y_{i-1} \bmod (2^{31}-1)$

Note that A = 742,938,285 is the multiplier selected for the new PRNG. This is the multiplier proposed by Fishman in [4] for use in simulation studies. The two-dimensional parallel hyperplanes for this multiplier and subsequently, for the new PRNG are as follows:

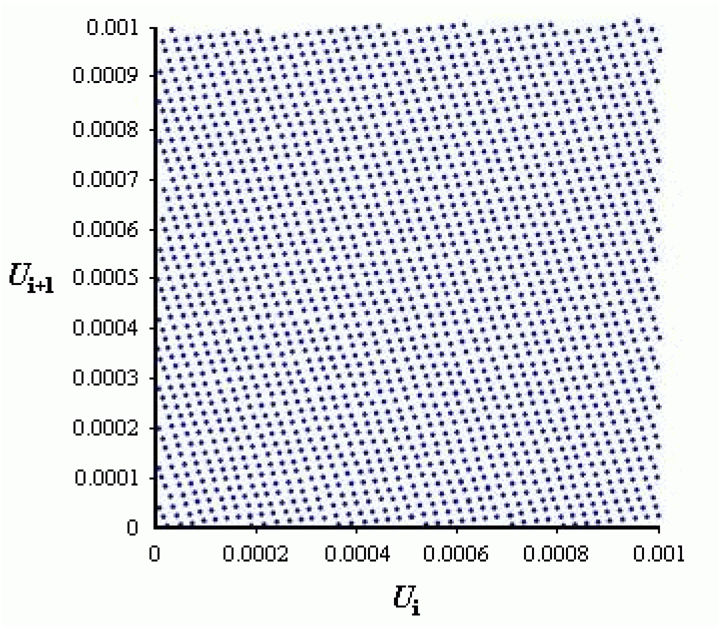


Figure 7.4: Parallel hyperplanes for the multiplier A = 742,938,285

7.3 Comparison with the original SimJava

7.3.1 Introduction

Having established the suitability of the new PRNG for simulation, we will proceed with a comparison study between the new version of SimJava and the original. This study will focus mainly on testing the new version's efficiency, in terms of speed and memory usage, against the performance of the original version. The performance characteristics of both versions can be easily quantified and compared through experimentation. Finally, further aspects and differences of the two versions that can't be precisely measured will be discussed to highlight their strengths and weaknesses.

In order to perform the performance tests presented in Sections §7.3.3 and §7.3.4, a simulation was built for both versions of SimJava. This simulation was kept as simple as possible in order to focus only on the kernels' performance rather than on introduced modelling complexity. The simulation consists of a Source entity that generates events for a randomly chosen Sink entity. The Source entity is set to measure the rate at which it generates events, and the Sink entities are set to measure their utilisation and event service time. An effort was made to build the simulation in the most similar manner for both versions. The resulting simulations are `SimpleTest.java` and `SimpleTestOld.java` for the new and original versions respectively. The source for these can be found in Appendix B.

Three sets of tests were performed for each performance characteristic: testing of the simulation for the original SimJava, testing for the new version with efficient measures, and testing for the new version with detailed measures. In the case of the new version's simulations, the set of tests using efficient measures is the most similar and directly comparable with the original version's results. The detailed measure tests are included mainly as a comparison between efficient and detailed measures.

7.3.2 Efficiency enhancements

7.3.2.1 Introduction

Before proceeding to the performance tests, a brief discussion must be made concerning the modifications made to SimJava's kernel. Obviously, much additional functionality and complexity was introduced in order to implement the sophisticated goals of this project. These modifications were expected to introduce a time and memory overhead to the performance of SimJava compared to the original version.

In order to minimise this overhead and improve efficiency where possible, several additional modifications were made to the original kernel. These modifications were optimisations of several aspects of SimJava and changes in the way internal actions are performed.

7.3.2.2 Efficient implementation of the event queues

Since the event queues in many simulations can reach very large sizes, it is imperative that they are implemented efficiently. The `Vector` and `Enumeration` data structures used for the event queues in the original SimJava are burdened with thread synchronisation overheads. However, these are unnecessary since the queue-manipulation methods of `Sim_system` are themselves thread-safe. Furthermore, the original implementation requires searching the event queue to find the appropriate position for each new event. Since most events are inserted at the end of each queue, this search process can often be avoided.

In the new version, a `LinkedList` was selected as the queues' data structure since it is free of synchronisation overheads, it is efficient for iteration (using a `ListIterator`), and is very efficient for appending elements at its end. Finally, the insertion algorithm was modified to avoid searching through the entire queue, by storing the last inserted event's time and comparing it to the new event's time. If the new event's time is greater it may immediately be appended to the queue's end.

7.3.2.3 Storing runtime predicates

Predicates are used by SimJava to selectively wait for certain event types. In the original SimJava, the entity itself has to check each incoming event to see if it matches the desired predicate. In the case of non-matching events, the entity simply returns to a paused state, therefore resulting in an unnecessary thread context switch. In experiments with many entities and event types, this process could cause a significant overhead.

In order to avoid this overhead in the new version, predicates used in runtime methods are stored by `Sim_system`. Since entities may only call one runtime method at a time, storing and retrieving an entity's predicate is a simple process. When an event for a waiting entity is due, a check is made to verify whether or not a predicate has been used. If a predicate has been used, the entity is activated only if the event matches the predicate, thus avoiding unnecessary context switches.

7.3.2.4 Efficient implementation of search methods

SimJava's kernel contains several search methods. These methods search internal data structures to find an entity, port, or event of interest. In the original version, all these methods are inefficiently implemented since the entire relevant data structure has to be searched even if the desired e.g. entity has already been found. This problem was corrected in all the new version's search methods.

7.3.2.5 Use of efficient data structures

The original SimJava makes use of the `Vector` and `Enumeration` data structures for all its internal requirements. As mentioned in Section §7.3.2.2, these data structures have a synchronisation overhead in order to be thread-safe. However, the kernel methods that use these data structures are all themselves thread-safe, removing the need for this overhead. For the new version of SimJava, all the `Vectors` and `Enumerations` were substituted with the more efficient and up-to-date `ArrayLists` and `LinkedLists`.

7.3.3 Speed tests

As mentioned in Section §7.3.1, the simulations used for the tests were kept as simple as possible in order to focus on the performance of both versions' kernels. To test their speed, two tests were performed for each simulation: the first one experimenting with a varying number of entities, and the second one with a varying number of run lengths.

The first test focuses on the problem that burdens most process-based simulation tools. When using large numbers of threads, speed is greatly compromised by the overhead of numerous context switches. Since in SimJava each entity is a thread, increasing the number of entities for each simulation will test how well both versions scale in extreme circumstances.

The run length for the first test was set to 25,000,000 time units. The time results presented are averaged over three runs. The results are as follows:

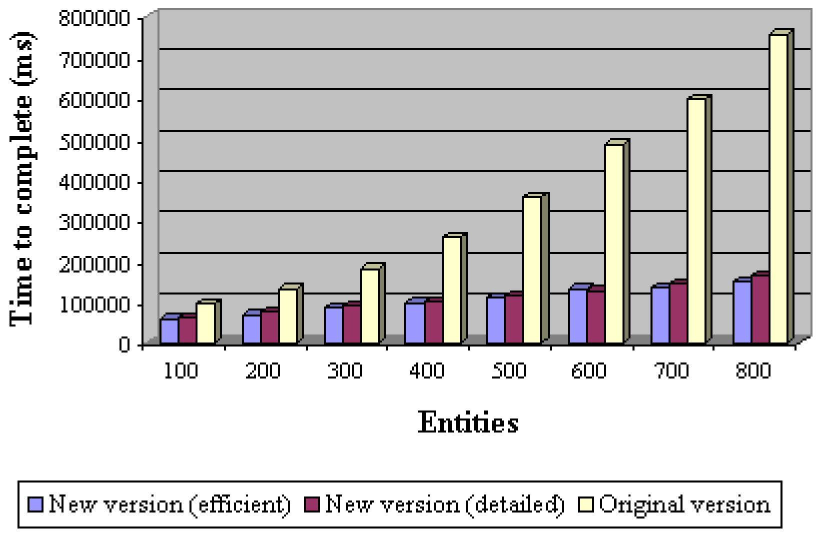


Figure 7.5: Time to complete for increasing entity counts

The second test focuses on the time required to complete for varying run lengths. For this test the number of entities was kept at 500 as the run length was increased. As previously, the results presented are averaged over three runs. The results are as follows:

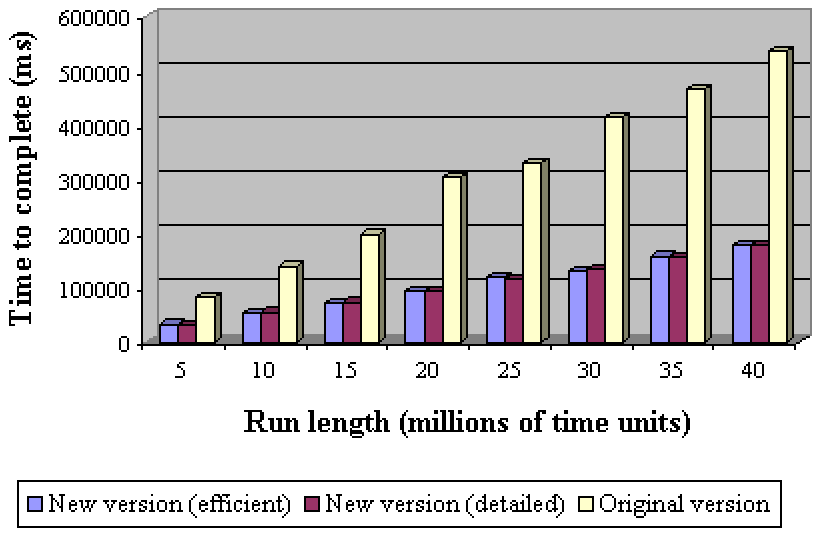


Figure 7.6: Time to complete for increasing run lengths

As the results of these two tests suggest, the speed increase in the new version is quite impressive. In the first test, for the most extreme case of 800 entities, the new version completed 5.046 times faster, while in the second test, for the run length of 40,000,000 time units, it completed 2.982 times faster. The results indicate that as the kernel is put to the test with long running simulations and large numbers of entities, the difference in performance between the new and the old versions grows exponentially.

The improvement in speed is expected to be at the cost of memory use. The results for memory consumption for these two tests follow in Section §7.3.4.

These tests also suggest that no significant difference in speed exists between the efficient and detailed measures. The overhead of recalculating measurements for efficient measures is matched by the overhead of storing each observation for detailed ones. In most cases, the efficient measure runs completed slightly faster than the detailed ones. This was mostly observed in the increasing entities tests, since the time required to initialise an entity using detailed measures is slightly more because of the need to instantiate and allocate memory for the observation collection data structures. Very small differences such as the ones observed in the increasing run length tests, including the runs where the detailed measure runs were shorter, can be attributed to operating system induced variations in time.

7.3.4 Memory usage tests

We will now proceed to test the performance of the new and the original SimJava on the basis of memory use. For each of the tests performed to measure the kernels' speed, memory consumption was also recorded. Note however that in this case the memory measurements are obtained through the Java virtual machine and may not be completely accurate. They do, however, provide a crude estimate of the memory used by the JVM in each case and therefore, are sufficient for this study.

Recall that the measurements of the original SimJava should be compared with the new version's measurements for efficient measures. The results for detailed measures are excluded form the graphs in order to increase clarity. The detailed measures' results are presented separately in tables 7.5 and 7.6.

For the first test with run length at 25,000,000 time units and varying numbers of entities the results are as follows:

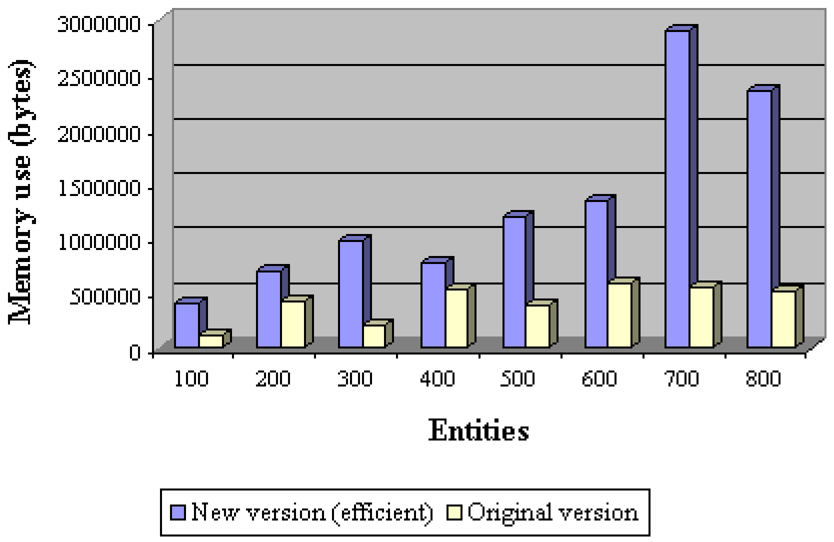


Figure 7.7: Memory use for increasing entity counts

Entities							
100	**200**	**300**	**400**	**500**	**600**	**700**	**800**
27830176	30199000	28974296	28370664	29485776	27046104	28538416	28684688

Table 7.5: Memory use results for detailed measures (increasing entities test)

For the second test with 500 entities and varying run lengths the results are as follows:

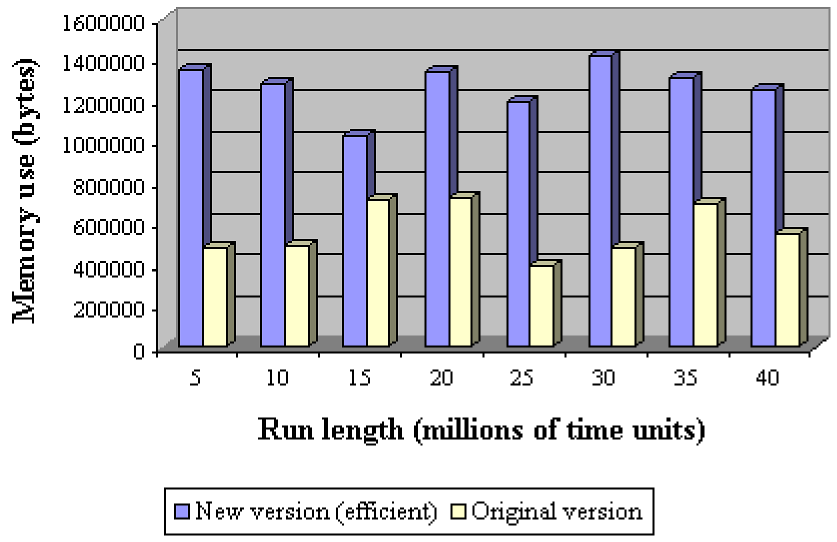


Figure 7.8: Memory use for increasing run lengths

Run length							
5	**10**	**15**	**20**	**25**	**30**	**35**	**40**
6673848	12766008	16548480	24718320	29485664	36039992	41728752	45652320

Table 7.6: Memory use results for detailed measures (increasing run length test)

The results obtained from the tests indicate that memory consumption is increased in the new version of SimJava. Comparing the two sets of results, it is apparent that memory use increases as more entities are introduced into the simulation, but remains roughly the same for increasing run lengths. For the case of 25,000,000 time units as the run length and 800 entities (last test in figure 7.7), the new version uses 4.565 more memory compared to the original, while for 40,000,000 time units and 500 entities (last test in figure 7.8), it uses 2.274 times more.

Although the new version of SimJava does exhibit much higher memory requirements than the original, it can be said that they remain within reasonable bounds. It must be considered that in the original version of SimJava, memory usage was minimal due to its simplicity. The only major data structures present within the kernel itself were the event queues and the entity list. The added complexity of the new version, necessary due to the increased automation and functionality, was bound to increase the memory requirements. However these added memory requirements can be considered acceptable since they are not extremely demanding.

The results obtained from testing the new version using detailed measures exhibit much larger memory requirements. These can prove to be quite problematic and limiting if more complexity is introduced into the simulation. In the increasing run length tests, the memory use for detailed measures grows as the run length increases since more observations are collected by each entity's `Sim_stat` instance. Note that this is the exact reason why efficient measures were introduced to the new version of SimJava. In situations where a large number of entities are required as well as an extremely large run length, as in the case of the experiments performed, efficient measures should be preferred at the cost of the detailed measures' information gain.

Finally, a comment should be made concerning the fluctuation of the memory use between experiments, observable in both graphs. As mentioned in the beginning of this section, using the JVM to produce memory usage figures is quite inaccurate. The results obtained can be used only to examine very large differences such as those between the new and original versions, or between the efficient and detailed measures. The fluctuations in memory use between tests for both the new and original versions should be attributed to this inaccuracy.

The results from all the experiments are also available in table form in Appendix C.

7.3.5 Modelling power, ease of use and clarity

The tests performed in Sections §7.3.3 and §7.3.4 produced interesting results. It is shown that the new version of SimJava is much faster than the original especially when simulations reach extremely large sizes, the price for this speed-up being the increased memory requirements. However, when efficient measures are used, the memory use remains within acceptable bounds even in highly demanding experiments. As a result, it can be safely assumed that the performance tests of the new version were successful. We will now proceed to discuss other, non-quantifiable issues concerning the two versions.

The main weakness of the new SimJava is the increased kernel complexity. This complexity consists partly of optimisations that served to provide SimJava with the speed-up presented in Section §7.3.3. Even with respect to memory consumption, the introduced complexity does not provide a great overhead. The problem, however, is the fact that the simplicity and extensibility of the kernel had to be sacrificed in order to introduce all the required functionality. One of the appeals of SimJava was the fact that its kernel was so simple that it could be easily modified and used as the basis for other simulation tools, such as those presented in Section §2.5. Even though SimJava itself was not powerful, it provided a proven, easy to use, and extensible simulation backbone.

Although this extensibility was sacrificed, SimJava gained much strength as a simulation tool. A first point to observe is the fact that writing simulations is now much simpler and cleaner. This can be easily seen by comparing the two simulations used for the performance tests. The problems of the simulation written for the original SimJava include:

- An additional entity needs to be defined in order to collect observations and produce the experiment's measurements and report.
- Observations must be manually obtained and measurements manually calculated.
- The simulation's termination is not clean since the Source entity needs to notify all other entities of the simulation's completion.

These issues are not a problem when using the new version of SimJava. Furthermore, the central definition of transient and termination conditions as well as other simulation settings and parameters serves to make simulations easier to define and understand.

The main benefit offered by the new version of SimJava is, of course, the added range of automated functionality. In order to reproduce this functionality in simulations written for the original version of SimJava, an extreme coding effort would have to be applied that would prove to be very time consuming and error-prone, limiting at the same time the clarity of the experiments' definition.

Examples of such cases can be located in each of the major goals of this project:

- Defining well-spaced seeds would require separate experimentation with the random number generator in order to produce a list of suitable seeds. Furthermore, since the parameters of SimJava's original PRNG are not publicly available, identifying the exact cycle length and therefore, the maximum effective run length for experiments would be difficult.
- Applying sophisticated output analysis methods would be an extremely demanding task. To begin with, new observation storage classes would need to be written in order to make the simulation's observations individually accessible. Furthermore, in the case of independent replications, the simulation's kernel would have to be modified in order to reset simulation parameters, store collected data, and perform entity backups.
- In order to provide more sophisticated termination conditions such as the ones based on a confidence interval's accuracy, variance reduction techniques would be required. Performing variance reduction would require implementing output analysis methods, in which case the modeller would be faced with the above problems.
- Graph support would require the complete implementation of a new set of classes. Such a coding effort would go completely beyond the modelling aspects of the simulation.

It should be apparent that manually attempting to introduce much of the functionality added by this project would be an extremely demanding task. This should also be considered along with the fact that modellers, although interested in extending their experiments in such ways, wish to focus only on the modelling aspects of their simulations.

7.4 Establishing correctness

In Section §7.3 several benefits of the new version over the original SimJava were presented concerning speed increases, as well as increased modelling power, ease of use and improved clarity. These benefits, however, are important only if the new version carries out experiments correctly. This section briefly discusses steps that were taken to ensure that the new version produces correct results.

The most difficult task was to ensure that the enhanced statistical support functioned correctly. The primary source of problems proved to be the automatically calculated measures that were made available to entities through `Sim_stat`. Once detailed measures had been implemented, the `Sim_stat` class was used in test programs in order to establish that the collection of observations was carried out correctly. These were not simulations since the new version's kernel was not yet complete. These test programs produced observations that were also used to manually calculate all possible measurements and compare them to the ones produced by `Sim_stat`. Once all the calculation methods were manually checked for all measure types, and the new basic structure of the kernel was ready, the `Sim_stat` class was tested within simulations. By manually checking all observations and calculating measurements, the process of automatic observation collection and measurement calculation was modified until correct results were produced.

Once detailed measures were in place, testing the newly added efficient measures was more straightforward. Efficient measure results could be verified against their equivalent detailed measures' results. In complicated situations, results were again manually checked by examining all the measures' observations. At this time output analysis methods were completed and as such, independent replications could be used to test results for a great number of simulation runs. The presentation of the replications' detailed results in the report file greatly helped to identify discrepancies between efficient and default measures. Once a problem had been identified for a specific replication, it was reproduced in a single run, the observations of which were then manually checked in order to pinpoint and correct the problem.

The implementation of output analysis methods required further strenuous testing. For independent replications, several runs were independently carried out to recreate exactly each replication. The results obtained from each replication were then compared to the results of the individual runs. For the case of batch means, the automatically produced batch results were verified again through manual examination of the observations. In both cases, the correctness of the obtained total measurements and confidence intervals were established by manually carrying out the relevant calculations. Furthermore, in order to check whether both output analysis methods were implemented correctly, experiments were run in which the output analysis method was switched. The fact that the obtained results displayed insignificant differences provided a further assurance of the methods' correctness.

Having a tested reporting facility in place also assisted in testing the correctness of the new transient and termination conditions. In the case of event completion or elapsed time conditions, results could be manually cross-checked with the obtained observations and the results presented in the report file. The minimum-maximum method for transient period definition was tested by manually checking its results with the collected observations. The main problems, however, occurred with the termination conditions that were based on variance reduction techniques. Apart from complicating the kernel's implementation they presented additional situations that needed to be considered for observation collection and measurement calculation. Several of these problems were identified while performing the project's evaluation when irrational results were being produced. All these problems were ultimately solved by manually checking all the observations and verifying that the obtained results were correct.

Several problem cases were also identified by running simulations of the original SimJava with the new version. Most problems found were related to the absence of an explicit termination condition. These problems were identified and the kernel was appropriately modified to handle such situations. Apart from these uncovered problems, testing the new version of SimJava with existing simulations served to increase the confidence in the new version's ability to handle existing SimJava simulations.

In addition, several existing simulations were modified to make use of the new version's functionality, such as the new statistical support, and the new transient and termination conditions. The results obtained from the modified simulations were compared to the original simulations' results and the difference was found to be insignificant. This was also the case for the results obtained from the test simulations built for the performance evaluation tests of

Sections §7.3.3 and §7.3.4. The resulting differences in all cases, which are mostly attributed to the different random samples produced, served as an additional assurance of correctness.

Concerning the graphical output analysis, correctness was established by comparing the graphs' displayed measurements with those contained in the report file. Although the existing sample average calculation method of `Sim_stat` proved to be too inefficient for plot generation, it was used to establish the correctness of the efficient calculation method that was ultimately used. Both methods were used to draw the sample average's plot and the efficient method was modified until the two plots were identical. Concerning the layout and the information displayed by the SimJava Graph Viewer, graphs from several simulations were presented to Dr. Jane Hillston as well as several fellow students involved in simulation studies. Their comments were used to finalise the viewer's layout and options.

Finally, it must be noted that the new version was tested in depth by Kannan Ratnasingham, a fellow MSc student, who used it for his simulation study of Freenet. Although much of the new statistical support was not used in his study, most other aspects of the new SimJava were put to the test. Through his work, SimJava's new kernel was tested to otherwise impractical extents, uncovering in the process several errors, both in this project's work, as well as in the original SimJava.

8 Conclusion

8.1 Introduction

Chapters 3 through 6 focused on each of the project's major goals, as presented in Section §1.2. Chapter 7 followed in order to present an evaluation study concerning SimJava's new PRNG and its performance characteristics. This chapter serves as the project's conclusion, presenting some last thoughts on the completed work.

Section §8.2 completes the list of work made by this project by briefly discussing additional minor goals and improvements to SimJava. Section §8.3 highlights the project's achievements and presents an evaluation of the project's overall progress and design approach. Finally, Section §8.4 discusses issues that could further enhance SimJava and recommended future work.

8.2 Additional improvements to SimJava

8.2.1 Improved trace output

As discussed in Section §2.3.6, tracing is the process of recording the simulation's internal actions as a time-ordered list of events. This information can be used to examine in detail the behaviour of experiments and serve as a model verification tool. In the original SimJava, trace was produced by the kernel by default to record all internal actions, and by the entities using the `sim_trace` method to include user-defined trace. The problem with tracing was that the only available options were to switch off tracing altogether, or produce the full amount of trace.

In the new version of SimJava, the modeller is provided with more control over which trace is produced. The previous options of switching off tracing or generating all possible trace are still maintained. However, the modeller now has the ability to specify which trace messages are of interest. Entity trace may now be produced without the overwhelming default trace, or events of interest may be tracked. In the latter case, trace is produced only when processing events whose tag matches a specified tag of interest.

8.2.2 Extended animation functionality

Several enhancements were also made to the functionality available to animated simulations. Although very few modifications were made in the way animation is actually produced, enhancements were made concerning the output of animated simulations. Additional methods were provided to enable the modeller to specify the amount of output desired; the options being to include the simulation's progress messages and the simulation's report. In any case, the animation's applet is appropriately extended to display this additional information.

8.2.3 Additional runtime methods

The runtime methods provided to entities were also modified. The methods that required modification were the delay-related methods for holding and waiting. The holding methods were deprecated and substituted with two new method families, one for pausing and one for processing. This was made necessary by the introduction of the new statistical support that required the knowledge of whether a holding entity was spending time processing or being inactive. Finally, all method families were provided with additional methods that offer greater control over an entity's behaviour, such as the ability to process until interrupted by a specific event.

8.2.4 Other minor improvements

Apart from the improvements mentioned in the above sections, several minor improvements were also completed:

- Additional predicate classes were implemented to extend the range of predicates that can be used by entities for event selection.
- An exception hierarchy was provided for SimJava to provide a cleaner and more informative error-reporting mechanism.
- The few bugs that were found in the original SimJava, mainly concerning animation, were fixed through the course of the project.

Finally, all the functionality of the new SimJava is documented and discussed in a detailed user manual, which can be located at `http://www.dcs.ed.ac.uk/home/s0129537`. The new Javadoc API specification can also be found here. These resources are based at the above link temporarily and will later be moved to the SimJava homepage.

8.3 Project achievements

Section §1.2 presented and discussed this project's major goals. In order to review the achievements made, each goal will now be briefly highlighted to examine the degree to which it was achieved.

The first goal of this project, covered in Chapter 3, was the improvement of SimJava's sampling methods. A new random number generator was provided for SimJava that has been extensively tested and proven to be suitable for simulation studies. Section §7.2 presented a successful evaluation of the new generator's suitability by presenting test results that testify to its statistical acceptability. After providing a new PRNG, the subject of distribution sampling was addressed and a new set of distribution classes was implemented. These distribution classes, including discrete and continuous distributions, serve to provide the modeller with more modelling flexibility, compared to SimJava's original limiting range of distribution classes. Finally, generator seeding was implemented as an automated procedure by allowing the kernel to generate seeds for the PRNGs used in experiments. This automation eliminates the error-prone process of manual seeding and ensures well-spaced sample sequences for the simulation's generators.

Chapter 4 focused on the goal of providing sophisticated statistical support to SimJava experiments. The `Sim_stat` class was implemented as an easily accessible statistics-managing class for each entity, providing a range of default and custom measures. The process of observation collection and measurement calculation was implemented to be fully automated or at least require minimal effort. The benefits of this automation can be seen in the simulations used in Section §7.3 to perform the comparative performance study, where the ease of use and modelling clarity offered by `Sim_stat` are made apparent. These examples also illustrate the benefits of the automated report generation that was provided for SimJava, removing the need for manual effort that was a necessity in the original SimJava. Finally, sophisticated output analysis methods were introduced to provide quality guarantees for the experiments' obtained results, again in an easy to use and fully automated manner.

Improving the range of transient and termination conditions for SimJava simulations was the focus of Chapter 5. SimJava was augmented with functionality for centrally defining such conditions and automatically checking if they are satisfied. The definition and checking of conditions based on elapsed time or event completions is made simple, reducing modeller effort and enhancing the clarity of a simulation's settings. Furthermore, more sophisticated conditions were made possible that employ output analysis methods for variance reduction. This new type of condition permits the kernel to automatically drive the simulation run based on the accuracy obtained for a certain measure of interest.

Chapter 6 covered the final major goal of this project, the provision of graphical output analysis. SimJava's kernel was modified to collect all the relevant experiment data and output it as graph data, if so requested, again through a fully automated process. A companion graph viewing utility was built for SimJava, capable of loading the generated data and producing detailed graphs for each of the simulation's defined measures. A range of functionality was implemented for examining these graphs, ranging from zooming to annotating. These graphs and the graph viewing utility serve to greatly increase the information gain from SimJava simulations.

This project's evaluation study was presented in Chapter 7. This study apart from establishing the new PRNG's suitability for SimJava in Section §7.2, highlights another interesting achievement of this project. In Section §7.3.3, where the speed performance of SimJava's new version is compared to the performance of the original, the tests made reveal that the new kernel is faster than the original, and that it scales better in cases of large simulations and increased run lengths. This performance increase is due to the efficiency enhancements that were introduced in order to minimise the overhead of the new version's added complexity.

This evaluation study did however also reveal the cost at which the speed improvements were achieved. The tests of Section §7.3.4 focused on the memory consumption of both versions and showed that the new version is burdened with increased memory requirements. In the case of efficient measures these added requirements are considered as acceptable since the overall memory use remains within reasonable bounds. However, in the case of detailed measures the memory requirements are quite demanding and limiting. Although the tests were made on large and long simulations, it could be expected that complicated simulations could require an extremely large amount of memory.

Originally, the implementation of SimJava's statistical support was focused on providing only detailed measures in order to maximise the information obtained from experiments. However, once the first development stage of the project was completed, tests were performed that revealed the detailed measures' large memory requirements. Since time was available, it was decided to attempt to solve this problem by providing efficient measures to substitute the detailed ones. Finally, having implemented the efficient measures, the decision was made to offer both alternatives to the modeller, allowing him to decide whether efficiency was to be preferred over detail.

One final issue of interest is the use of SimJava as the basis of this project. It was mentioned in Section §1.3 that the simplicity of SimJava's kernel required a large amount of re-implementation in order to support this project's functionality. A good question in this case would be why the decision was not made to build a new simulation package from scratch instead of using SimJava. SimJava's simplicity, although requiring significant redesign effort, did provide an easily extendible simulation backbone. The kernel, the simulation building blocks, and the event handling methods have been extensively used and provided a solid framework upon which additions could be made and from which design guidelines could be extracted. Testaments to this fact are the numerous extensions that have been made to SimJava, presented in Section §2.5. If SimJava was not used and a new simulation API had to be built, much of the functionality provided through this project would not have been achieved since much time and effort would be spend on building the new API's internal structure.

In addition, SimJava has an established community of users that would welcome a new and enhanced version. Since this project's ultimate aspiration is to be widely used and provide simulation practitioners with a useful simulation tool, SimJava's existing user community was quite an attractive benefit. If a new simulation API were to be built, the result would probably be another tool with average functionality that would go largely unnoticed.

8.4 Future work

The goals set out for this project were to enhance SimJava with statistical support, improve its modelling strength and clarity, and maximise the information gain from its experiments. The completion of these goals also provided ideas for further enhancements. Such ideas could be adopted as a motivation for future work on SimJava:

- One aspect that provides ground for improvement is SimJava's use of threads. Being a process-based simulation tool, threads are essential to SimJava in order to enable multiple processes or entities to execute concurrently. Although the process-based approach to simulation allows easier definition of experiments, it is comes at the price of the increased overhead by the constant context switches between threads. This is the major problem that burdens most process-based simulation tools and leads simulation practitioners to select event-based simulation as a faster alternative. An idea for future work would be to minimise the thread context switches of SimJava by combining several Java threads into a single operating system thread. This could be achieved with work at the bytecode level, where multiple threads could be restructured into method invocations within a single thread. This would effectively transform SimJava process-based simulations to event-based ones at the bytecode level, greatly improving SimJava's performance.
- Recall that in Section §2.5, Distributed SimJava was presented as an extension to SimJava. Using Java RMI, it offered the distribution of work by having entities run at different, remote hosts, thus sharing computation load for complex simulations. Such an extension could also be made for the new version of SimJava that would combine the enhancements of this project with the benefits of a distributed environment.

- The last major goal of this project, the provision of graphical output analysis, significantly increases the information gain from SimJava experiments. However, graph data is produced at the simulation's completion, often requiring significant time to be stored and loaded. A useful alternative for experiments would be to have graphs that were updated at runtime to display the current state of the simulation's measures. However, in order to minimise the graphs' introduced overhead, an efficient way of achieving this would have to be employed, possibly making use of the javabeans framework used in `simdiag`. Finally, apart from displaying the measures' sample mean, the option could be provided to build graphs of observations. These graphs could be used to graphically identify transient periods for subsequent simulation runs.
- A final idea for future work involves SimJava's animation facilities. Currently, animated simulations are built as applets to be run in web browsers. Although applets provide attractive aspects of accessibility and ease of distribution, they do impose certain restrictions. To begin with, the JRE (Java Runtime Environment) provided in web browsers is rather outdated, requiring in most cases installation of a new version's plug-in. More seriously, applet sandboxing restricts animated simulations especially with respect to their ability to produce an experiment's output. These problems could be easily overcome by providing the ability to animated simulations to be run as applications. The use of applets would be substituted with other windowing components that would free simulations from applet restrictions.

References

[1] Bratley P., Fox B. L., Schrage L. E., *A Guide to Simulation*, Springer-Verlag 1987, Chapters 1-6.

An excellent source for most aspects of this project. It provided good introductory material for simulation as a modelling technique as well as excellent background for output analysis and variance reduction methods. The algorithm for performing variance reduction using batch means was adapted from here as well as the technique of jack-knifing for calculating the batch means' serial correlation. Additionally it was a very good source for PRNG issues, especially random variate generation, providing algorithms for several implemented distributions.

[2] Buyya R., Murshed M., *GridSim: a Toolkit for the Modelling and Simulation of Distributed Resource Management and Scheduling for Grid Computing*, The Journal of Concurrency and Computation: Practise and Experience, pages 1-32.

Used to obtain general information on GridSim, one the extensions to SimJava that was presented in the related work section.

[3] Entacher K., *A collection of selected pseudorandom number generators with linear structures*, 1997, pages 5-13.

Used as a source for comparing different PRNGs. The Spectral test results presented here were compared to the results in [5], assisting in the selection of the multiplier and modulus for SimJava's new PRNG.

[4] Fishman G. S., *Discrete-event simulation: modelling, programming, and analysis*, notes 2000, Chapters 1, 3, 4, 6, 8, 9.

Although being quite complicated and detailed, this book was one of the primary sources for this project. The first chapters provided introductory material but its main use was for defining details of the new statistical support. It provided the basis for defining default measures, measure types, available measurements, observation collection methods, measurement calculation methods, and also influenced the form of the simulation report file. In addition, it provided an excellent source for selecting a suitable PRNG and useful algorithms for random variate generation.

[5] Fishman G. S., Moore L., *An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31}-1$*, SIAM Journal on Scientific and Statistical Computing 7 (1986), no. 1, pages 24-45.

Provided the basis for the selection of SimJava's PRNG. After the suggestions made in [4], the Spectral test results were obtained from this source and used to select an appropriate multiplier and modulus. The test results are those presented for the Spectral test in the project's evaluation.

[6] Howell F., MacNab R., *Using Java for Discrete Event Simulation*, (1998) Proceedings of the 12th UK Computer and Telecommunications Performance Engineering Workshop, University of Edinburgh, pages 219-228.

This source was primarily used for the background chapter. The information on the HASE project and SIM++ were obtained from here, as well as the motivations that led to the development of SimJava.

[7] Howell F., MacNab R., *SimJava: a discrete event simulation package for Java with applications in computer systems modelling*, Proceedings of the 1st International Conference on Web-based Modelling and Simulation, San Diego CA, Society for Computer Simulation, 1998.

Describes the details and simulation model adopted by SimJava. This source served as background for the understanding of SimJava's internal structure and to obtain information on the animation and graph drawing facilities already present.

[8] Jain R., *The art of computer systems performance analysis*, John Wiley & Sons Inc. 1991, Chapters 25-29.

One of the main sources used for this project. It provided excellent introductory material for the background chapter. Furthermore it was excellent regarding issues of steady state analysis and the identification of a transient period. The minimum-maximum method for transient period identification was adapted from here. In addition, it provided very good material on several output analysis methods and techniques for variance reduction. It was also the primary source for issues concerning the selected PRNG, providing information on generating well-spaced seeds, random variate generation, and PRNG tests. Most distribution classes were implemented based on algorithms provided here, and all the simple PRNG tests presented in the evaluation were adapted from here.

[9] Kapuno R. R. A. Jr., Nagarur N. N., *SimProd: A web-based flexible simulation package for production systems*, Electronic Journal of the School of Advanced Technologies, Asian Institute of Technology, Pathumthani, Thailand.

Provided general information on SimProd, one of the SimJava extensions presented in the related work section.

[10] Kleijnen J. P. C., *Statistical tools for simulation practitioners*, Marcel Dekker Inc. New York 1987, Part I.

Quite a detailed book. It was mainly used to obtain background knowledge on the statistics involved in output analysis methods.

[11] Kleijnen J. P. C., *Statistical techniques in simulation Parts I & II*, Marcel Dekker Inc. New York 1974, Chapters 1, 6.

Although not used for most of the project's aspects, its introductory chapters were used as background on general simulation issues.

[12] Kreutzer W., System simulation: *Programming styles and languages*, Addison Wesley 1986, Chapter 1.

This source was mainly used as background by studying the approach adopted by other simulation modelling languages and styles.

[13] MacDougall M. H., *Simulating Computer Systems: Techniques and Tools*, the MIT Press 1987, Chapters 1-4.

A very good source on output analysis methods and variance reduction techniques. It provided useful background on how and why output analysis is used, discussing the effects of correlation and why variance reduction should be applied. It was especially useful for independent replications and batch means. The algorithm for variance reduction using independent replications was adapted from here.

[14] Matsumoto M., Nishimura T., *Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Transactions on Modelling and Computer Simulation 1998 8(1), pages 3-30.

Provided the source for the Mersenne twister, one of the PRNGs that were considered as candidates for the new version of SimJava.

[15] Page E. H., Moore R. L. Jr., Griffin S. P., *Web-based simulation in SimJava using remote method invocation*, Proceedings of the 1997 Winter Simulation Conference.

Provided general information on Distributed SimJava, another extension to SimJava that was briefly discussed in the related work section.

[16] Urbán P., Défago X., Schiper A., *Neko: A single environment to simulate and prototype distributed algorithms*, Proceedings of the 15th International Conference on Information Networking, Beppu City, Japan, 2001.

Provided general information on the Neko project, a project that extended and uses SimJava, presented in the related work section.

[17] The JavaSim user guide, The University of Newcastle Upon Tyne Computing Laboratory 1999, pages 18, 19.

Provided the source for examining the PRNG used by JavaSim. This was one of the PRNGs that were considered as candidates for the new version of SimJava.

[18] Modelling and Simulation course notes, by Jane Hillston, School of Informatics, University of Edinburgh.

An excellent background source. Several Markovian modelling techniques are discussed here as well as simulation as a modelling approach, including a detailed discussion of SimJava. Very useful was the comparison between the two approaches as well as the discussion of respective benefits and shortcomings. Most of the background chapter was based on these notes.

[19] The Swarm user guide, The Swarm Development Group 2000, Appendix C Section C.3.1.

This was the source for numerous PRNG test results. These results were not related to the studies in [3][4][5] concerning the PRNG selected for SimJava, but were rather used to establish the qualities of the Mersenne twister [14] as a possible generator choice.

[20] ACME Laboratories, http://www.acme.com, class GifEncoder.

This site provided the source for the GifEncoder class, used by the SimJava Graph Viewer to store graphs as GIF images.

[21] Sundar Dorai-Raj homepage, The Department of Statistics, Virginia Tech, http://www.stat.vt.edu/~sundar/java, class StatFunctions.

This site provided the source for the Student's-t and Normal quantile calculation methods. These were adapted for use by SimJava's output analysis methods to calculate confidence intervals for any number of replications or batches. An online Chi-square quantile calculation method found here was also used to test the Serial test results presented in the project's evaluation.

[22] The JavaHASE homepage, The Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh, http://www.dcs.ed.ac.uk/home/hase/javahase.

This is the source that provided general information on JavaHASE, one of the latest SimJava-influenced efforts of the HASE project that was presented in the related work section.

[23] The HASE homepage, The Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh, http://www.dcs.ed.ac.uk/home/hase.

Provided useful information on the HASE project that was used to discuss HASE in the related work section.

[24] The SimJava homepage, The Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh, http://www.dcs.ed.ac.uk/home/hase/simjava.

The homepage of SimJava. Most of the SimJava-related material was obtained from here, as well as links to projects that use or have extended SimJava.

Appendix A: PRNG test programs

1. Chi-square test (`ChiSquareTest.java`)

```
import eduni.simjava.distributions.Sim_random_obj;
import java.io.*;

public class ChiSquareTest {
  public static void main(String args[]) {
    if (args.length != 3) {
      System.out.println("Usage: java ChiSquareTest <#intervals> <#samples> <seed>");
      System.out.println("Where:");
      System.out.println("   - <#intervals>: The number of intervals.");
      System.out.println("   - <#samples>: The number of samples.");
      System.out.println("   - <seed>: The root seed for the generator.");
      System.exit(0);
    }
    int intervals = -1, samples = -1, seed = -1;
    // Parse arguments
    try {
      intervals = Integer.parseInt(args[0]);
      samples = Integer.parseInt(args[1]);
      seed = Integer.parseInt(args[2]);
    } catch (NumberFormatException nfe) {
      System.out.println("Non numeric parameters");
      System.exit(0);
    }
    // Initialise generator
    Sim_random_obj source = new Sim_random_obj("Test", seed);
    // Obtain samples
    double expected = ((double)samples)/((double)intervals);
    int[] observed = new int[intervals];
    double width = 1.0/intervals;
    double s;
    boolean found;
    for (int i=0; i < samples; i++) {
      s = source.sample();
      found = false;
      for (int j=1; (j <= intervals) && !found; j++) {
        if (((s > (j-1)*width) && (s < j*width))) {
          observed[j-1] += 1;
          found = true;
        }
      }
    }
    // Print out results
    try {
      PrintWriter out = new PrintWriter(new FileOutputStream("results_X2"));
      out.println("INTERVAL - OBSERVED - EXPECTED - VALUE");
      out.println();
      double result = 0.0;
      for (int i=0; i < intervals; i++) {
        double value = Math.pow((observed[i]-expected), 2.0)/expected;
        result += value;
        out.println("[" + (i+1) + "]    " + observed[i] + "    " + expected + "    " + value);
      }
      out.println();
      out.println("TOTAL RESULT: " + result);
      int df = intervals-1;
      out.println("DEGREES OF FREEDOM: " + df);
      System.out.println("Total result: " + result);
      System.out.println("Degrees of freedom: " + df);
      out.flush();
      out.close();
    } catch (IOException ioe) {
      System.out.println("Error while writing results");
    }
  }
}
```

2. Kolmogorov-Smirnov test (`KSTest.java`)

```java
import eduni.simjava.distributions.Sim_random_obj;

public class KSTest {
  public static void main(String args[]) {
    if (args.length != 2) {
      System.out.println("Usage: java KSTest <#samples> <seed>");
      System.out.println("Where:");
      System.out.println("   - <#samples>: The number of samples.");
      System.out.println("   - <seed>: The root seed for the generator.");
      System.exit(0);
    }
    long seed = -1;
    int count = -1;
    try {
      count = Integer.parseInt(args[0]);
      seed = Long.parseLong(args[1]);
    } catch (NumberFormatException nfe) {
      System.out.println("Non numeric parameters");
      System.exit(0);
    }
    // Obtain samples
    Sim_random_obj source = new Sim_random_obj("Test", seed);
    double[] samples = new double[count];
    for (int i=0; i < count; i++) {
      samples[i] = source.sample();
    }
    // Sort the samples (BubbleSort)
    for (int i=samples.length; --i>=0;) {
      for (int j=0; j < i; j++) {
        if (samples[j] > samples[j+1]) {
          double temp = samples[j];
          samples[j] = samples[j+1];
          samples[j+1] = temp;
        }
      }
    }
    // Get the maximum values
    double kplus = -1.0;
    double kminus = -1.0;
    double kp, km;
    for (int i=0; i < count; i++) {
      kp = ((i+1.0)/((double)count)) - samples[i];
      km = samples[i] - ((double)i)/((double)count);
      if (kp > kplus) {
        kplus = kp;
      }
      if (km > kminus) {
        kminus = km;
      }
    }
    // Calculate K+ and K-
    kplus = Math.sqrt(count) * kplus;
    kminus = Math.sqrt(count) * kminus;
    // Print out results
    System.out.println("K+ = " + kplus);
    System.out.println("K- = " + kminus);
    System.out.println("Degrees of freedom: " + count);
  }
}
```

3. Serial-correlation test (`SCTest.java`)

```
import eduni.simjava.distributions.Sim_random_obj;

public class SCTest {
  public static void main(String args[]) {
    if (args.length != 4) {
      System.out.println("Usage: java SCTest <lagmin> <lagmax> <#samples> <seed>");
      System.out.println("Where:");
      System.out.println("   - <lagmin>: The minimum lag.");
      System.out.println("   - <lagmax>: The maximum lag.");
      System.out.println("   - <#samples>: The number of samples.");
      System.out.println("   - <seed>: The root seed for the generator.");
      System.exit(0);
    }
    long seed = -1;
    int count = -1, lagmin = -1, lagmax = -1;
    try {
      lagmin = Integer.parseInt(args[0]);
      lagmax = Integer.parseInt(args[1]);
      count = Integer.parseInt(args[2]);
      seed = Long.parseLong(args[3]);
    } catch (NumberFormatException nfe) {
      System.out.println("Non numeric parameters.");
      System.exit(0);
    }
    // Obtain samples
    Sim_random_obj source = new Sim_random_obj("Test", seed);
    double[] samples = new double[count];
    for (int i=0; i < count; i++) {
      samples[i] = source.sample();
    }
    // Get results
    for (int lag=lagmin; lag <= lagmax; lag++) {
      double Rk = 0.0;
      for (int i=0; i < count-lag; i++) {
        Rk += (samples[i] - 0.5)*(samples[i+lag] - 0.5);
      }
      Rk = Rk/((double)(count-lag));
      // Hardwired For 90% confidence level (1.645)
      System.out.println("Lag: " + lag);
      double val = 1.645/(12.0*Math.sqrt(count-lag));
      System.out.println("Interval: ["+(Rk-val)+", "+(Rk+val)+"]");
    }
  }
}
```

4. Serial test in two dimensions (SerialTest2D.java)

```
import eduni.simjava.distributions.Sim_random_obj;
import java.io.*;

public class SerialTest2D {
  public static void main(String args[]) {
    if (args.length != 3) {
      System.out.println("Usage: java SerialTest2D <#intervals> <#points> <seed>");
      System.out.println("Where:");
      System.out.println("   - <#intervals>: The number of cells on each dimension.");
      System.out.println("   - <#points>: The number of 2-dimensional points.");
      System.out.println("   - <seed>: The root seed for the generator.");
      System.exit(0);
    }
    int intervals = -1, points = -1, seed = -1;
    // Parse arguments
    try {
      intervals = Integer.parseInt(args[0]);
      points = Integer.parseInt(args[1]);
      seed = Integer.parseInt(args[2]);
    } catch (NumberFormatException nfe) {
      System.out.println("Non numeric parameters");
      System.exit(0);
    }
    // Initialise generator and obtain samples
    Sim_random_obj source = new Sim_random_obj("Test", seed);
    double expected = ((double)points)/Math.pow(intervals, 2.0);
    int[][] observed = new int[intervals][intervals];
    double width = 1.0/intervals;
    double s1, s2;
    boolean found;
    for (int i=0; i < points; i++) {
      s1 = source.sample();
      s2 = source.sample();
      System.out.println(s1 + "         " + s2);
      found = false;
      for (int j=1; (j <= intervals) && !found; j++) {
        for (int k=1; (k <= intervals) && !found; k++) {
          if (((s1 > (j-1)*width) && (s1 < j*width)) &&
              ((s2 > (k-1)*width) && (s2 < k*width))) {
            observed[j-1][k-1] += 1;
            found = true;
          }
        }
      }
    }
    // Print out results
    try {
      PrintWriter out = new PrintWriter(new FileOutputStream("results_2D"));
      out.println("CELL - OBSERVED - EXPECTED - VALUE");
      out.println();
      double result = 0.0;
      for (int i=0; i < intervals; i++) {
        for (int j=0; j < intervals; j++) {
          double value = Math.pow((observed[i][j]-expected), 2.0)/expected;
          result += value;
          out.println("[" + (i+1) + "," + (j+1) + "]    " + observed[i][j] + "    " +
                      expected + "    " + value);
        }
      }
      out.println();
      out.println("TOTAL RESULT: " + result);
      int df = ((int)Math.pow(intervals, 2.0))-1;
      out.println("DEGREES OF FREEDOM: " + df);
      System.out.println("Total result: " + result);
      System.out.println("Degrees of freedom: " + df);
      out.flush();
      out.close();
    } catch (IOException ioe) {
      System.out.println("Error while writing results");
    }
  }
}
```

Appendix B: Comparison study simulations

1. Simulation for new version (`SimpleTest.java`)

```
import eduni.simjava.*;
import eduni.simjava.distributions.*;

class Source extends Sim_entity {
  Sim_stat stat;
  Sim_random_obj prob;
  Sim_negexp_obj delay;
  int cpu_count;
  double prob_inc;
  double end_time;

  Source(String name, double mean, int count, double end_time) {
    super(name);
    prob = new Sim_random_obj("Probability", 4851);
    delay = new Sim_negexp_obj("Delay", mean, 4851);
    add_generator(prob);
    add_generator(delay);
    this.end_time = end_time;
    cpu_count = count;
    prob_inc = 1.0/(double)count;
    stat = new Sim_stat();
    stat.add_measure("Generation rate", Sim_stat.RATE_BASED);
    stat.set_efficient("Generation rate");
    set_stat(stat);
  }

  public void body() {
    while (Sim_system.sim_clock() < end_time) {
      double sample = prob.sample();
      for (int i=1; i <= cpu_count; i++) {
        if (sample < i*prob_inc) {
          sim_schedule(i, 0.0, 0);
          stat.update("Generation rate", Sim_system.sim_clock());
          break;
        }
      }
      sim_pause(delay.sample());
    }
  }
}

class CPU extends Sim_entity {
  Sim_stat stat;
  Sim_normal_obj delay;

  CPU(String name, double mean, double var) {
    super(name);
    delay = new Sim_normal_obj("Delay", mean, var, 4851);
    add_generator(delay);
    stat = new Sim_stat();
    stat.add_measure(Sim_stat.UTILISATION);
    stat.add_measure(Sim_stat.SERVICE_TIME);
    stat.set_efficient(Sim_stat.UTILISATION);
    stat.set_efficient(Sim_stat.SERVICE_TIME);
    set_stat(stat);
  }

  public void body() {
    while (Sim_system.running()) {
      Sim_event e = new Sim_event();
      sim_get_next(e);
      sim_process(delay.sample());
      sim_completed(e);
    }
  }
}
```

```
public class SimpleTest {
  public static void main(String args[]) throws Exception {
    long start_time = System.currentTimeMillis();
    long start = Runtime.getRuntime().totalMemory() -
                 Runtime.getRuntime().freeMemory();
    Sim_system.initialise();
    int count = Integer.parseInt(args[0]);
    double term_time = Double.parseDouble(args[1]);
    Source source = new Source("Source", 150.0, count, term_time);
    for (int i=1; i <= count; i++) {
      CPU cpu = new CPU("CPU_" + String.valueOf(i), 140.5, 74.5);
    }
    Sim_system.run();
    long end_time = System.currentTimeMillis();
    long end = Runtime.getRuntime().totalMemory() -
               Runtime.getRuntime().freeMemory();
    System.out.println("############## RESULTS ##############");
    System.out.println("TIME:");
    System.out.println("  " + (end_time-start_time));
    System.out.println("MEMORY:");
    System.out.println("  " + (end-start));
  }
}
```

2. Simulation for original version (`SimpleTestOld.java`)

```
import eduni.simjava.*;
import java.util.Random;
import java.io.*;

class Source extends Sim_entity {
  Random prob;
  Sim_negexp_obj delay;
  int cpu_count;
  double prob_inc;
  double end_time;

  Source(String name, double mean, int count, double end_time) {
    super(name);
    prob = new Random(4851);
    delay = new Sim_negexp_obj("Delay", mean, 4851);
    this.end_time = end_time;
    cpu_count = count;
    prob_inc = 1.0/(double)count;
  }

  public void body() {
    while (Sim_system.sim_clock() < end_time) {
      double sample = prob.nextDouble();
      for (int i=1; i <= cpu_count; i++) {
        if (sample < i*prob_inc) {
          sim_schedule(i, 0.0, 0);
          sim_schedule(Sim_system.get_entity_id("Monitor"), 0.0, 0);
          break;
        }
      }
      sim_hold(delay.sample());
    }
    for (int i=1; i <= cpu_count; i++) {
      sim_schedule(i, 0.0, -1);
    }
    sim_schedule(Sim_system.get_entity_id("Monitor"), 0.0, -1);
  }
}

class CPU extends Sim_entity {
  Sim_normal_obj delay;

  CPU(String name, double mean, double var) {
    super(name);
    delay = new Sim_normal_obj("Delay", mean, var, 4851);
  }

  public void body() {
    int my_id = get_id();
    while (true) {
      Sim_event e = new Sim_event();
      sim_get_next(e);
      if (e.get_tag() == -1) {
        break;
      }
      double delay_time = delay.sample();
      sim_hold(delay_time);
      sim_schedule(Sim_system.get_entity_id("Monitor"), 0.0,
                   my_id, new Double(delay_time));
    }
  }
}

class Monitor extends Sim_entity {

  int count;
  int source_generations = 0;
  double[] util_time,
           service_min,
           service_max;
  int[] interval_counter;
  double end_time;
  Monitor(String name, int count) {
    super(name);
    util_time = new double[count];
    interval_counter = new int[count];
    service_min = new double[count];
    service_max = new double[count];
    for (int i=0; i < count; i++) {
      service_min[i] = Double.POSITIVE_INFINITY;
      service_max[i] = Double.NEGATIVE_INFINITY;
    }
    this.count = count;
  }

  public void body() {
    boolean keep_running = true;
    while (keep_running) {
      Sim_event e = new Sim_event();
      sim_get_next(e);
      int tag = e.get_tag();
      switch (tag) {
        case -1: // Simulation complete
          end_time = Sim_system.sim_clock();
          output_report();
          keep_running = false;
          break;
        case  0: // From the source
          source_generations++;
          break;
        default: // From a cpu
          double interval = ((Double)e.get_data()).doubleValue();
          util_time[tag-1] += interval;
          interval_counter[tag-1] += 1;
          if (interval < service_min[tag-1]) {
            service_min[tag-1] = interval;
          }
          if (interval > service_max[tag-1]) {
            service_max[tag-1] = interval;
          }
          break;
      }
    }
  }

  private void output_report() {
    try {
      PrintWriter out = new PrintWriter(new FileOutputStream("sim_output"));
      out.println("#################################");
      out.println("#       SIMULATION RESULTS       #");
      out.println("#################################");
      out.println();
      out.println("Source:");
      out.println("- Generation rate");
      out.println("  + Average: " + (((double)source_generations)/end_time));
      out.println("  + Count:   " + source_generations);
      out.println();
      for (int i=0; i < count; i++) {
        out.println("Disk " + (i+1) + ":");
        out.println("- Utilisation");
        out.println("  + Average: " + (util_time[i]/end_time));
        out.println("- Service time");
        out.println("  + Average: " + (util_time[i]/interval_counter[i]));
        out.println("  + Minimum: " + service_min[i]);
        out.println("  + Maximum: " + service_max[i]);
        out.println();
      }
      out.flush();
      out.close();
    } catch (IOException ioe) {
      System.out.println("Unable to create the report file.");
    }
  }
}

public class SimpleTestOld {
  public static void main(String args[]) throws Exception {
    long start_time = System.currentTimeMillis();
    long start = Runtime.getRuntime().totalMemory() -
                 Runtime.getRuntime().freeMemory();
    Sim_system.initialise();
    int count = Integer.parseInt(args[0]);
    double term_time = Double.parseDouble(args[1]);
    Source source = new Source("Source", 150.0, count, term_time);
    for (int i=1; i <= count; i++) {
      CPU cpu = new CPU("CPU_" + String.valueOf(i), 140.5, 74.5);
    }
    Monitor monitor = new Monitor("Monitor", count);
    Sim_system.set_auto_trace(false);
    Sim_system.run();
    long end_time = System.currentTimeMillis();
    long end = Runtime.getRuntime().totalMemory() -
               Runtime.getRuntime().freeMemory();
    System.out.println("############## RESULTS ##############");
    System.out.println("TIME:");
    System.out.println("  " + (end_time-start_time));
    System.out.println("MEMORY:");
    System.out.println("  " + (end-start));
  }
}
```

Appendix C: Comparison study test results

1. Speed tests (results in milliseconds)

Version	Entities							
	100	200	300	400	500	600	700	800
Efficient	59400	69423	87040.7	97815.3	111940.7	132648.3	136413.7	149835.3
Detailed	61745.3	76198.7	89662.3	101206	116551	128259.7	143799.7	163718
Original	96166.7	133415.3	181382.7	258758.7	354810.3	485709	596957.3	756054

Version	Run length (millions of time units)							
	5	10	15	20	25	30	35	40
Efficient	32845.7	52684.7	71550.3	95197.7	119794.7	131476	158232.7	179310
Detailed	32534	55111.3	74050.7	94529	116566.7	133958.3	156188.3	177878
Original	80365	139438	198609.7	302769	328341	413877.3	465221	534715.7

2. Memory usage tests (results in bytes)

Version	Entities							
	100	200	300	400	500	600	700	800
Efficient	408912	693520	976392	778240	1195120	1347000	2908752	2359392
Detailed	27830176	30199000	28974296	28370664	29485776	27046104	28538416	28684688
Original	119256	419088	198216	532936	398256	591152	561528	516872

Version	Run length (millions of time units)							
	5	10	15	20	25	30	35	40
Efficient	1354997	1285533	1031269	1339669	1193792	1417258	1316885	1251586
Detailed	6673848	12766008	16548480	24718320	29485664	36039992	41728752	45652320
Original	480152	487576	718520	729816	398312	479648	693400	550280