UNIVERSITY OF SOUTHAMPTON

A Simulation Study of the Effectiveness of an Aircraft Operations Logistic Support Package

Colin Benford BSc

Master of Philosophy

Faculty Of Mathematics

February 2001

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF MATHEMATICS

OPERATIONAL RESEARCH

Master of Philosophy

A SIMULATION STUDY OF THE EFFECTIVENESS OF AN AIRCRAFT OPERATIONS LOGISTIC SUPPORT PACKAGE

by Colin Benford

ABSTRACT

Military organisations today operate small fleets of unique aircraft and need to be sure that the spares purchased to support operations meet the organisations needs whilst remaining the minimum necessary to minimise unnecessary government expenditure. Historically this task was undertaken by using historical consumption as the basis of the calculation. This is not seen as appropriate today and a range of deterministic models are used to produce the spares lists. However, their failure to apply a particular flying programme means that the output is viewed with some scepticism by military staffs. Simulation provides the means to apply that flying programme and, moreover, allows a series of what if evaluations to be undertaken.

This thesis covers the work undertaken by myself to design and produce a suitable simulation application to meet the above requirement. Whilst data was available it was of a simple form without sufficient fidelity to allow the underlying distributions to be derived. Consequently, the opportunity to examine the effect of applying different distributions for both failure and repair times was taken allowing the scope of the work to broaden. Having produced the simulation a number of alternative flying programmes were simulated to identify their impact on the overall achievement and aircraft availability.

This work has allowed me to not only provide a model which can be used with a deterministic application to assess the validity of the spares list but, has also allowed investigation into the effect of applying different distributions to both failure and repair times.

LIST OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

32. Mean Aircraft Availability for the 1920 Flights Option.

33. 36 Aircraft Mean Daily Flights Flown for the 1920 Flights Option.

34. 36 Aircraft Mean Daily Aircraft Availability for the 1920 Flights Option.

35. Flights Flown for the Variable Flights Option.

36. Percentage of Flights Flown for the Variable Flights Option.

37. Variance for the Variable Flights Option.

38. Upper and Lower 90% Confidence Boundaries for the Variable Flights Option.

39. Distance of the Lower Confidence Limit from the Mean for the Variable Flights Option.

LIST OF APPENDICES

ACKNOWLEDGEMENTS

My thanks to Professor Lillian Barras for her encouragement to go ahead with undertaking this study and Dr Arjin Shahani and Professor Russell Cheng for their help and guidance throughout the research and production of this thesis.

LIST OF ABBREVIATIONS

ALSSim	Analysis of Logistic support Simulation model.
LITS	Logistic Information Technology System. The new computer system being introduced by the RAF to collect and analyse failure and repair data for LRIs. This system will replace the MDC.
LRI	Line Replaceable Items. The repairable components removed from the aircraft.
MDC	Maintenance Data Computer. The computer system used by the RAF to collect and analyse failure and repair data for LRIs.
MTBF	Mean Time Between Failures. The average time that it takes a given LRI type to fail in use.
MTTR	Mean Time To Repair. The average time taken to repair a given LRI.
MWO	Maintenance Work Order. The document completed by a technician which details the work done and the time taken to complete the task.
RAF	Royal Air Force.
SA	Support Authority. The organisation that decides the engineering support policy for the aircraft and its components.
USAF	United States Air Force

INTRODUCTION

Since its creation on 1 April 1918, the RAF has faced the problem of how to provide sufficient spares and repair facilities at the right place to ensure maximum operational availability whilst minimising the support costs. Although the model for support inherited from the Army met the initial needs, as the complexity of aircraft has grown so the RAF has had to develop its own resupply model. Today the RAF makes use of a suite of highly capable deterministic computer tools designed to meet the unique needs of a high tech military operation. However, there remains a lack of confidence in the fidelity of the model's solutions. This is because the model gives an output in terms of general system availability rather than the probability that a particular target flying programme can be met. To achieve this level of confidence it is necessary to use a model which produces the results in a form that is meaningful to the operational customer.

THE DEVELOPMENT OF LOGISTIC SUPPORT WITHIN THE RAF

During the First World War the support of aircraft was relatively simple due to four factors. Firstly, the aircraft in use were of limited complexity and many of the spares required were manufactured at Aircraft Support Parks close to the Operational Squadrons [1]. Secondly, as most aircraft operated from airfields located in Western France the logistical support tail to the United Kingdom was quite short. Thirdly, a policy of cannibalising war and accident damaged aircraft was used as a means of increasing the spares available at the operating bases. Finally, the rate of development of new aircraft types was rapid with most types having an effective in-service life in the range of 10 months.

With the end of the First World War the pressing need for constant improvement of aircraft was removed. During the 1920s and 1930s the RAF took on the responsibility of policing the remote areas of the Empire and for the first time experienced major problems with the provision of spares. Aircraft complexity was increasing, operations were mounted thousands of miles from the manufacturing base and the lack of effective hostile fire reduced the availability of damaged aircraft for cannibalisation. Two problems had to be resolved in order that the RAF provide effective support to its widely dispersed operations. Firstly, there was a need to provide well equipped support facilities in-theatre capable of manufacturing and repairing a wide range of components with an extensive

spares holding. Secondly, there was a need to ensure that the lengthy supply line from the United Kingdom did not adversely affect aircraft availability. The first problem was solved by developing the Aircraft Support model from the First World War into larger centralised maintenance establishments. The second was more difficult to solve in that it required logisticians to attempt to guess future requirements and the model adopted was to assume that the known historical demands were representative of future requirements.

The early stages of the Second World War had a close correlation with operations in the First World War in that the majority of operations were close to the United Kingdom manufacturing base. Indeed, after the fall of France and throughout the Battle of Britain the supply pipeline was extremely short. However, there were aspects of this war as it continued that made support different. Firstly, the complexity of aircraft was much greater which resulted in greater reliance on industry support. Secondly, there was a need to support operations in the Middle and Far East and this placed a considerable strain on the ability to provide effective support in those theatres. As the War developed the RAF found that it was fighting a highly mobile war and consequently spent much time and money redeveloping the mobile support sites that had been the norm in the First World War.

The end of the Second World War saw the introduction of the Jet engine and the start of the period of history known as the Cold War. Aircraft complexity was vastly greater than had been the case hitherto and the cost of the aircraft combined with the time taken to develop new aircraft meant that Air Forces were now operating aircraft types for several decades. Moreover, the cost of spares were many times higher than hitherto and it was recognised that there was a need to find an effective method of determining the correct number of spares required and the most appropriate location to store them. Early models made use of a Poisson distribution applied to the failure times for components, the required shelf satisfaction rate and the known or anticipated repair time. From this was derived the list of spares required and the expected cost of these spares. As is often the case, the use of computer models to ascertain the spares required to support aircraft operations was met with some scepticism and was only accepted into general use within the USAF once it had proved its worth in a practical field test at George Air Force base in 1966 [2]. These early models, and the algorithms contained within them, form the basis of all the deterministic scaling models in use within the USAF and RAF today.

PROBLEM BACKGROUND

In common with most other nation's air forces, the RAF operates relatively small fleets of unique aircraft. This means that, unlike commercial airlines whose use of similar aircraft means that they can purchase shares cheaply, the RAF is forced to procure small stocks of expensive spares to support its operations. Moreover, the need to operate at remote locations worldwide has meant that those spares can spend considerable time within the pipeline between those bases and the centralised repair locations. For this reason a great deal of time and money has been expended over many years to derive not only the appropriate number of spares to purchase but also where best to store them when not required for use. Although it would appear to be most efficient to hold those stocks forward, the need for expensive repair equipment and warehouse facilities in theatre often precludes that option. Thus the RAF has developed a system whereby repairs are undertaken at the location where the cost is minimised. Thus, items with a low cost of repair are repaired at the base and repairs that are high cost or require specialist skills and/or equipment are carried out in industry. Similarly, spares which fall into the category of high turnover are held on operational units with other spares held at the centralised depot. A deterministic sparing model is used to derive the best location to store items and the quantities of any given item at each location.

The Royal Air Force Support Chain

The RAF engineering support infrastructure is organised as a series of levels, known as lines, with each undertaking different depth of maintenance on Line Replaceable Items (LRIs) dependent on the policy set by the aircraft's engineering Support Authority (SA). The first line is at the aircraft operating squadron and the only work undertaken here is replacement of complete LRIs. The second line is located on the operating base and undertakes some limited deeper rectification work comprising of module and circuit board level changes as appropriate. Third and fourth line undertake deeper component repairs, the difference between the two being that third line is manned by government employees whilst fourth line is industry. The decision as to whether the repair is undertaken at third or fourth line is taken by the aircraft SA and takes account of such factors as the repair capacity at third line, the cost of repair at fourth line and the cost of any specialist repair equipment.

Figure 1 shows a representation of the simplified logistic cycle for military aircraft operations. The aircraft crew identifies that a system is inoperative and report the failure to the squadron's technical staff. The system is tested by the first line ground crew and the defective component removed from the aircraft and replaced with a serviceable item from stock. The removed LRI is then passed to second line where the item is either repaired and returned to stock if it is within the capability of the bay or returned to third or fourth line as appropriate if deeper repair is required. Those items returned to the deeper repair lines line are repaired there and returned to stock.

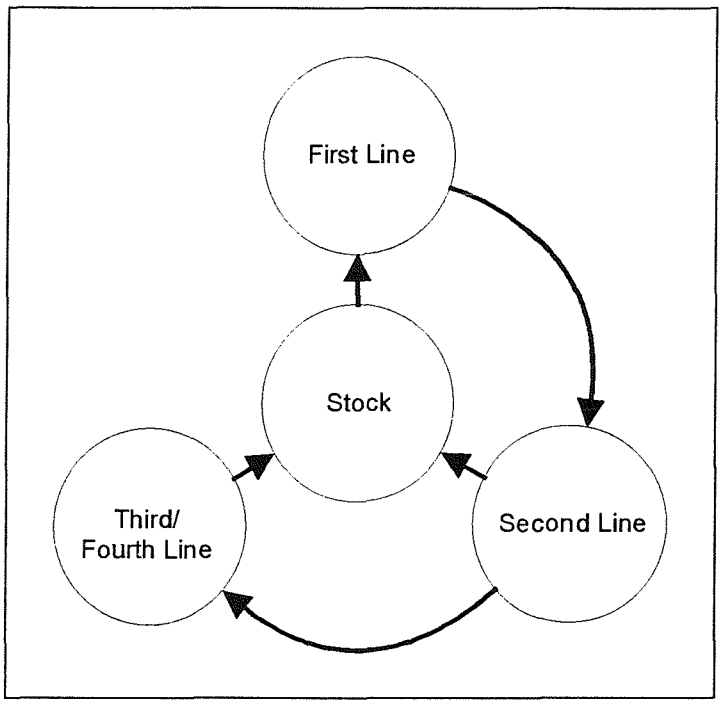


Figure 1. The Simplified Logistic Cycle.

The deterministic models in use within the defence modelling institutions are powerful applications which make use of accepted marginal analysis algorithms to assess the effect on the overall system availability of a purchase of one of each spare in isolation. Having done so, the spare with most effect on system availability is added to the purchase list and the same process repeated. This continues until the model achieves the desired system availability or the maximum cost limit is reached.

The Use of Simulation to Support the Outputs Of Deterministic Models

There are, however, two main limitations to deterministic models which have given rise to the desire to use simulation to support the findings of the model. The first relates to the way that the model addresses system utilisation. Having been given the target e.g. number

of flying hours per aircraft per month the model assesses the amount of that period that the system is available and uses this to produce the overall system availability. In real life, the operator is interested not in this overall system availability but that which pertains to the time that the aircraft is required. Thus an average system availability of say 75% appears at first examination to give a good result. However, this is of little use if the aircraft if the normal flying day is between 8:00 am and 5:00 pm and the aircraft are never available between 8:00 am and 2:00 pm. Secondly, the mathematical model used within the application does not assess the ability of the support model being analysed to support a particular flying programme. The use of a simulation allows an assessment of the validity of the deterministic model's output against a particular flying programme to derive the mission success rate of a particular scale. This allows the operational customer to have confidence that the proposed scale will meet the operational needs.

THE ANALYSIS OF LOGISTIC SUPPORT SIMULATION (ALSSIM)

The Design Process

There is a tendency for much of the literature dealing with the use of simulation as a tool to problem solving to assume that the decision has already been made that simulation is the most appropriate means to examine the problem being considered. Therefore, they deal extensively with the issues to be considered when designing and using the simulation model and largely ignore the broader aspects of the methods that should be employed to address the whole problem from its inception to the presentation of the results. Simulation is not the only way to arrive at the solution and often the most appropriate method is one or more practical trials of each of the various solutions. The analyst must always remember that a simulation is merely a theoretical representation of the actual environment and should, therefore, only be undertaken when actual trials of the alternative options are either inappropriate or not practical. For a simple problem such as a shopkeeper wanting to ascertain whether the sales of bread would be increased if the location of the product within the store was changed a practical trial would clearly show the effect of moving stock to a new location. However, employing an additional member of staff for 4 weeks to identify whether the addition of another till would improve sales by reducing queuing time would not make commercial sense and in this case a relatively simple simulation would be an appropriate way to undertake the study. Thus, before designing a simulation to solve a problem, the following questions should be considered:

Could this problem be solved by physical experimentation?

Is the solution by physical experimentation practical?

Is the use of simulation appropriate?

Only once these questions have been satisfactorily answered should the problem of the design of the simulated environment be considered. Once the decision has been taken that simulation is the best option there is a need for a structured approach to the design and implementation of the model which is best achieved by adopting a process such as that shown below at figure 2. The version shown here is somewhat simplified and in practice this would be an iterative process incorporating tests and feed back loops throughout with each iteration adding to the knowledge base until the end product provides an acceptable representation of the environment being examined [3].

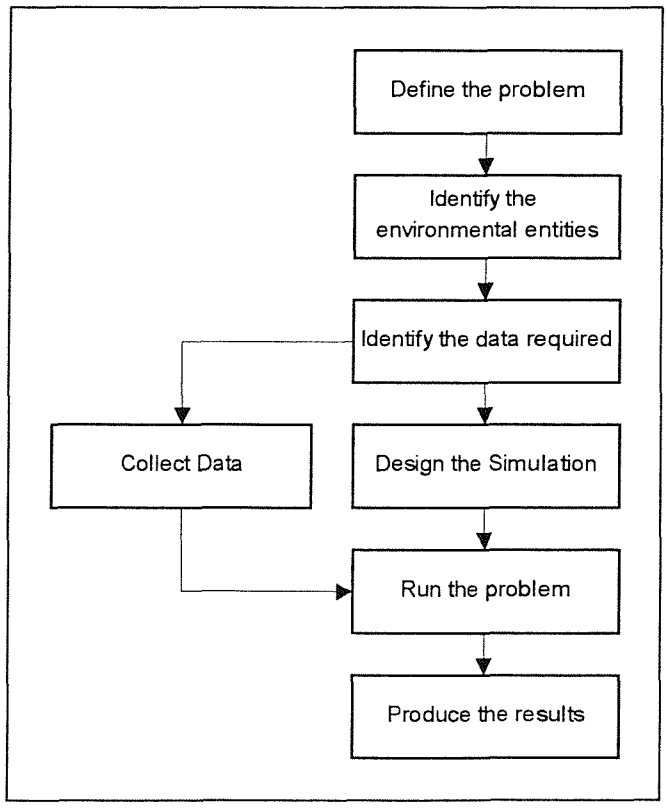


Figure 2. The Simulation Design Process.

Before addressing the specifics of the design of ALSSim it is necessary to examine the three fundamental questions raised above to decide whether the use of simulation as a means to assess the spares scaling produced by the RAF's deterministic scaling model is practical and, more importantly, appropriate. Clearly, as demonstrated in the trial

undertaken by the USAF in 1965 [4] it is possible to undertake a practical trial to confirm the validity of the scale of spares. However, this test was undertaken over a period of 6 months to ensure that short term variability in the demand levels was evened out. Thus, although possible, the use of physical trials is not a practical solution to the need to assess the impact of derived scales of spares. A shorter duration for a test would not allow a representative demand population to be achieved as the arising rates for individual LRIs tends to be very low and there is a high risk that decisions on the final composition and distribution of the spares could be taken on the basis of a skewed demand distribution rather than a representation of a steady state demand profile. Therefore, the use of simulation in this case is highly appropriate in that it allows a simulated trial of a protracted operational period to be undertaken in a very short time ensuring that short term transients in the demand distribution are damped out allowing the analyst to take a view based on the steady state requirement for spares.

Having concluded that simulation is a viable way of examining the problem, it is worth examining the general design of the ALSSim application. Figure 3 shows this in terms of the input files required and the output file from the simulation. There are 3 main factors which will have an influence on the final results of the simulation: the target flying programme, the LRI reliability and the LRI scales to be examined. This data is input into the simulation by means of reading in files prepared off-line thus reducing the time taken to run the overall simulation problem. The results are output to a single text file allowing the use of whatever word processing or spreadsheet application is to analyse the data.

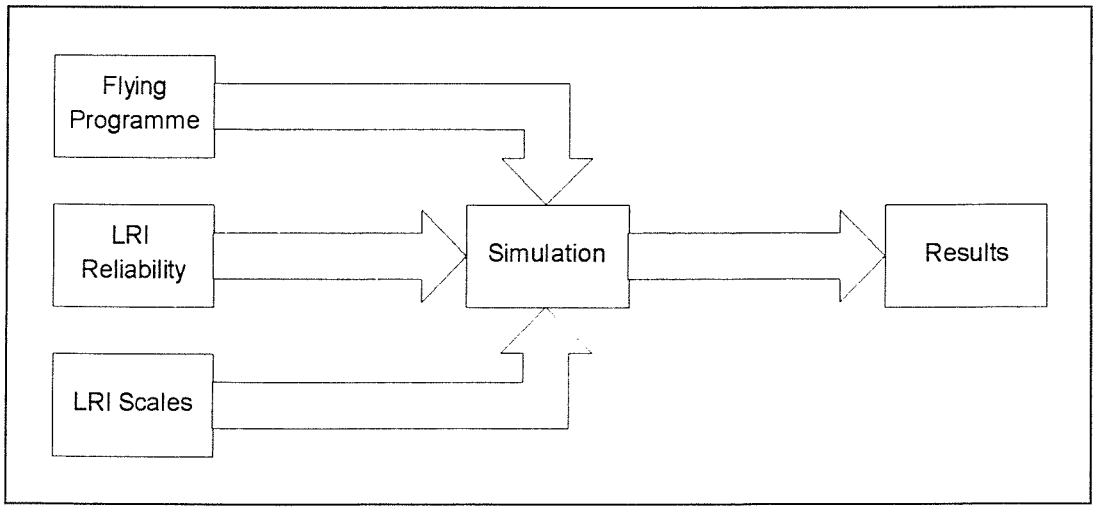


Figure 3. ALSSim Generic Design.

Entities

In any operational research study making use of simulation as the modelling tool it is necessary to determine the elements of the model and in particular which of these need to be modelled as distinct entities and which can be simply represented as variables. Moreover, it is necessary to ensure that a balance is reached between tracking too many individual entities leading to the simulation running too slow and too few reducing the quality of the solution to an unacceptably low level. The basis of any discrete event simulation is the selection of the core entities which are required to represent the system to be analysed. These entities may either be permanent or temporary and could take the form of physical items e.g. a hospital bed or a queue such as that of customers awaiting service. In the latter case, the entity may be created several times during the course of the model running or could exist throughout the entire run. Therefore, it is not of importance whether an entity is temporary or permanent but rather that it will exist within the simulation and that it will interact with other entities. The selection of the correct items to be dealt with as entities within the simulation is crucial if the application is to run quickly and the selection of something as an entity must depend on its relevance to the problem being considered. Thus, in the case of a hospital simulation the consideration of individual patients may not be of relevance and, therefore, the model would not include an entity of patient although of course we all recognise that each patient exists as an individual in the actual hospital.

In this study the main candidate entities were as follows: pre flight servicing teams, missions, aircraft, and LRIs. The criteria used to decide whether to use a variable or to have distinct entities were as follows:

> Are the states for the entity digital or can it have a number of states?
> Does the entity have permanence throughout the simulation?
> Does the entity move from one location to another in the course of the simulation?

Application of these criteria led to the conclusion that the pre flight servicing teams were either occupied or available and, therefore, could be modelled effectively by a counter. The lack of permanence for the missions coupled with the fact that they were the main input parameter setting the simulation target meant that they were best modelled as non-permanent entities. The aircraft were core elements of the simulation which existed

throughout it and could take a range of states and thus, they are dealt with as permanent entities. Finally, as the LRIs were components whose simulation location would change throughout the simulation they could be modelled by means of individual entities. However, further analysis revealed that to model each individual LRI would not only lead to excessive memory overheads but also a considerable reduction in execution speed. For this reason it was decided that there should be two separate entities for LRIs, one to cover the positions on the aircraft and one to deal with them when removed from the aircraft. In the latter case, the entity concerned LRI types containing counters for the various states and locations thus reducing considerably the amount of memory required.

The Mission Entity

The mission requirements are read in from a data file and placed within the event queue such that they are called at the relevant time within the simulation run. Each mission event comprises of a particular number of flights to be flown, a target take off time and a mission duration. For a mission to be launched there must be sufficient aircraft available for tasking for all of the flights contained within it. Once the mission has either been launched or cancelled it is discarded. The state diagram for the mission entity is as shown in figure 4.

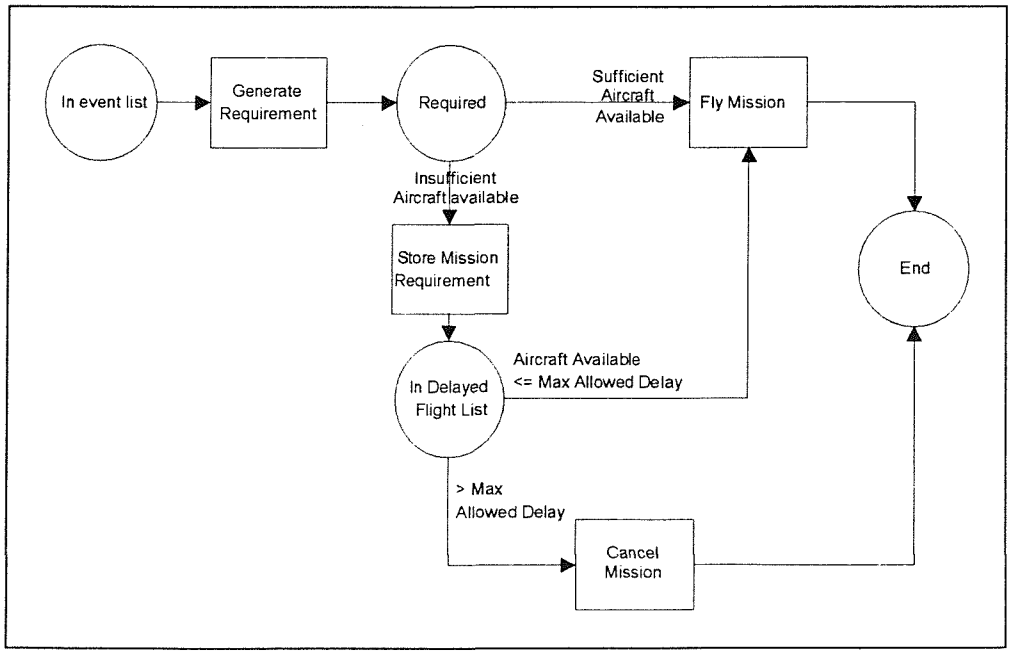


Figure 4. The Mission Entity.

The mission event is loaded by the simulation controller and if sufficient serviceable aircraft are available the number of flights contained within the mission is identified. For each flight within the overall mission an aircraft is allocated, the landing time inserted into the event queue and the counter recording the number of flights launched on time incremented. Once all flights within the mission are dealt with the mission event is ended. If insufficient aircraft are available the mission is placed into another queue which contains all those missions which could not be launched on time. After this each time an aircraft becomes available for tasking the first mission in the delayed mission queue is checked and the latest launch time for the mission compared to the current simulation clock time. If the simulation clock time is less than the latest acceptable take off time the process described above takes place with the aircraft allocated and landing times inserted into the event list. There are 2 counters for late launches which cover success in the first or second half of the acceptable delay period respectively and the relevant one is increased by the number of flights launched. If the simulation clock time is greater than the latest acceptable take off time for the mission then the mission is removed from the delayed mission list and the cancelled flights counter increased by the number of flights associated with it. In order to minimise the number of cancelled flights, ALSSim has been designed such that the delayed missions have priority over new missions for aircraft allocation and the queue is sorted to ensure that the earlier the planned launch time the closer the mission is to the front of the queue.

The Aircraft Entity

Unlike the mission entities, the aircraft entities have permanence and exist throughout the simulation. Each aircraft can, for the purposes of the simulation, be considered to be a collection of LRI positions which either have a LRI fitted or not. Each fitted LRI can either be serviceable or unserviceable as appropriate. The state diagram for the aircraft entity is at figure 5.

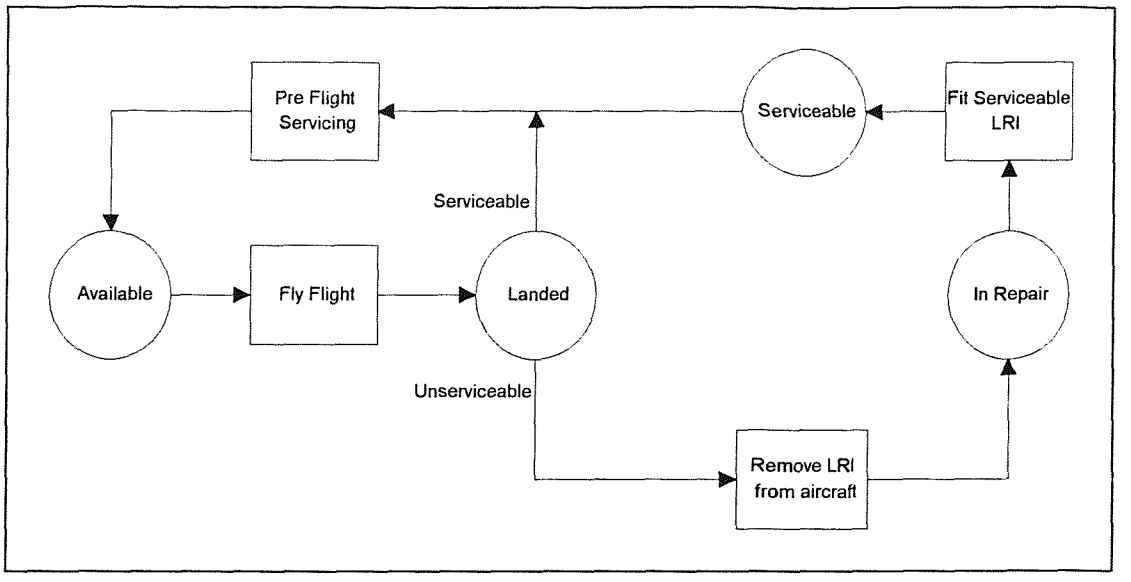


Figure 5. The Aircraft Entity.

The aircraft entities are created at the start of the simulation and have their states set to available for tasking at the start of each run. The aircraft is allocated to the first available flight and once it has landed will be examined for serviceability. If one or more LRIs are unserviceable it will have all of these unserviceable LRIs replaced to restore it to a serviceable state. If all LRIs are serviceable, or the activity necessary to restore it to a serviceable state has been completed, the aircraft will have a pre flight servicing. A pre flight servicing is a relatively low level activity which comprises of a refuel and some basic oil and fluid level checks and, if necessary, replenishments. Once the pre flight servicing is completed the aircraft state is changed to available at which point it will then be available for further tasking.

The Line Replaceable Item Entity

Although the LRIs within the simulation have been treated differently from the other entities in that they are not treated as a separate entity but are incorporated within other simulation entities, each LRI does follow a logical path between the various states and repair sites and could have been individually tracked if the simulation required that degree of granularity. Thus, although, each individual LRI is not instantiated as an unique object within the simulation it is necessary to understand the various states that are being represented within the application. The state diagram which describes the activities which occur for each LRI is at figure 6.

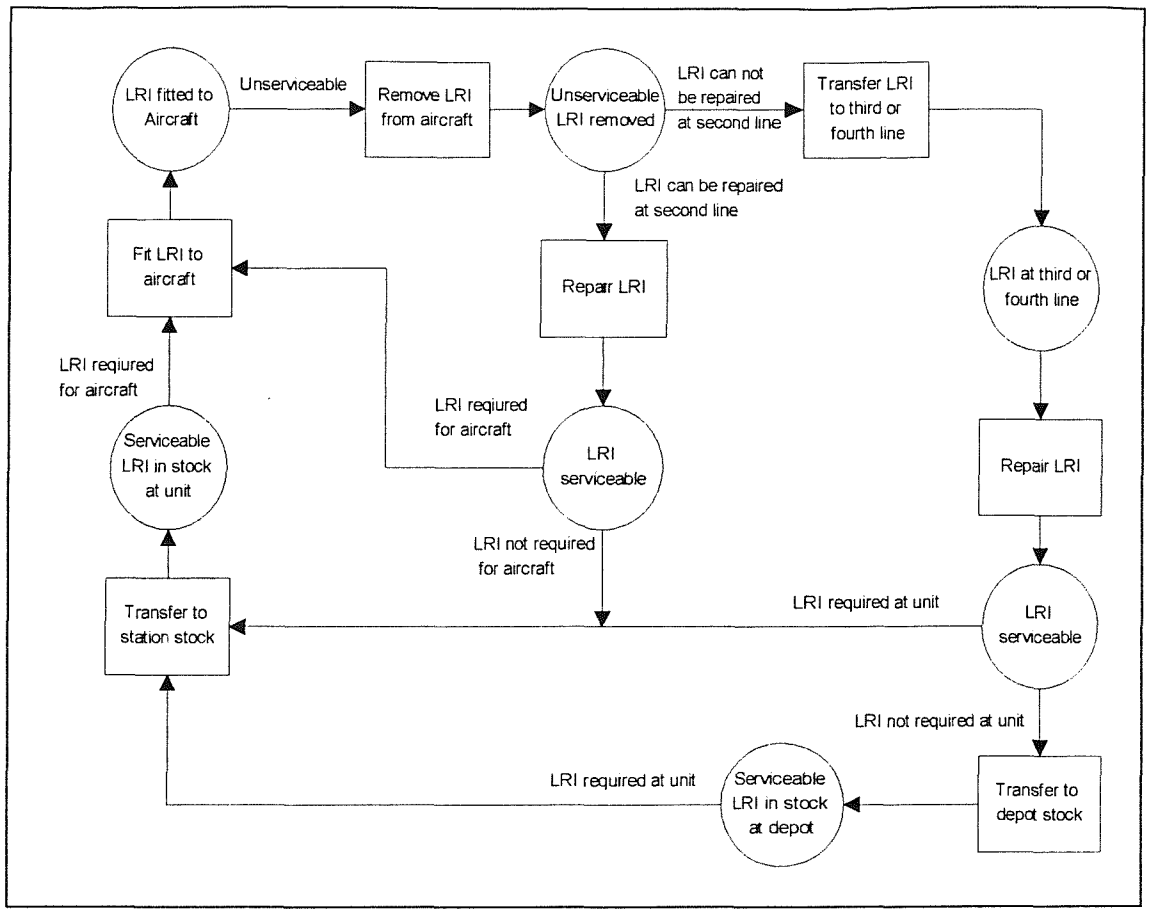


Figure 6. The LRI Entity.

The repair cycle starts when a LRI fails on an aircraft. The failed LRI is removed and, if available, a serviceable replacement from stock fitted to return the aircraft to a serviceable status. If the repair is within the capability of the unit the removed LRI is returned to the second line facility on the base for repair before being fitted to an aircraft if there is an aircraft that requires that component or placed within unit stock if not required for an aircraft. If the repair is beyond the capability of the second line bay the LRI is transported to the third or fourth line repair facility as appropriate. Once the repair is complete at this deeper repair location the repaired LRI is either dispatched to the unit if it is required for an aircraft or to depot stock if not. Within ALSSim all repairs are dealt with in terms of elapsed time and the manpower involved in undertaking repairs at the various maintenance levels are not modelled.

Simulation Language or Computer Language

The decision as to whether to produce the simulation using a standardised simulation language or to by means of programming using a computer language such as FORTRAN or C++ lies with the analyst. The former has the advantage that it has been optimised to meet the requirements of a simulation application and contains a large number of building blocks which when assembled will produce a simulation application to meet the needs of the analyst. However, it may result in an application with a number of redundant features within it slowing down the overall execution. The use of a pure programming language allows the analyst to design the application such that it only uses those features which are needed resulting in taut code and a faster application. Its drawback is that it requires extensive knowledge of the language to make best use of it, moreover, the need to programme the application line by line increases the development time of the simulation.

Recent developments to the Microsoft C++ computer language has meant that there is now a range of class libraries available to undertake such tasks as linked list production and management. This simplifies the task for the programmer in that it allows the development of the application as a hybrid between the historical computer languages which required the entire application to be built from first principles and simulation languages which were a combination of a series of building blocks linked together. The ALSSim simulation used to support this study was, therefore, written in Microsoft C++ version 5.0 achieving the dual aim of simplicity of code and speed of execution. The ALSSim application code is detailed at Appendix 1.

The ALSSim Simulation Application

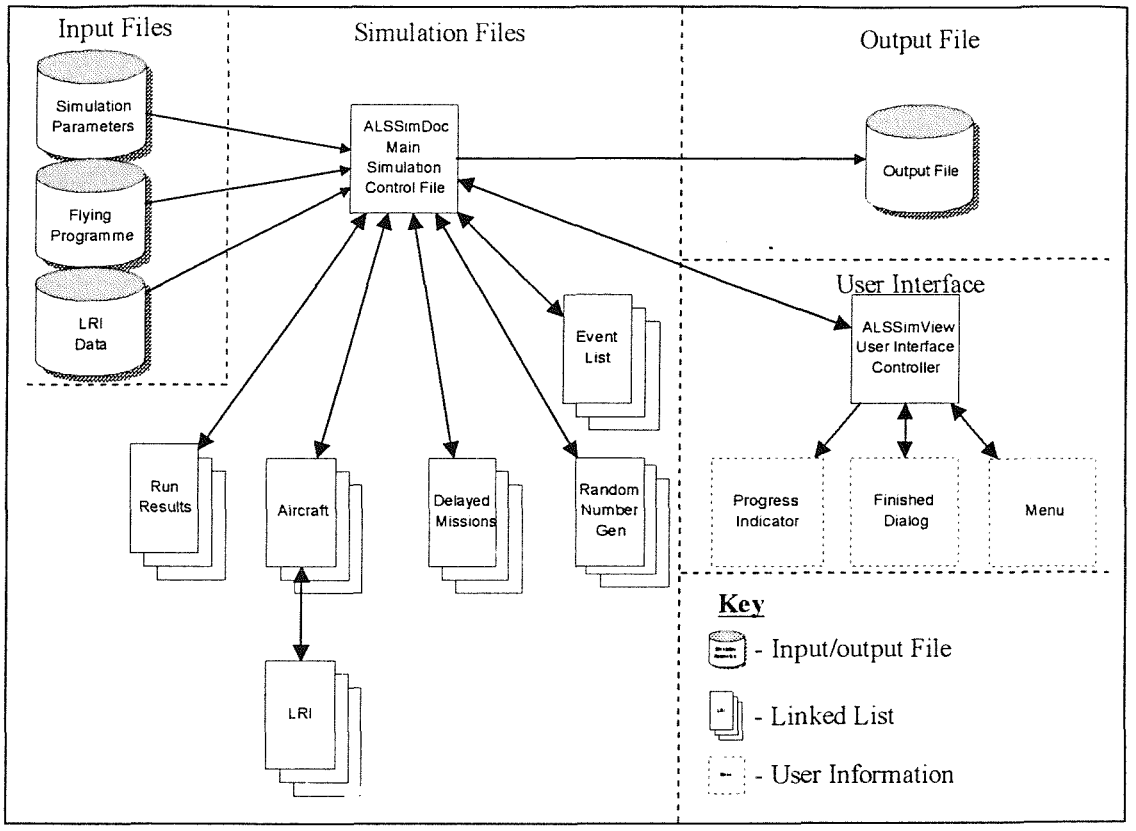


Figure 7. The Simulation Files.

As can be seen in figure 7 there are several of elements within the simulation that interact with each other. The application can be considered to comprise of 4 separate elements; the input files, the simulation files, the output file and the user interface. There is also an additional file created by Visual C++ which acts as an overarching control for the rest of the application. However, as this file is created and used by the operating system and not modified in any way whilst producing this simulation, it has been excluded from the figure. The 3 input files and the output file are external to the actual simulation and have been covered earlier in this treatise and will, therefore, not be covered in depth at this stage.

It is a convention within Microsoft Visual C++ that the file that manages the storage and flow of data within the application is described as a document file and in the case of ALSSim this file is named ALSSimDoc. As well as providing the overall control of the simulation this file also contains the majority of the simulation code. The other files within the simulation contain the data and code specific to particular entities by means of linked lists.

The final element of the simulation relates to the interface with the application user. As with the document file Visual C++ convention describes this as a View file and in the case of ALSSim it is named ALSSimView. Underpinning this file are 3 separate user interfaces which are used at different times within the simulation run. The menu comprises of both menu choices and buttons and provides the facility for the user to enter the names of the input and output files and having done so to initiate a run of the simulation. The run option is greyed out until all file names are entered thus ensuring that the simulation cannot be started inadvertently before all necessary information is available. The progress indicator is a simple bar which gives an indication of progress whilst the simulation is actually running. This indicator serves 2 purposes. Firstly, as it grows at the end of each run it gives confidence that the simulation has not stopped and secondly, it gives an visual indication of what proportion of the runs have been completed. The finished dialogue box gives an indication of the achievements and gives the user the choice between exiting the application or running another option.

Data

Clearly, before any simulation of a system can be carried out there is a need to collect data which will then be used within the simulation. In some cases there will already be a plentiful supply of data and the analyst need only select the appropriate data from that available in order to meet the requirements. Often, however, there is little or no available data and it is necessary to identify what is needed in order to produce the solution to the question being considered. Thus it is essential not only to know what data could be collected but also to have a clear understanding of the structure of the model and thus the data required to solve the problem. For this reason, the identification of the data to be collected is not the first stage of the process but should only be undertaken once the bounds of the problem have been defined, the entities that are contained within it identified and the state changes that will take place to those entities derived.

Since the 1970s, The RAF has been collecting failure and repair data for all its aircraft. This is achieved by the technician undertaking a task completing a Maintenance Work Order (MWO) on which is detailed the task undertaken and the time taken to complete that task. Where this task involves the a component's removal from or refit to an aircraft, the MWO also contains the aircraft flying hours allowing the tracking of the time that a

particular component has been fitted to the aircraft and thus the failure time for that component to be calculated. This data is collected at every RAF unit and forwarded to a central point to be stored on the RAF's Maintenance Data Computer (MDC). For each LRI there are 4 parameters that are required for use within the models that can be calculated from the data stored on this computer, the Mean Time Between Failure (MTBF), the mean time for removal, the Mean Time To Repair (MTTR) and the mean time for refit. This cycle will repeat many times for an LRI and is shown in figure 8 below.

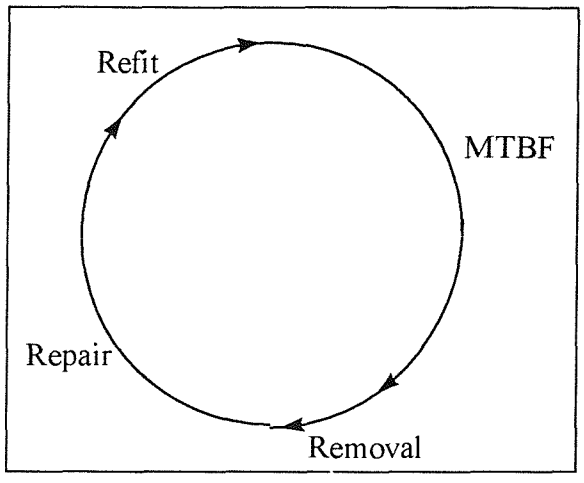


Figure 8. The LRI Life Cycle.

The calculation of the times for each of the 4 parameters for each LRI is achieved by taking a snapshot of the total data and using this to calculate the values which will be used within the models. MTBF for each LRI type is derived by calculating the mean time that each LRI is fitted to an aircraft before it fails in use. The other 3 are identified by examination of the time taken to undertake the work as recorded on the MWO by the tradesman. Although this method of data collection and analysis is simple and ensures that the recent data is available for use within the scaling models, it does suffer from the introduction of small errors which will affect the results of the modelling. Some distortion of the MTBF occurs because the time recorded on the MDC as the LRI failure time may not be that at which the failure actually occurred but rather the time of the landing of the aircraft post mission. Thus, a component that was fitted to an aircraft on a mission of 1 ½ hours duration which failed on take off would have a recorded failure time 1 ½ hours later than actually occurred. However, as most LRIs have MTBFs of thousands of hours this is unlikely to have much impact on the simulation results. Similarly, the quality of the data obtained from the information entered by the tradesmen on the MWO depends on the tradesman accurately entering the total time taken to carry out the task. Thus, there are

likely to be some small errors inherent in this data although, the magnitude of the repair time is such that minor errors in terms of a few minutes is not considered to be significant. This problem is compounded by the fact that fourth line does not provide repair time data and, therefore, the RAF uses elapsed time to model the time taken to repair at fourth line. Notwithstanding these problems with the data it is considered that the values obtained are within an acceptable degree of acceptability for use within the models in use today within the RAF.

There is one more issue related to the examination of data that relates to the MDC and the production of the values to be used within the models. Although the hardware and software are not the original many of the routines contained within it are old machine code instructions which are by their very nature difficult to read and modify. At the time that the routines for the production of the MTBF, MTTR and removal and refit values were written, the use of modelling within the RAF was in its infancy and it was considered that a single value for each was acceptable. Moreover, aircraft were not designed with maintenance in mind and little was known within the RAF about failure modes of equipment. It was recognised that mechanical components would eventually fail as a result of wearing out and it was the practice to set the maintenance policy for this type of component such that the component would be removed from the aircraft before this wear out occurred. In the case of electronic components it was believed that wear out was not a factor and their maintenance policies tended to be set up with replacement on condition. That is that the LRI would only be removed if it had failed. Even in the mid 1980s teaching within the RAF's training schools took little account of reliability and personnel were taught that failures occurred in accordance with the "bathtub" curve shown at figure 9. It was believed that after the first 2 to 3 years of an aircraft's operation that it and the LRIs contained within it were in the steady state portion of the curve and that, therefore, all LRIs could be considered to be operating in constant hazard.

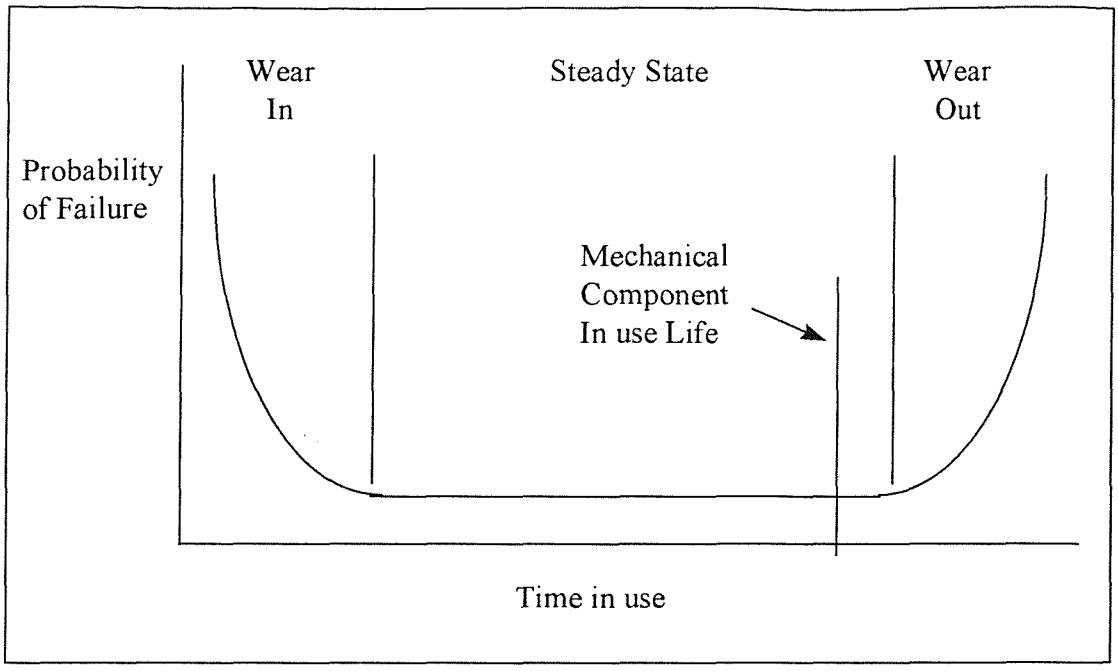


Figure 9. The Bathtub Curve.

It is now recognised that this logic is flawed and that any given component could exhibit a number of failure modes throughout its total life giving rise to a complicated distribution with a profile that may well change as the LRI ages. Moreover, the time taken to repair the defective components is subject to variability and is also likely to have a correlation with the failure mode. Therefore, repair time should also have a distribution that could be applied to the model to provide a more realistic output. However, it would be extremely costly to rewrite the code contained within the MDC at this late stage of its life to produce this level of functionality. The RAF is presently in the process of introducing a new Logistics Information Technology System (LITS) in conjunction with IBM which will include software routines to produce the underpinning data necessary to allow the derivation of distributions for both failure and repair parameters. It will also provide a mechanism by which better information about fourth line repair data can be collected. The inclusion of these improvements into the scaling models in the future will in turn lead to an improvement in the quality of the output of the models.

Pseudo Random Number Generation

In order to effectively operate a stochastic model there must be some degree of randomness within the simulation. In order to achieve this there should be some means by which numbers from a uniform distribution U(0,1) can be produced for use within the simulation. However, true random numbers are not only difficult to produce but are also

likely to be of little value if the requirement is to replicate the simulation to examine an number of alternative configurations in order to determine the best solution. There is, therefore, a need to provide a balance between the need for an randomness for each number produced and the requirement to replicate the sequence of numbers each time the simulation is run. One means of doing this is to apply a sequence of numbers from a table of genuine random numbers such as that produced by the Rand corporation in 1955 [5]. The main drawback with using tables is the need to store all the numbers that will be used within the simulation on the computer. More recently, in 1996 a CD-ROM containing 4.8 billion random bits was produced by Marsaglia [6] which overcomes the need to store the numbers within the computers memory but suffers in that accessing a CD-ROM is relatively slow. An alternative, would be to derive an algorithm that could be used within a digital computer which, whilst it would not produce truly random numbers, produces numbers of sufficiently acceptable randomness that they can be considered to be pseudo random. Moreover, the number produced by the algorithm depends totally on the original seed used within the algorithm ensuring that it is an easy matter to replicate the values used allowing the same sequence of numbers to be used to examine each alternative option. Law and Kelton [7] put forward the following 4 properties that a good pseudo random number generator must possess:

> *1. Above all, the numbers produced should appear to be distributed uniformly on [0,1] and should not exhibit any correlation with each other; otherwise, the simulation's results may be completely invalid.*
>
> *2. From a practical standpoint, we would naturally like the generator to be fast and avoid the need for a lot of storage.*
>
> *3. We would like to be able to reproduce a given stream of random numbers exactly, for two reasons. First, this can sometimes make debugging or verification of the computer easier. More important, we might want to use identical random numbers in simulating different systems in order to obtain a more precise comparison...*
>
> *4. There should be provision in the generator for producing several separate "streams" of random numbers...*

A practical means of producing pseudo random numbers which does meet all the above requirements is to use a Linear Congruential Generator along the lines of that originally proposed by Lehmer in 1951 [4] [5] [6]. This type of generator takes the form

$$n_{i+1} = (an_i + c)(\mathrm{mod}\, m) \quad \text{for } n \geq 0 \qquad (1)$$

where n is the pseudo random integer produced and a, c and m are fixed integer constants. For this generator to work all of the factors must be non-negative integers as must the seed value n_0. The simulation is likely to use a large number of pseudo random numbers and, therefore, for the generator to be of use it must have a long cycle length. It has been shown that in order to achieve a full cycle of numbers the factors within the generator must meet the following conditions [7]:

1. *There must be no integer other than 1 that is an integer divisor of both c and m.*
2. *If a number p is a prime number that is an exact divisor of m, then p must also be an exact divisor of (a-1).*
3. *If m is divisible by 4 then (a-1) must also be divisible by 4.*

It is not difficult to devise values that can be used within a Linear Congruential Generator that meet the requirements given above and as a result it was used for a number of years as the basis for most pseudo random number generators supporting simulation modelling. However, it is somewhat simplistic in that each new number is only calculated using the previous value. A better algorithm to use [8] makes use of a generator which, whilst still meeting the requirements detailed above is recursive in nature and is of the form

$$x_n = (a_1 x_{n-1} + \ldots + a_k x_{n-k}) \ mod\ m \qquad (2)$$

where the order k and modulus m are positive integers and the coefficients $a_1, \ldots, a_k$ are integers in the range -(m - 1), ... , m - 1.

As computer processing speed gets faster so the complexity of the random number generator can increase without there being a marked increase in the processing speed of the simulation. In their paper [9] L'Ecuyer and Andres proposed a generator that

provided uniform random number generation by means of a combination of four linear congruential generators. The paper also includes C code for the proposed generator and it this code that is used within ALSSim to provide the pseudo random numbers that are used to support the simulation.

Although it is acceptable to use all the numbers from a single stream with each event as it occurs picking up the next number in the sequence, this is not a good way to control a simulation. When undertaking a study with a number of alternative options to be considered, it is necessary to be sure that any given event occurs at the same time for all the alternative options being examined otherwise there is a risk that the results for some options are distorted by the simulation events happening at different times. The effect of this type of approach is explained in detail by Pidd [14]. This problem is easily solved by using several streams of numbers with each stream specific to a particular event within the simulation. However, it is important to ensure that the starting seed for each stream is sufficiently far apart from the others to minimise the risk of reusing the same numbers and thus the streams may start at n_1, n_{100001},n_{200001}...etc thus reducing the risk of replication. Of course there is always a risk of streams overlapping if a run uses a large number of random numbers in which case the same number will be used more than once.

The ALSSim simulation makes use of 5 pseudo random number streams with separate streams used for LRI failure time, time for LRI repair, selection of whether the repair occurs at the unit or the depot, no fault found in the LRI at the unit repair facility and no fault found in the LRI at the depot. In each case the starting stream seeds are separated by 100,000 numbers virtually eliminating the likelihood of repeated use of any given numbers. Similarly, each run has a different set of starting seeds to eliminate unintentional distortion of the final results caused by reusing the same numbers for more than one run.

Testing of Pseudo Random Number Generators

The pseudo random number generator used within the simulation will produce a series of numbers which lie within the range $0 \leq n \leq m - 1$. If the generator is valid it will produce a uniform distribution of values U(0,m-1) and we convert those numbers to a U(0,1) uniform distribution before using within the simulation. It would, nonetheless, be unwise to accept at face value that the particular combination of generator, seed value and constants produce a valid and hence acceptable uniform distribution without undertaking

some tests to confirm this. The testing of generators is covered in depth in the literature [15][16][17][18], however, there is no requirement to carry out all the available tests in order to prove the acceptability of the generator used as no pseudo random generator is capable of producing true random numbers and all that is required is an acceptable degree of randomness. Thus a range of tests which are not closely related are selected giving a sufficient breadth of examination for an acceptable level of confidence that the generator is sufficiently random to meet the needs of the simulation. These range from simple to apply tests such as those testing for uniformity, through scatter graph examinations to tests examining length of runs and poker tests. In each case the results are examined for Chi-square uniformity and the overall results examined. The failure of any one test is not be sufficient to dismiss the generator as the aim is to obtain an overall assessment not a series of yes/no criteria all of which must be met.

Although the use of the values contained within the design of the generator proposed by L'Ecuyer and Andres [19] gives a degree of confidence that the generator within ALSSim will produce valid numbers for use within the simulation, it would be unwise to accept them at face value. Moreover, only by undertaking testing is it possible to be sure that the particular implementation within ALSSim provides a valid generator producing acceptable pseudo random numbers. In testing ALSSim a total of 50,000 random numbers split into 10 equal sized groups were produced and the following tests applied to them: Stagger Chart plots, Frequency Test, Poker test, Gap test, Distribution of Pairs test, Frequency of Pairs test and Runs test. The results of these tests for the ALSSim pseudo random number generator is shown at Appendix 2.

Input Distribution Selection

Having selected a good pseudo random number generator it is necessary to address the selection of the input distribution for events. Dependent on the nature of the problem it may not be practical to collect large quantities of data or storage considerations within the simulation application or its host computer may preclude the storage of large numbers of empirical values. Clearly, the selection of an appropriate distribution is important if the correct deductions are to be drawn from the results obtained. Where the data is available it is possible to examine it in order to ascertain the best correlation between it and the various distributions which can be used. However, when data is either scarce or of insufficient fidelity for this analysis to be undertaken it is necessary to consider the various

options available before choosing what is considered to be the most appropriate distribution for the aspect of the problem being examined. Fortunately, considerable work has gone on in the past to identify appropriate distributions that can be used in the simulation dependent on what the input is. A particularly good coverage of this is covered by Law and Kelton [20] which also covers the various factors to be considered.

In the case of ALSSim, for the reasons explained earlier, it was not possible to derive appropriate distributions for MTBF and MTTR from the data available. Therefore, it was necessary to identify appropriate distributions by experimentation and to prevent the interaction between multiple distributions affecting this analysis it was split into 2 distinct phases. The first phase fixed the repair time and examined alternative distributions which could be used to model the LRI MTBFs. Once a distribution had been selected it was used to model MTBF and a similar process was applied to identify the distribution that would be used to model LRI MTTR. The following candidate distributions were examined in both phases: Exponential, Lognormal, Normal, Triangular and Weibull. In each case the particular equations used to generate the particular distributions within the simulations were as described by Cheng [21]. For each distribution a series of simulations were run against the same flying programme, spares failure and repair information and spares scale with the number of aircraft available incremented each simulation to give the number of flights flown for aircraft availability numbers from 1 to 36.

Derivation of MTBF Distribution

Before going on to derive the actual distribution to be used to model MTBFs it is appropriate to define the baselines against which the particular distribution's results will be judged. In terms of the number of flights flown it is known from examination of the flying programme file that the maximum possible is 1920 flights. This provides the first comparative measure. The second involves comparing the flights flown against those that would be achieved if the failure and repair times were both fixed to the mean values. The third is a subjective measure of examining the variance. If we recall that the flying programme being used has been designed to be met by an overall availability of 36 aircraft we can reasonably expect the variance to reduce as the number of aircraft available approaches 36.

Fixed Failure - Fixed Repair

The first runs of the simulation were undertaken with both the failure and repair times fixed to the values read in from the LRI data input file. This set of runs provided a set of values which could then be used to examine the results of the runs for the various failure distributions. For this case the failure time (X)for any given LRI with a mean failure time (μ) is:

$$X = \mu \tag{3}$$

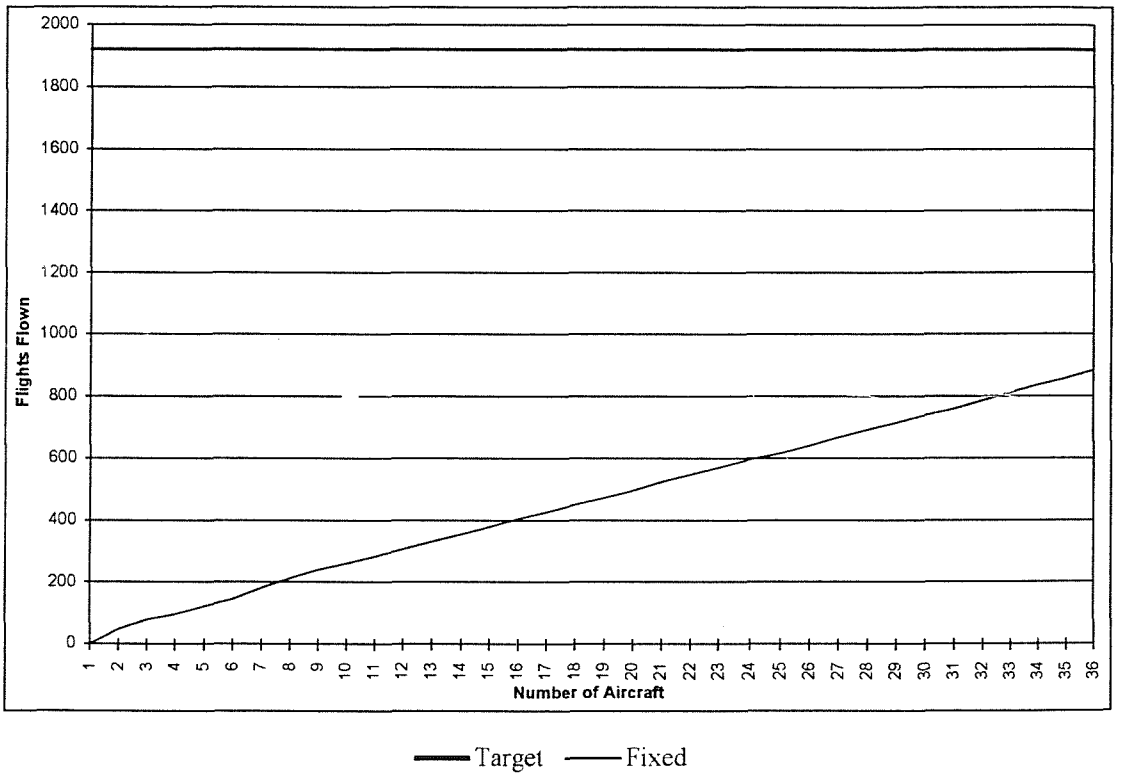


Figure 10. Flights Flown for Fixed Failure and Repair Time.

As can be clearly seen by examination of figure 10 the use of fixed failure and repair times gives a poor result with a final achievement of less than 50% of the tasked flights flown. This is not perhaps what would be expected, however, the results are valid and can be explained by considering what is happening within the simulation. Where a fixed failure time is used, any given LRI fails at the same elapsed flying hours on every aircraft. In the case of some of the LRIs the fixed repair time is longer than the overall simulation time. Moreover, there are insufficient spares to allow all failures to be solved from stock. The combination of these 2 factors means that an aircraft suffering one of these failures never

becomes serviceable which results in a steady reduction in the number of aircraft available and hence a poor final achievement.

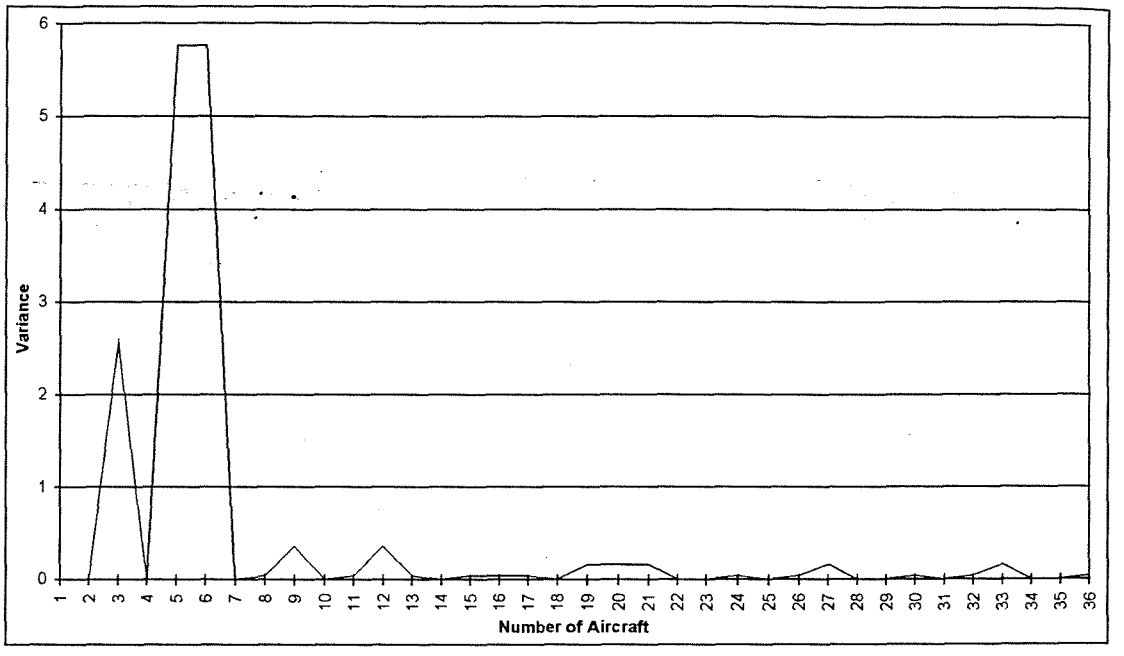


Figure 11. Variance for Fixed Failure and Repair Time.

Figure 11 shows the variance for the fixed failure case. Although the failure and repair times are fixed there is a small variance in this case which results from the use of a uniform distribution to model the likelihood that the component is found to have no fault on it when it arrives at the repair location. In this case, the LRI is returned to the shelf in a relatively short period of time. As can be seen by the very small values for variance this has little impact on the overall results of the simulation run.

Exponential Failure - Fixed Repair

One of the easiest distributions to model is the exponential in that the only factor that needs to be known is the mean MTBF(μ). Thus for an Exp(μ) where $\mu > 0$ the failure time for a particular LRI (X) calculated when fitted to a particular aircraft is:

$$U = RN(0,1)$$
$$X = -a\ln(1-U) \qquad (4)$$

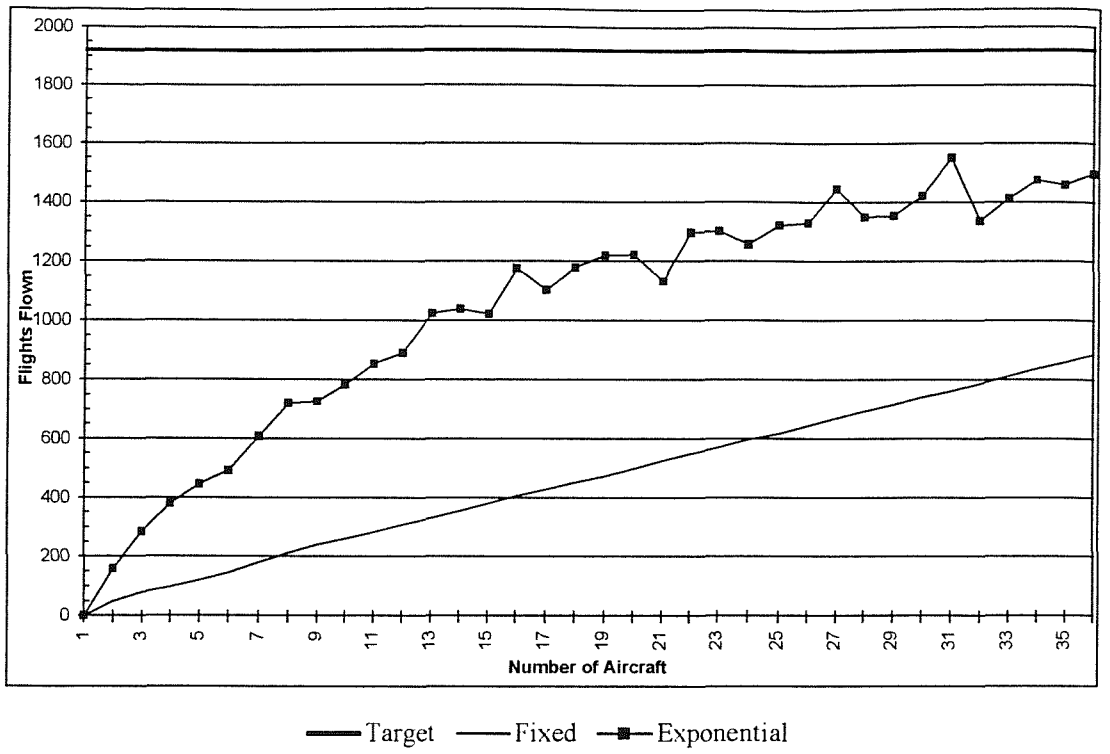


Figure 12. Flights Flown for an Exponential Failure Distribution.

Figure 12 shows that the use of an exponential distribution to model failure times results in a much improved flight success rate than the fixed failure time for all runs with a maximum achievement in the order of 78%. The initially high rate of increase in the flight achievement reduces to a lower rate from the 13 aircraft case onwards.

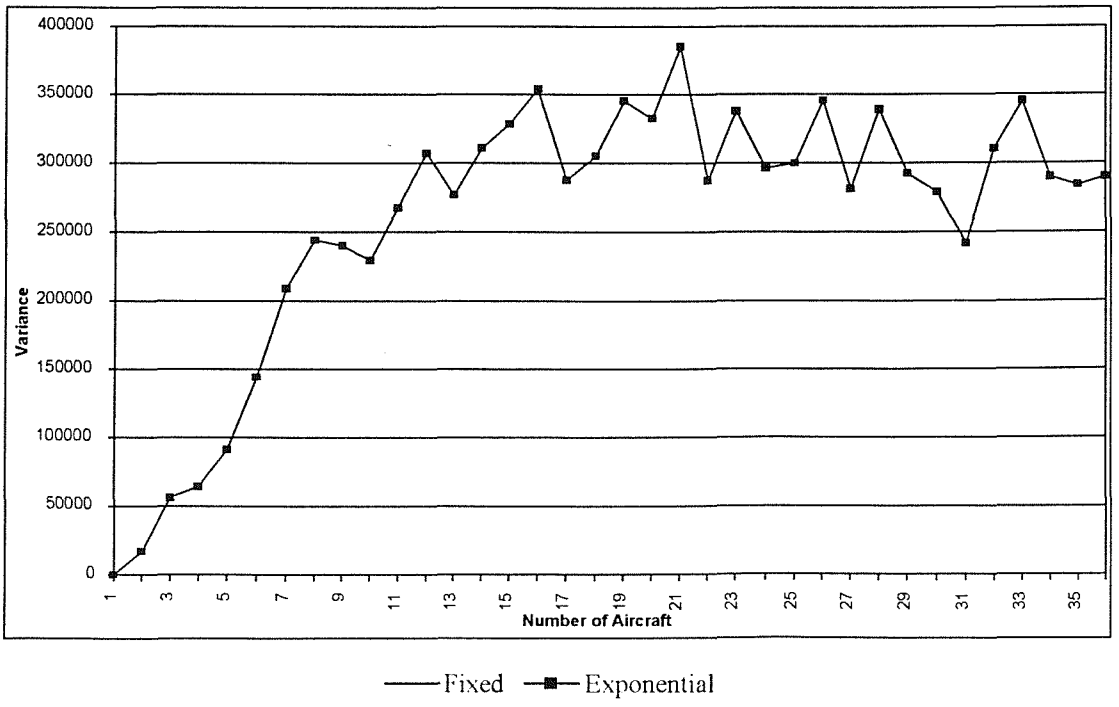


Figure 13. Variance for an Exponential Failure Distribution

Figure 13 shows the variance for the exponential distribution. In this case there is a rapid rise in variance until the 13 aircraft case after which the variance is relatively constant throughout the rest of the runs with most results falling within a band lying between 250,000 and 360,000.

Normal Failure - Fixed Repair

There are a number of different ways of returning a normal value for a distribution with a mean MTBF (μ) and a known variance (σ^2). In this case it has been decided to use the Polar version of the Box-Muller transform. Thus for a Normal (μ,σ^2) the failure time for a particular LRI (X) is:

$$U_1 = RN(0,1), U_2 = RN(0,1)$$
$$V_1 = 2U_1 - 1, V_2 = 2U_2 - 1$$
$$W = V_1^2 + V_2^2$$
$$If\,(W < 1)$$
$$\{$$
$$Y = \sqrt{\frac{-2LnW}{W}}$$
$$X_1 = \mu + \sigma V_1 Y \qquad (5)$$
$$X_2 = \mu + \sigma V_2 Y \qquad (6)$$
$$\{$$

In the case of ALSSim we only require one value to be returned so the second value of X is discarded each time the calculation is undertaken. The Normal distribution is an open distribution which returns values which lay in the range $-\infty \leq 0 \leq \infty$. However, as ALSSim is using this distribution to model failure times of equipment a negative value for failure time is neither wanted nor appropriate. Thus the above value for failure time (X_1) must return a value greater than 0 in order to be of use within the simulation. In order to achieve this the ALSSim Normal distribution is constrained such that all values lie within the range $0.1\mu \leq \mu \leq 1.9\mu$ thus ensuring that results are valid and that the results are not skewed by having one tail longer than the other. Bounding the possible values in this way does result in the elimination of very long times to failure but, as these are likely to be very rare, their elimination is not likely to have a significant impact on the overall result.

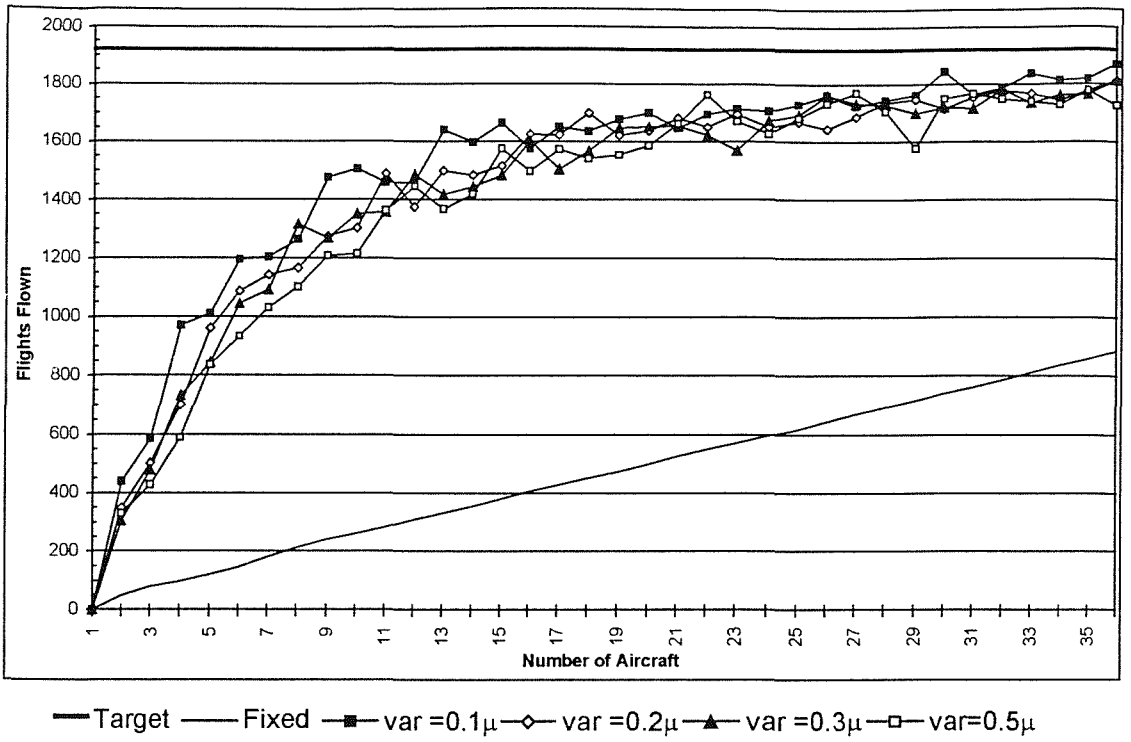


Figure 14. Flights Flown for the Normal Failure Distributions.

Figure 14 shows the results for a range of normal distributions where the mean values remained the same for all simulation runs but the variance was varied. As can be seen there is little difference in the results obtained for each alternative variance throughout the range of simulations and all show a marked improvement on the fixed case and a maximum flight achievement in the order of 95% of those tasked.

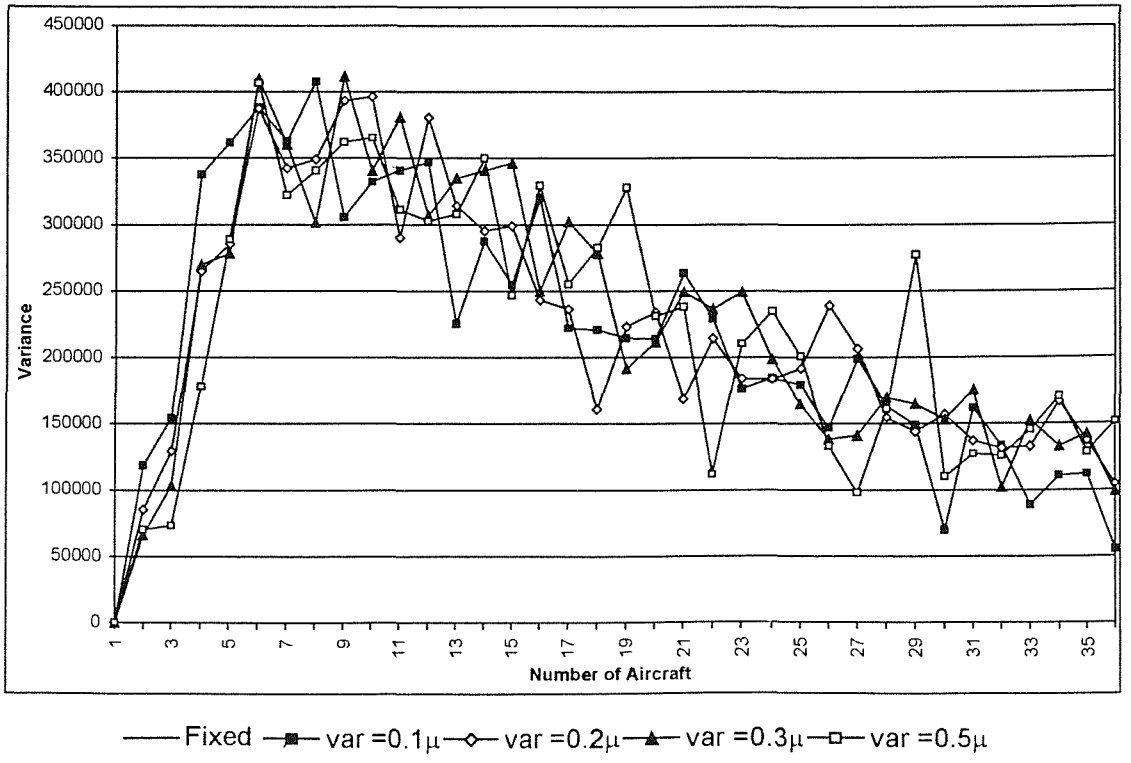


Figure 15. Variance for the Normal Failure Distributions.

Figure 15 shows the variance for the normal distribution runs and as with the flight achievement shows little difference for the various variances examined. It can be clearly seen that there is an initially steep increase in the variance where few aircraft are available where the requirement for lengthy repairs to an individual aircraft followed by a relatively steady decline as the number of aircraft available increases. Whilst there is little to choose between the various alternative options the line for a variance of 0.1μ shows the closest approximation to the mean value and is, therefore, taken forward for consideration as a candidate distribution.

Lognormal Failure - Fixed Repair

In order to find the failure time for a LRI (X) for a LRI subject to Lognormal distribution for failure with a mean MTBF (μ) and a known variance (σ^2) we must first find the normal return and then calculate e raised to the power of that value. Thus for a Lognormal (μ,σ^2) the failure time for a particular LRI (X) calculated when fitted to a particular aircraft is:

$$Y = N(\mu, \sigma^2)$$
$$X = e^Y \qquad (7)$$

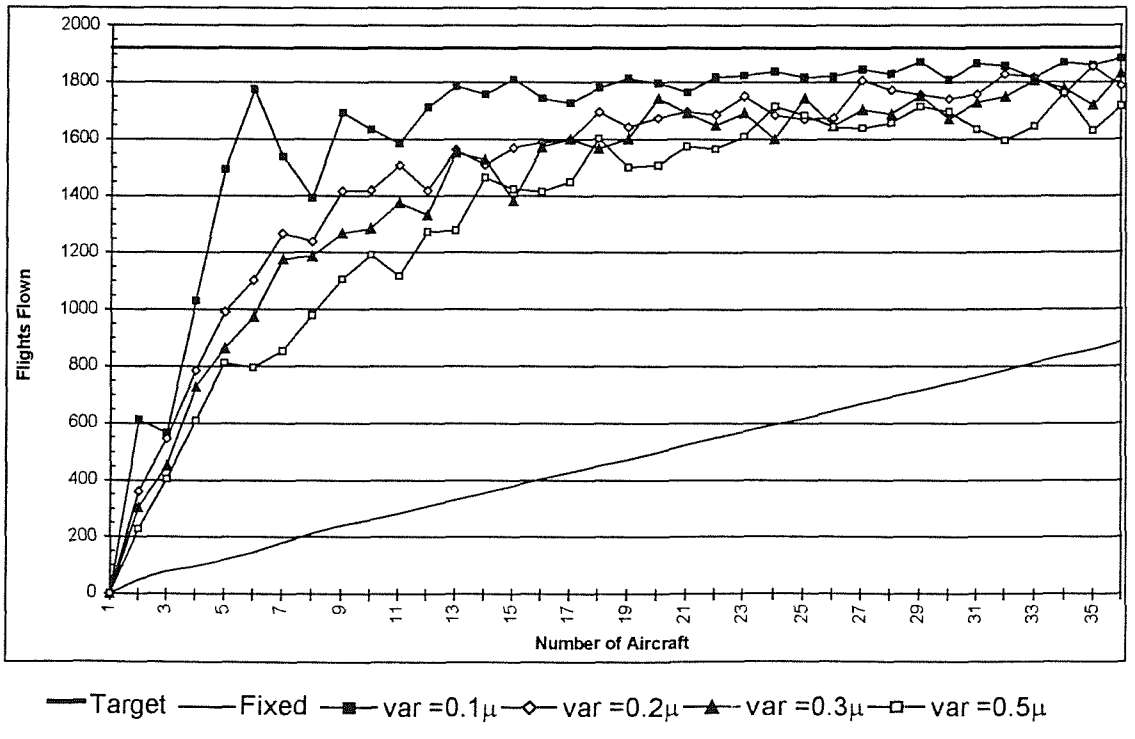


Figure 16. Flights Flown for the Lognormal Failure Distributions.

Figure 16 shows the results for a range of Lognormal distributions with a constant mean for each LRI failure but a variable variance. As with the previous options examined the results are considerably better than that for the fixed failure time resulting in a maximum flight achievement in the order of 95%.

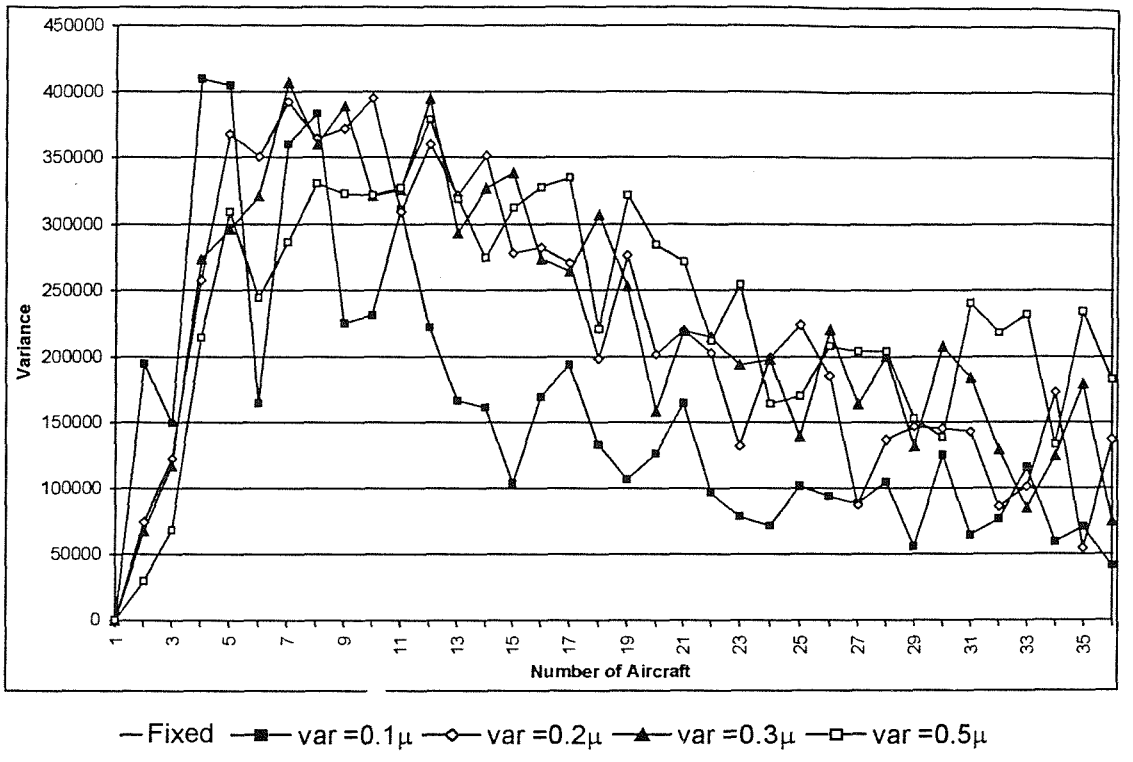


Figure 17. Variance for the Lognormal Failure Distributions.

Figure 17 shows that, whilst the results for the runs for the Lognormal distribution has the same sort of shape as that of the Normal distribution model, the Lognormal model exhibits considerably greater variability in the variance results. However, it can be seen above that for most values of available aircraft the variance = 0.1μ results in a lower variance value and a higher overall flight achievement. Thus the Lognormal (μ,0.1μ) distribution will be considered further as a candidate distribution for the failure distribution.

Triangular Failure - Fixed Repair

The triangular distribution is a simple distribution that is often used when little is known about the data distribution. For this distribution 3 parameters are required: the minimum (a), the mode (b), and the maximum (c). The input file for ALSSim gives values for the minimum, maximum and a divisor (d). The mode is calculated by:

$$b = a + \frac{c - a}{d} \qquad (8)$$

The failure time (X_t) for a Triangular(a, b, c) $a < b < c$ is given by the formula:

$$\beta = \frac{(b-a)}{(c-a)}$$
$$U = RN(0.1)$$
$$If\,(U < \beta)$$
$$\{$$
$$T = \sqrt{\beta U}$$
$$\}$$
$$Else$$
$$\{$$
$$T = 1 - \sqrt{(1-\beta)(1-U}$$
$$\}$$
$$X_t = a + (c-a)T \qquad (9)$$

The mean (μ_t) for a Triangular(a,b,c) is given by

$$\mu_t = \frac{a+b+c}{3} \qquad (10)$$

Whilst this gives a value for failure for the Triangular(a,b,c) distribution it must be related back to the LRI mean before it can be used within the simulation. Thus the true failure time (X) for a LRI with MTBF (μ) is:

$$X = \frac{X_t \mu}{\mu_t} \qquad (11)$$

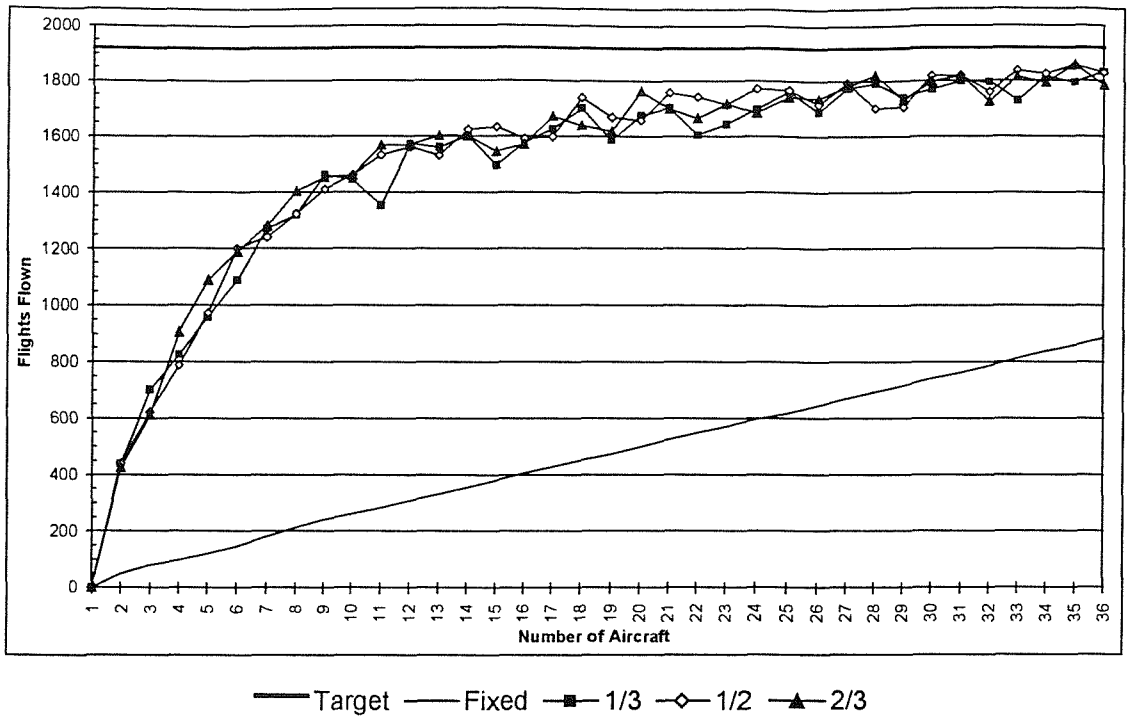


Figure 18. Flights Flown for the Triangular Failure Distributions.

Figure 18 shows the number of flights flown for a series of Triangular distributions with a minimum of 0.1μ, a maximum of 1.9μ and a range of divisors. It can be observed that the shape of the various Triangular distributions being considered has little impact on the results achieved throughout the range of available aircraft. The maximum achievement for the Triangular distribution is approximately 93%.

Figure 19 below reveals a similar generic shape to the variance results for the Lognormal distribution already considered albeit there is greater variability in the actual values for the various alternative Triangular distributions. None of the 3 options considered stand out as a better model than the other 2 and whilst the divisor of 1/3 was chosen as the candidate choice for further evaluation either of the other 2 could just as easily have been selected.

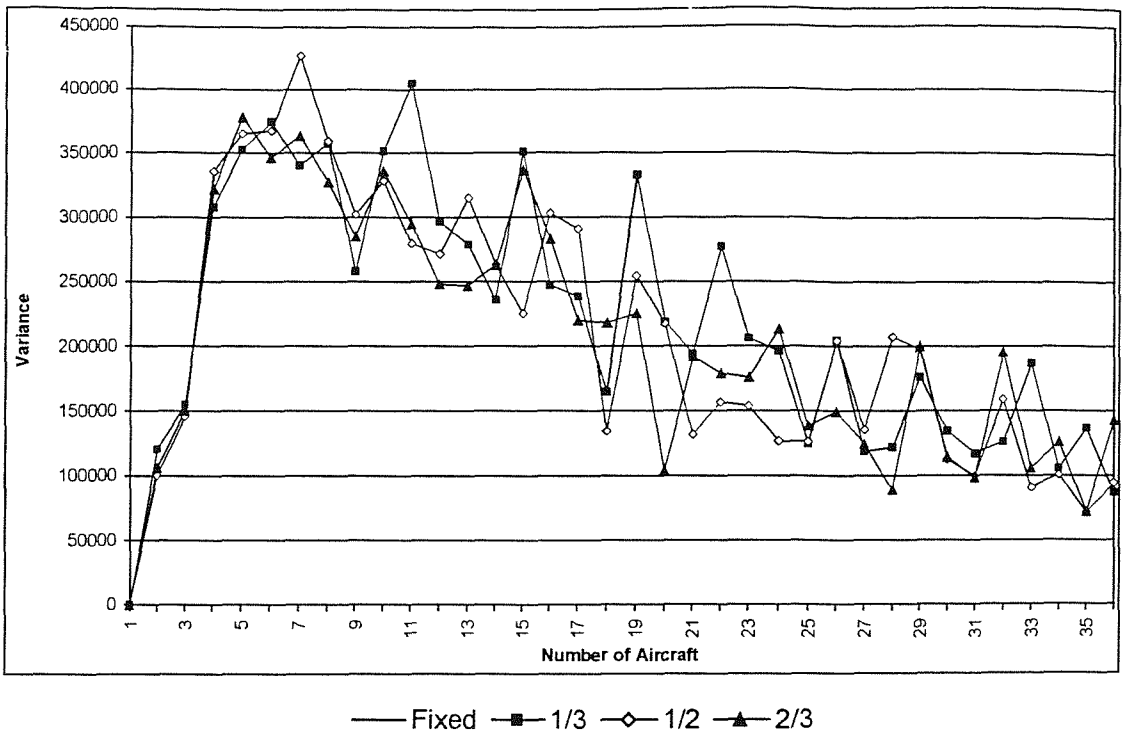


Figure 19. Variance for the Triangular Failure Distributions.

Weibull Failure - Fixed Repair

The final distribution family considered as possible models for the failure distribution is the Weibull. The Weibull distribution requires 2 parameters, the scale parameter (a) and the shape parameter (b). The scale parameter is based on the LRI MTBF (μ) and is given by

$$a = \frac{\mu}{\Gamma\left(1+\frac{1}{b}\right)} \tag{12}$$

Thus it can be seen that the scale parameter for each individual LRI is different and that, therefore, the failure time for a particular LRI (X) subject to a Weibull(a,b) distribution is given by:

$$\begin{aligned} U &= RN(0,1) \\ X &= a^b \sqrt{-\ln(1-U)} \end{aligned} \tag{13}$$

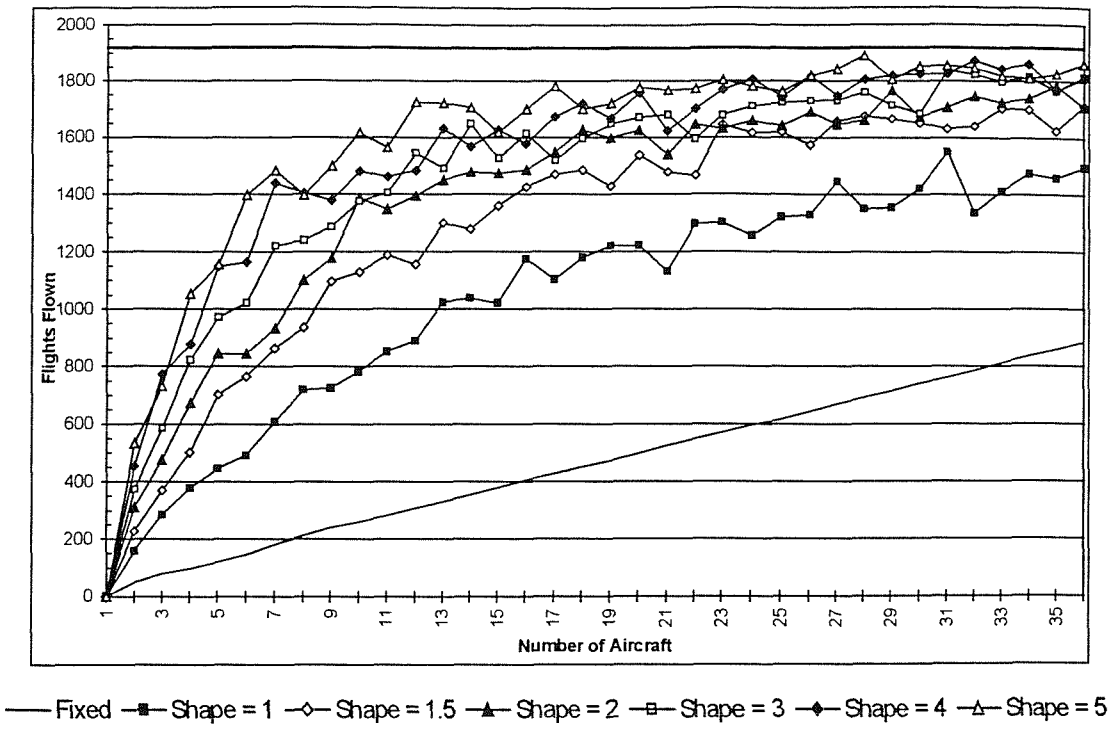


Figure 20. Flights Flown for the Weibull Failure Distributions.

Figure 20 shows the results of the runs using various shape parameters for the Weibull distribution. Worthy of note is the set of results for the shape parameter = 1. When this is the case the Weibull distribution is the same as the exponential distribution. Examination of the results for b = 1 and the exponential model reveals exactly the same results as for the exponential distribution discussed earlier. In terms of the number of flights achieved, the number of flights achieved at the higher end of the availability range is very similar and gives a maximum of approximately 93%. As the number of aircraft available is reduced there is a wider spread for shape parameters less than 3.

Figure 21 below gives the variance results for the various Weibull runs and again the runs for a shape parameter of 1 replicates the results of the exponential runs. In terms of the choice of candidate distribution we should select a distribution with a shape parameter which gives a good result and a low value for variance. Examination of the Flights Achieved graph shows that shape parameters of 4 and 5 gives very similar, but better than the other values, and examination of the Variance graph reveals that the shape parameter of 4 gives a slightly less variable variance than that for 5. Therefore, the Weibull distribution Weibull(a,4) is used as a candidate distribution for the failure distribution.

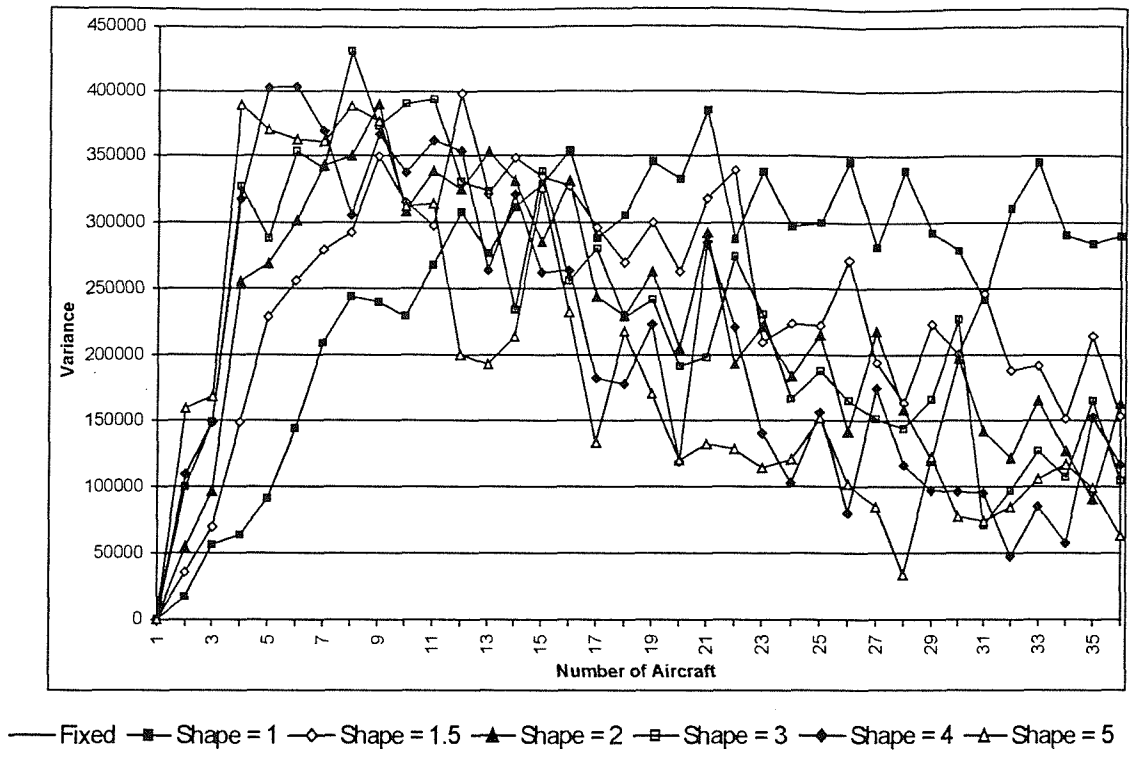


Figure 21. Variance for the Weibull Failure Distributions.

Having examined a range of alternative distributions to identify which should be considered as candidates for the LRI failure distribution, it is now appropriate to examine them against one another to identify which is the most appropriate.

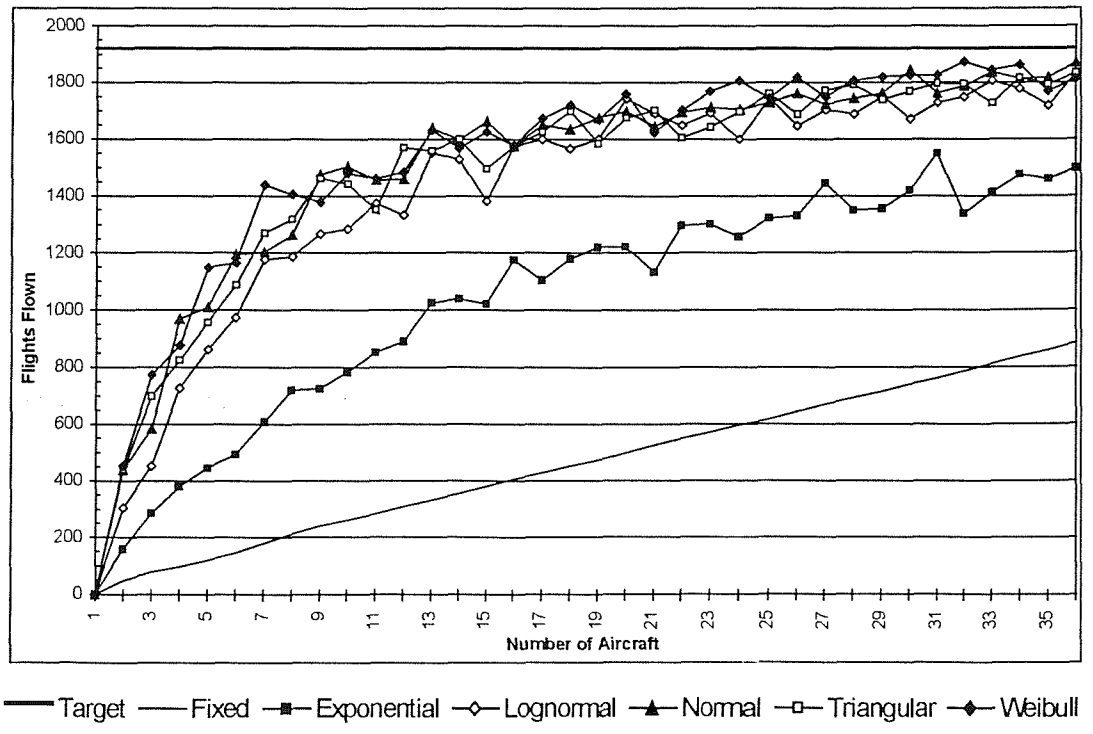


Figure 22. Flights Flown for the Alternative Failure Distributions.

Figure 22 shows graphically the number of flights achieved for each distribution being considered. In order to eliminate distributions as not appropriate, they were evaluated against 2 criteria with those that failed either excluded from further consideration. Firstly, it was known that the target achievement for 36 aircraft was 1920 flights and that, therefore, any distribution that did not give a result within a reasonable range of that value was excluded. Secondly, as the scaling was based on the requirements for 36 aircraft to achieve the mission task it was unlikely that there were sufficient spares to allow the task to be met with far fewer aircraft and, therefore, any distribution that met the task with far fewer aircraft would be excluded. As can be clearly seen both the fixed failure time and the Exponential distribution fail the first of the criteria with achievements of approximately 50% and 78% of the target flight achievement respectively. All of the other distributions give similar results throughout the range of simulations and thus the Normal, Lognormal, Triangular and Weibull distributions need to be considered further to decide which is most appropriate.

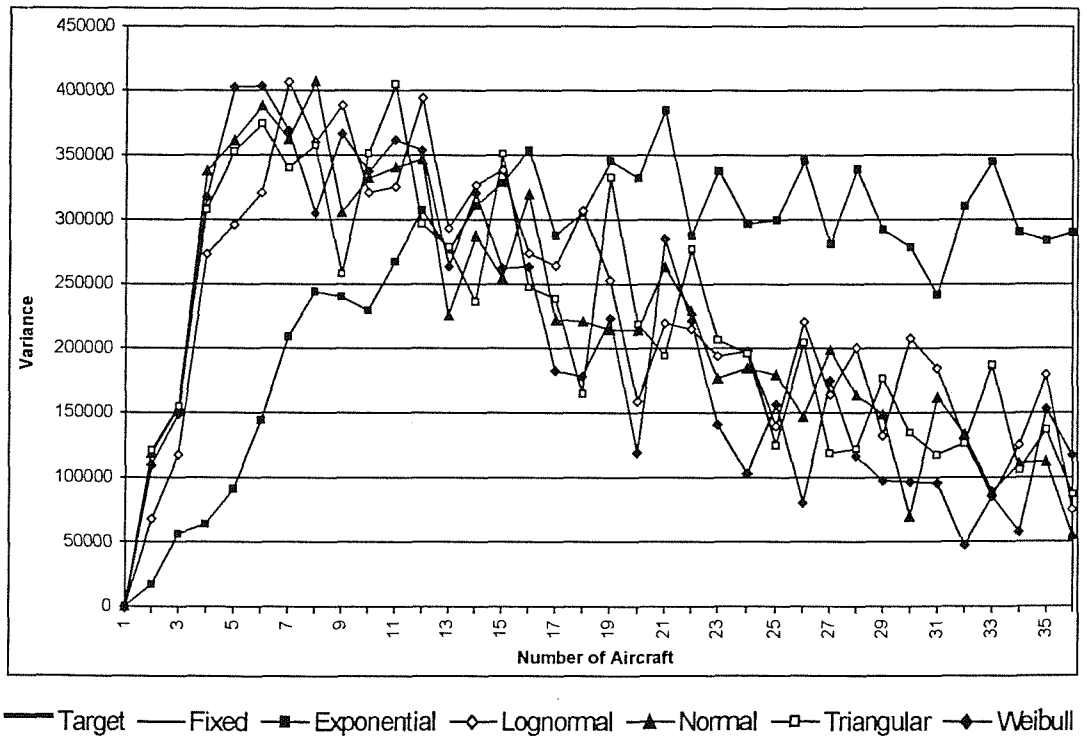


Figure 23. Variance for the Alternative Failure Distributions.

Figure 23 shows the variance for the distributions which generated the results in figure 22. Examination of variance traces for the distributions reveals that each shows a similar trend with an initial rise in variance, as the number of aircraft available builds so variability in the random numbers starts to play a role, followed by a continual decline as the number of

aircraft available increases thus reducing the variability between individual runs. There is little to choose between the various alternatives although, the Weibull distribution generally achieves a slightly higher flight achievement and has lower values for variance. Therefore, it was decided to use the Weibull distribution with a shape parameter of 4 to model LRI failures.

Derivation of the Distribution to be used as the LRI Failure Distribution

Having decided that Weibull is an appropriate distribution to use to represent LRI failure, a view that is supported in [22], a similar process was carried out as described above to determine the most appropriate distribution to use to model variability in the repair times. In this case the Weibull (μ,4) distribution was used to model failure times with repair times modelled using the Exponential, Lognormal, Normal, Triangular and Weibull distributions. For simplicity the same values were used for each distribution as had been selected for final consideration for the failure distribution. Figure 24 shows graphically the results of the various distributions.

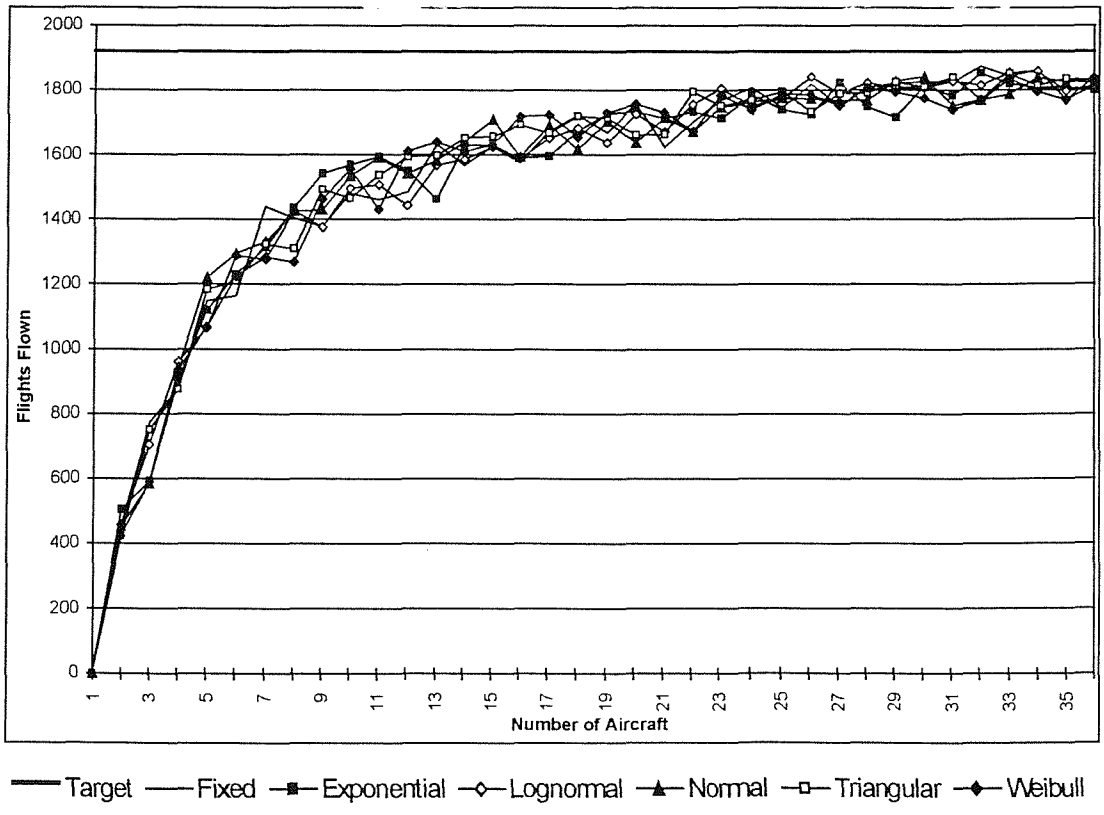


Figure 24. Flights Flown for the Alternative Repair Distributions.

Examination of figure 24 shows that there is little difference in the results of the various distributions and that, therefore, it would appear that any of the distributions would

provide an acceptable option for modelling repair time. This graph presents insufficient evidence to make a positive choice and, in order to do so, the variance for the runs, shown at figure 25 needs to be examined.

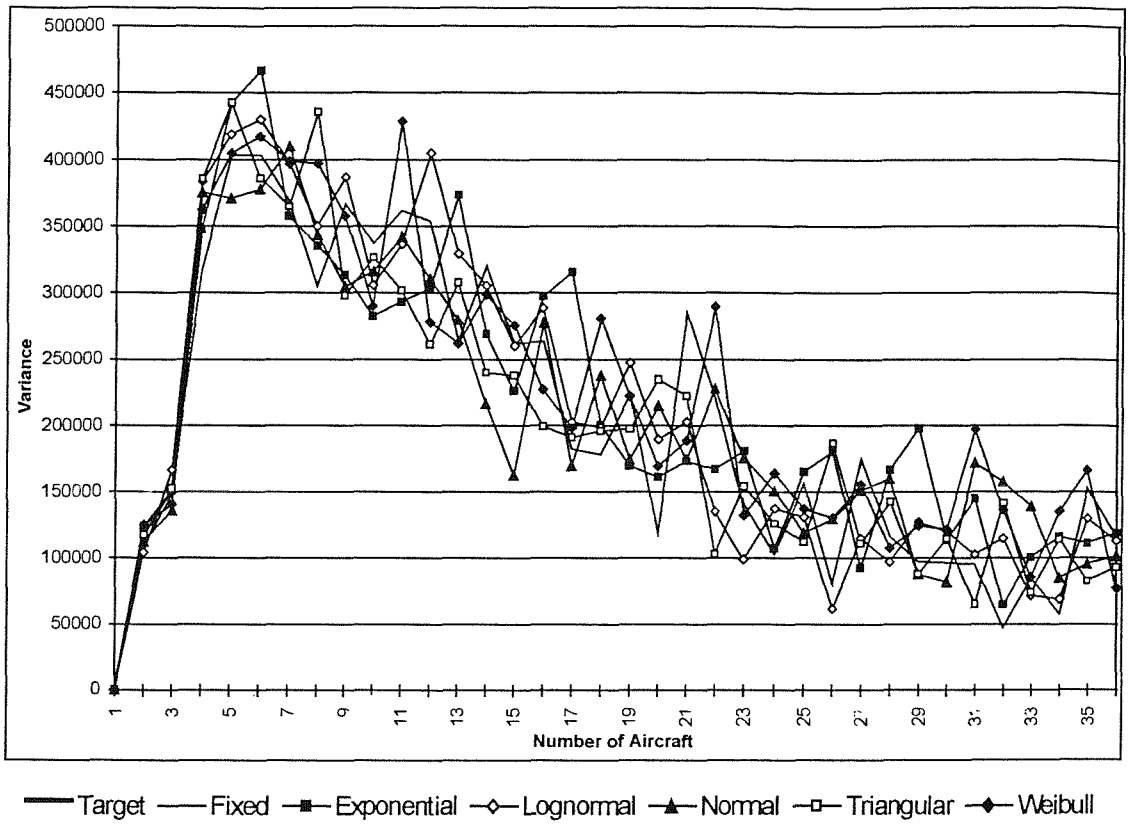


Figure 25. Variance for the Alternative Repair Distributions.

Examination of figure 25 reveals that for once again there is no distribution that stands out clearly as the most appropriate to model repair times. In order to make that decision it is appropriate to undertake closer examination of the variance for the runs undertaken with the higher numbers of aircraft available. This is achieved by figure 26, below, which focuses in on the variance results for aircraft availability numbers from 20 to 36. Examination shows that all of the alternative distributions have a degree of variation in the results. However, the Lognormal distribution has fewer large peaks and troughs than the others leading to the conclusion, confirmed in [23], that a Lognormal distribution is an appropriate distribution to use for LRI repair times.

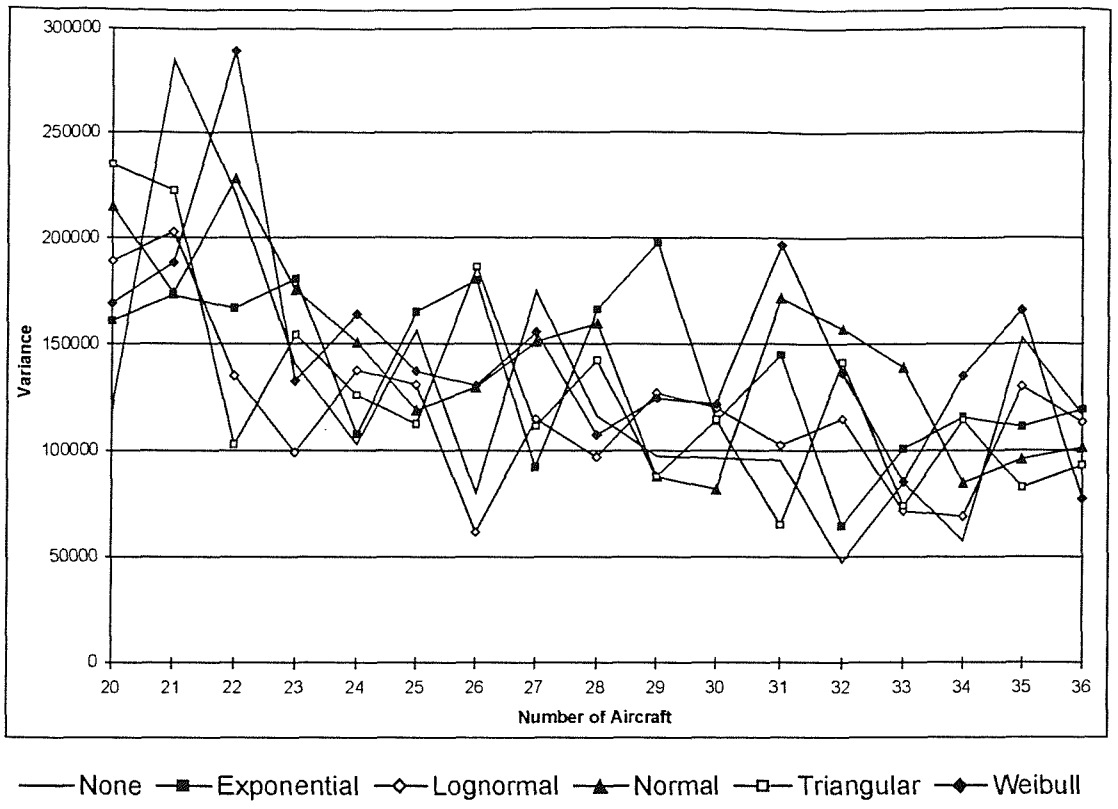


Figure 26. Variance for the Alternative Repair Distributions for Upper Range of Aircraft Available Runs.

Thus the ALSSim simulation application makes use of 3 distributions to introduce variability, a Weibull distribution for failure time, a uniform distribution to select the repair location and a lognormal distribution for the LRI repair times.

Results

As has already been stated the ALSSim simulation produces a single output file which contains all the results within a single series of runs. This file is produced as a text file which can then be read into a range of word processing and spreadsheet packages for further analysis off line. Whilst it was possible to include graphical functionality within the simulation this would have required extensive additional design to provide a feature that is already available within all spreadsheet packages. Moreover, by providing the raw figures rather than a graph it is a relatively simple matter to present the results in a slightly different fashion thus permitting different views of the overall information available.

The actual file can be considered to have 5 different sets of results contained within it. The first contains the basic information necessary to repeat the simulation if required. It details the input and output file names, the number of runs per simulation, the number of

aircraft available, the duration of the pre flight servicing, the maximum delay permitted and the distributions used within the simulation and their parameters. The second section, shown at table 1, deals with the mean mission statistics and the resultant variance.

	Total	Variance	Percentage
Tasked Flights	1920.00		
Flights Achieved	1889.68	6740.18	98.42
On Time	1886.40	8093.41	98.25
First Half Flight Delay Maximum	2.30	51.34	0.12
Second Half Flight Delay Maximum	0.98	10.70	0.05
Cancelled Flights	30.32	6740.18	1.58
In Flight Aborts	121.78	20210.40	6.34
Launched Flights Succeeded	1735.78	37352.80	91.86
Launched Flights Failed	69.72	8114.08	3.69

Table 1. Mission Achievement Means.

The third section gives, for each aircraft, the proportion of the simulation time that it spent in the various alternative serviceability states in both terms of number of hours and percentage of the total simulation. An example of these results is given at table 2 for the first 5 aircraft.

Aircraft Number	Unserviceable	Awaiting Flight Servicing	In Flight Servicing	Serviceable	Flying
1	1248.31	0	478.05	1784.02	473.62
2	1261.69	0	475.43	1775.89	470.98
3	1191.01	0	321.05	2154.52	317.41
4	1154.50	0	291.96	2248.97	288.57
5	1161.82	0	277.18	2270.80	274.20

Table 2. Time Spent in Alternative States.

The fourth section gives, for each day of the simulation, the mean number of aircraft in each state as at the start of each day. Table 3 shows the results for the first 5 days of the simulation.

Day Number	Serviceable	Flying	Unserviceable	In Pre Flight Servicing	Awaiting Pre Flight Servicing
1	36.00	0	0	0	0
2	33.66	0	0.43	1.91	0
3	32.88	0	1.26	1.86	0
4	32.24	0	1.91	1.85	0
5	31.81	0	2.35	1.84	0

Table 3. Daily Aircraft States.

The fifth, and final, section of the results file gives the mission achievements for each day of the simulation. As before this is a mean and table 4 shows the results for the first 5 days of the simulation.

Day No	Tasked Flights	On Time	Delay		Cancel	Successful Take Offs	In Flight Abort	Successful Missions	Failed Missions
			Less Than 1/2 Max	More Than 1/2 Max					
1	16	16.00	0	0	0	16.00	0.79	14.30	0.47
2	16	16.00	0	0	0	16.00	0.90	13.78	0.60
3	16	15.78	0	0	0	15.78	0.70	13.87	0.37
4	16	15.52	0	0	0	15.52	0.60	13.41	0.34
5	16	15.92	0	0	0.7	15.92	1.10	13.11	0.75

Table 4. Daily Mission Achievements.

Simulation Verification and Validation

The final aspect to be considered within a general review of features of simulation is that of the verification and validation of the model. Having designed the simulation to represent the environment to be examined, it is necessary for the analyst to be satisfied that the interactions that take place within the simulation accurately represents those of the real world. The mistake that can be easily made by an inexperienced analyst is to wait until the application has been programmed before undertaking this activity. The correct approach is to validate each stage of the development with the customer to ensure that not only are the assumptions made within the model correct but also that the questions that require answering are addressed by the simulation. There are a number of techniques for doing this dependant on the design technique that has been used by the analyst ranging from the production of activity cycle diagrams for an activity approach to the simulation [24] to

state diagrams for an object oriented approach [25]. Only when the conceptual model has been validated and accepted as an accurate representation of the real world problem to be modelled should actual programming commence. Having completed the production of the simulation application the analyst needs to verify that the code written accurately models the agreed conceptual model and also that the results produced are correct. simulation models are by the nature of what they are complicated models which contain a number of subroutines each of which need to be checked for accuracy and their effect on the rest of the application. There are a number of techniques that can be used to achieve this function[26][27]. Verification is not a quick process but is essential if the customer is to have confidence in the output of the simulation. Therefore, it is necessary to undertake sufficient verification to prove the efficacy of the simulation whilst ensuring that no more of this process is undertaken than is necessary to achieve that aim. There are 2 main questions in this process that must be addressed. Firstly, does the simulation and its internal features accurately model the problem as articulated by the agreed system model. Secondly, are the outputs of the model correct. The first can best be answered by obtaining an event trace and manually working through it. This will ensure that the correct flow through the events are occurring and if compared with a manual work through of the problem that the various subroutines are producing the correct inputs to the event decision making process. The latter can be simply achieved by one of 2 methods. If there is a similar model in use the outputs of both can be compared to see if they are similar or the simulation could be used to model the system as it stands and a comparison made between the actual achievements and those predicted by the simulation.

The validation of ALSSim was undertaken as a 2 stage process. Firstly, before undertaking any programming, it was necessary to develop the underlying model and, having done so, to examine it against reality to ensure that the proposed model represented reality to a sufficient fidelity to be acceptable. This was accomplished by producing the activity cycle diagram at Appendix 3 and then discussing this proposed model with the RAF's logistic modelling staffs and practical aircraft engineers to determine whether it contained a sufficient level of detail to adequately represent the system being examined. Once that was agreed the simulation application could be designed and programmed. Once that was complete, the second stage of the validation process was undertaken by running the simulation and obtaining an event trace showing all events that occurred and the time at which each took place. Comparison of this trace manually against the model

already agreed confirmed that the programmed simulation matched the design model. Once the application was accepted as a valid representation of the real world it was then necessary to validate the output to ensure that the results produced are acceptable. For some years, the RAF has been using an simulation application which has been developed in-house using FORTRAN as the programming language. This application used a simple underlying model and suffered from memory constraints and a lack of speed, taking some hours to run each option. This compares with ALSSim which achieves the same run in minutes. The outputs for mission achievement for both were compared using test data and were found to produce similar results. Further verification was achieved by comparing the results of ALSSim with the deterministic model which revealed similar values for system availability. Having completed this process, and accepted that ALSSim was both valid and verified, it was ready for use.

THE USE OF ALSSIM AS A PROBLEM SOLVING TOOL

Baseline Problem

All that has gone before describes the theoretical aspects of the problem leading to the design and programming of a simulation application for use as an evaluation tool. The remainder of this treatise describes the practical use of the simulation and considers the results produced and the message to a user. Before using ALSSim as a tool to test the effect of various changes to the input parameters it is first necessary to provide a baseline against which the changes can be examined. This was achieved by running the following simulation problem:

> What is the effect on the number of flights flown of altering the number of aircraft available to fly a specified flying task of 8 missions of 2 aircraft each mission for 5 days each week over a 24 week period? No mission will launch unless 2 aircraft are available to fly and the current time is no later than one hour past the original planned launch time for the mission. The reliability of the LRIs and number of flight servicing teams remain constant over all alternative options. The LRI spares file is that produced using the deterministic models to provide an overall system availability of 80% over the simulation period.

The above problem is actually the set of parameters used to develop the simulation features that have already been covered earlier. Figures 27 and 28 respectively show the

number of flights achieved for each variation in the number of aircraft and the run variances for these results.

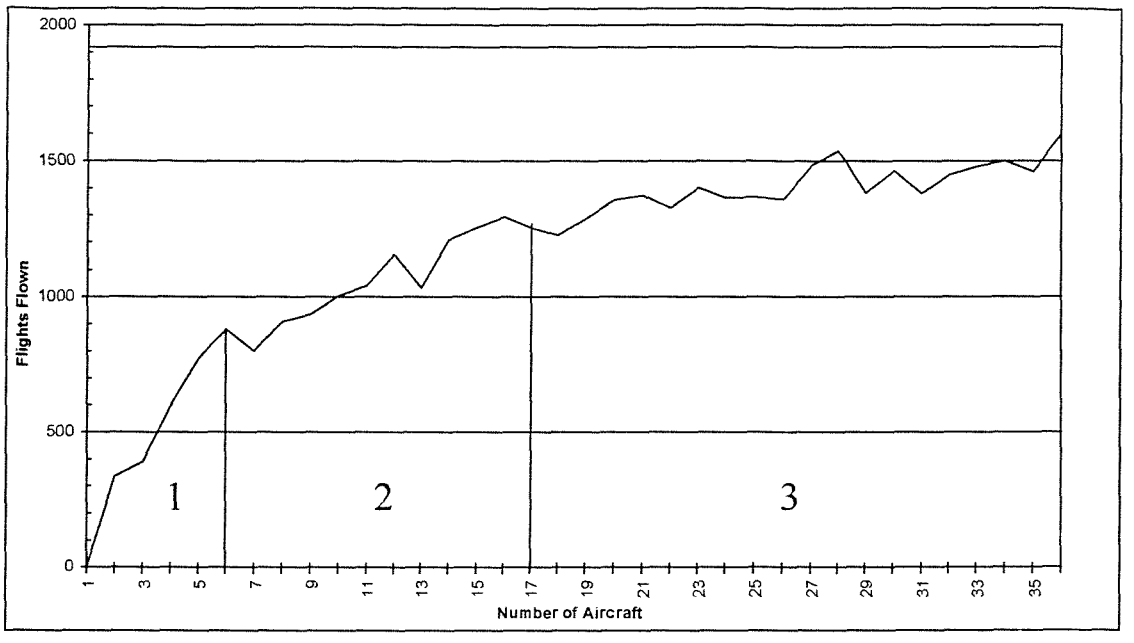


Figure 27. Flights Flown for the Baseline Option.

Initial Analysis of figure 27 reveals that there are 3 distinct regions. Region 1 shows a rapid increase in the number of flights flown which reflects the marked effect that each increase in the number of aircraft has on the number of flights that are successfully flown. Region 2, whilst continuing to exhibit an improving achievement has a reduced gradient reflecting a reduction in the impact of each increase in the number of aircraft available. In region 3 the gradient reduces further and reflects an approach to the asymptotic point where further increases in the number of aircraft available will nit improve overall result achieved. Figure 28 shows the variances for these runs and permits further examination of the 3 zones to see if there is any correlation between the observed changes in Figure 27 and the variances.

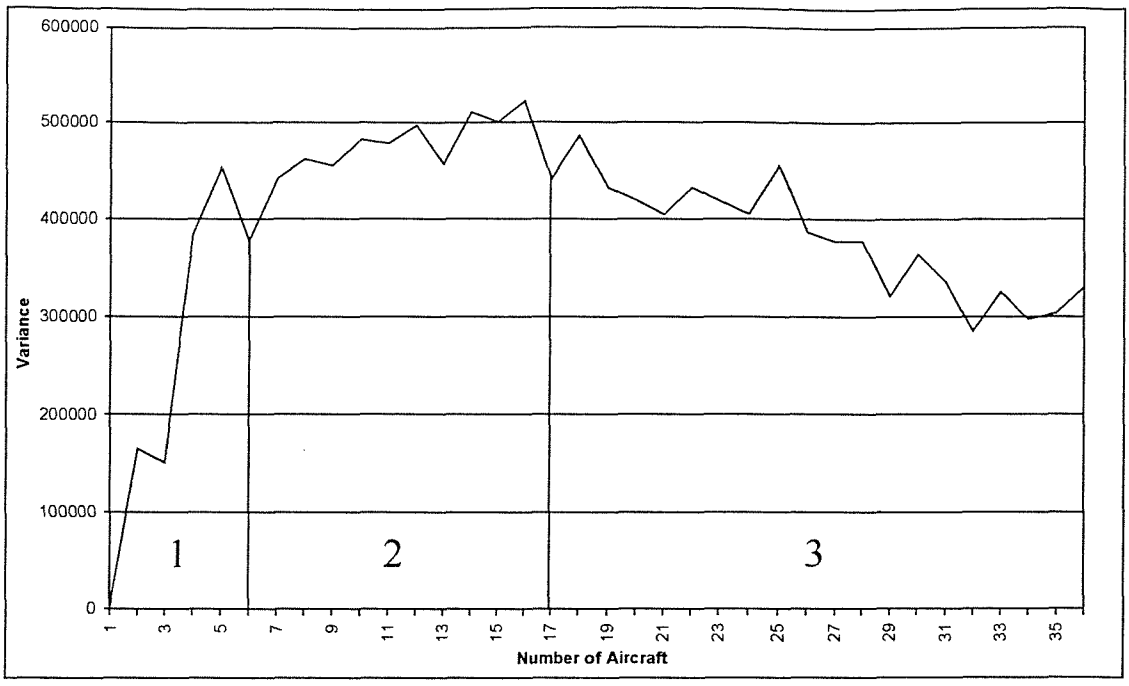


Figure 28. Variance for the Baseline Option.

Examination of the variance for the three regions described above shows that the results for region 1 show a rapidly increasing variance which reflects the wider variation that occurs as the number of aircraft is increased and the range of possible outcomes increases rapidly. Region 2 shows a reduction in the gradient but nonetheless reveals that the variance continues to increase over this range. decreasing variance as the number of aircraft increases to the point that there is sufficient availability to meet the requirement and individual unserviceabilities has a decreasing impact on the overall availability. Region 3 rapidly decreasing variance as the number of aircraft increases to the point that there is sufficient availability to meet the requirement and individual unserviceabilities has a decreasing impact on the overall availability.

Whilst the results of this baseline study reflect the expectation it is worthwhile examining the results in order to construct a confidence interval for the mean values achieved. This will give an indication of the degree of spread of acceptable outcomes and hence a feel for the acceptability of the final result. The formula to be used in calculating the confidence interval is [28]

$$\bar{X}(n) \pm t_{n-1,1-\frac{\alpha}{2}} \sqrt{\frac{S^2(n)}{n}} \qquad (14)$$

where $\overline{X}$ is the derived mean, $t_{n-1,1-\alpha/2}$ is the $1-\alpha/2$ critical point for a t distribution with n-1 degrees of freedom and is obtained from the table [29], α is the confidence interval, S^2 is the sample variance and n the number of runs. An explanation and proof of this is given in [30]. For the baseline case it was decided to use a confidence interval of 0.9 leading to the use of a $t_{n-1,1-\alpha/2} = 1.66$. The resultant upper and lower bounds for each result was calculated and is presented graphically at figure 29.

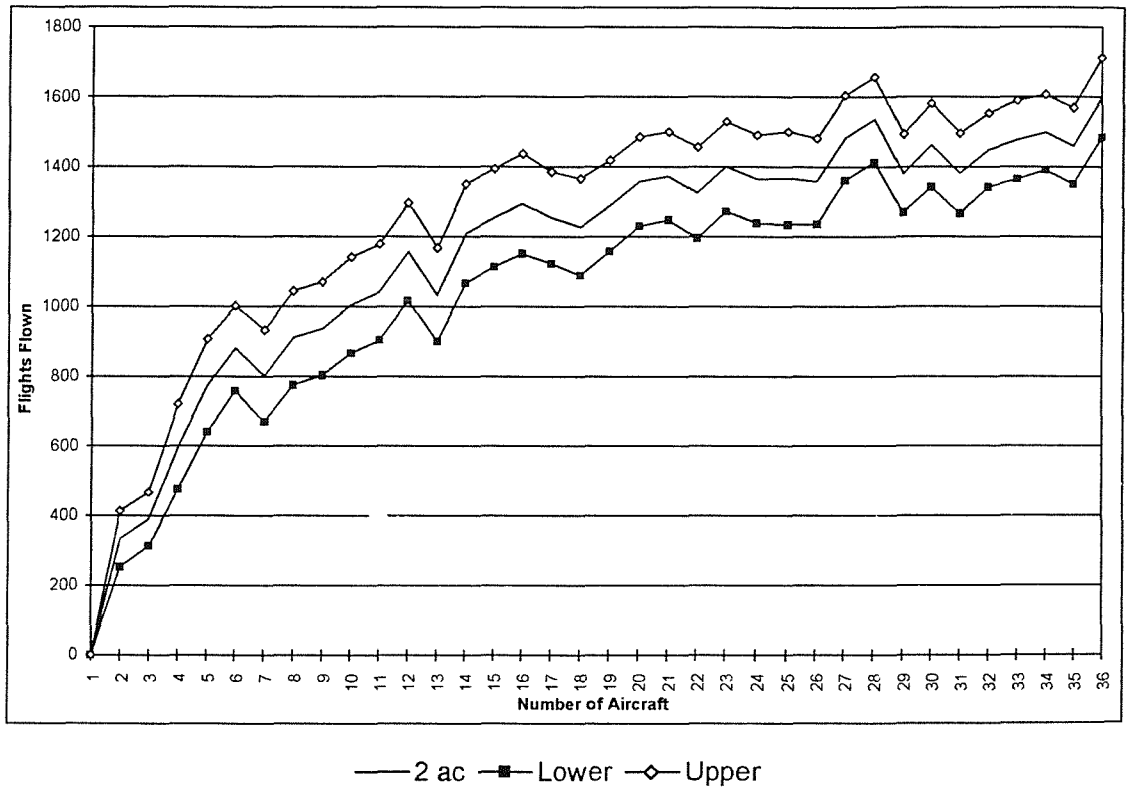


Figure 29. Flights Flown for the Baseline Option showing Upper and Lower Confidence Boundaries.

Figure 29 shows a close correlation between the result achieved and the upper and lower boundaries of the 90% confidence interval which reflects a simulation with limited variance in the various runs. This is particularly apparent in the high range of aircraft availability and further reinforces the earlier expressed views that the simulation application uses appropriate input distributions and models the environment correctly.

Variation in number of aircraft per mission with total number of flights remaining the same

This baseline case can now be used as a means to assess the effect of varying the various parameters to see their impact on the final result. The first variation to be considered is

that of varying the mission requirements in terms of the number of flights launched on each mission whilst keeping the total number of flights and thus the flying commitment constant. Figures 30 and 31 show the flight achievement and variance for 3 separate cases: the baseline 2 aircraft missions, 4 aircraft missions, and 1 aircraft missions.

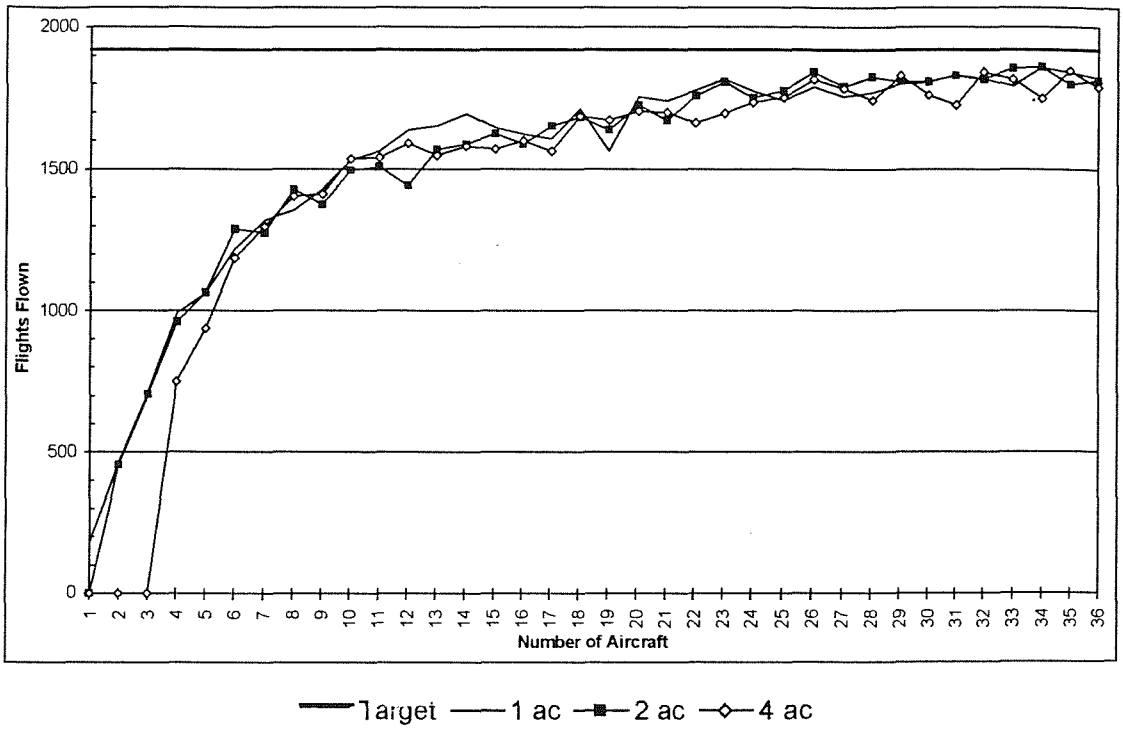


Figure 30. Flights Flown for the 1920 Flights Option.

Examination of figure 30 shows that, for the 1 aircraft per mission option, the curve of the graph shows a close correlation with that derived for the 2 aircraft per mission baseline case throughout the range of maximum aircraft available. For the 4 aircraft per mission the results are close to those for the baseline case but there is some deviation below 12 aircraft and above 21 aircraft available. The former is a direct result of the requirement to have a minimum number of aircraft available before a mission will be launched. Thus for a 4 aircraft per mission tasking no flights will be flown where the maximum possible number of aircraft available is less than 4. Above 21 aircraft the 4 aircraft per mission option tends to underperform the other 2 options slightly. This leads to the conclusion that whilst the number of aircraft required per each mission has some impact on the total number of flights launched for any given maximum aircraft availability this impact is not marked.

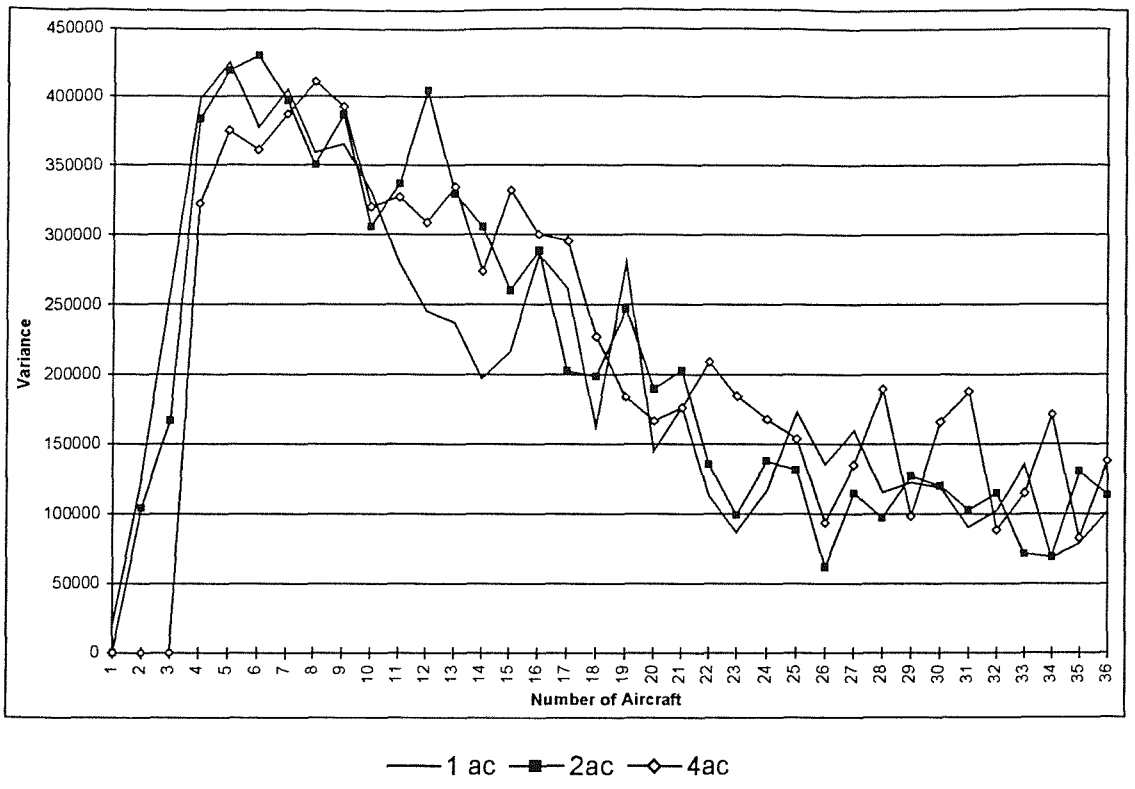


Figure 31. Variance for the 1920 Flights Option.

For this conclusion to be supported it would be expected that the variance for the 1 aircraft per mission and the 2 aircraft per mission option would be similar whilst that for the 4 aircraft per mission would be greater than the others. Indeed, examination of figure 31 supports this conclusion but reveals that the variance at the higher number of aircraft availability is markedly reduced when compared to that for the earlier runs. Thus both graphs support the view that increasing the number of aircraft required per mission has little overall impact on the final achievement at the higher number of aircraft availability. Neither of these graphs provide sufficient information to allow an objective view to be taken of why this should be so and it is, therefore, necessary to examine aircraft availability in order to ascertain whether that explains the high task achievement.

Figure 32 shows the mean aircraft availability for the various mission options. Examination of this graphs shows that, with the exception of the start where minimum aircraft requirements plays a large part, the mean aircraft availability rises at a relatively constant rate as the number of aircraft available increases. There is evidence of a curve with the increases in availability less for as the available aircraft is incremented at the higher numbers. Moreover, the high overall system availability supports the high flight

achievements and also shows a good correlation with the outputs of the deterministic model which gave an overall system availability of 80% for the 36 aircraft case.

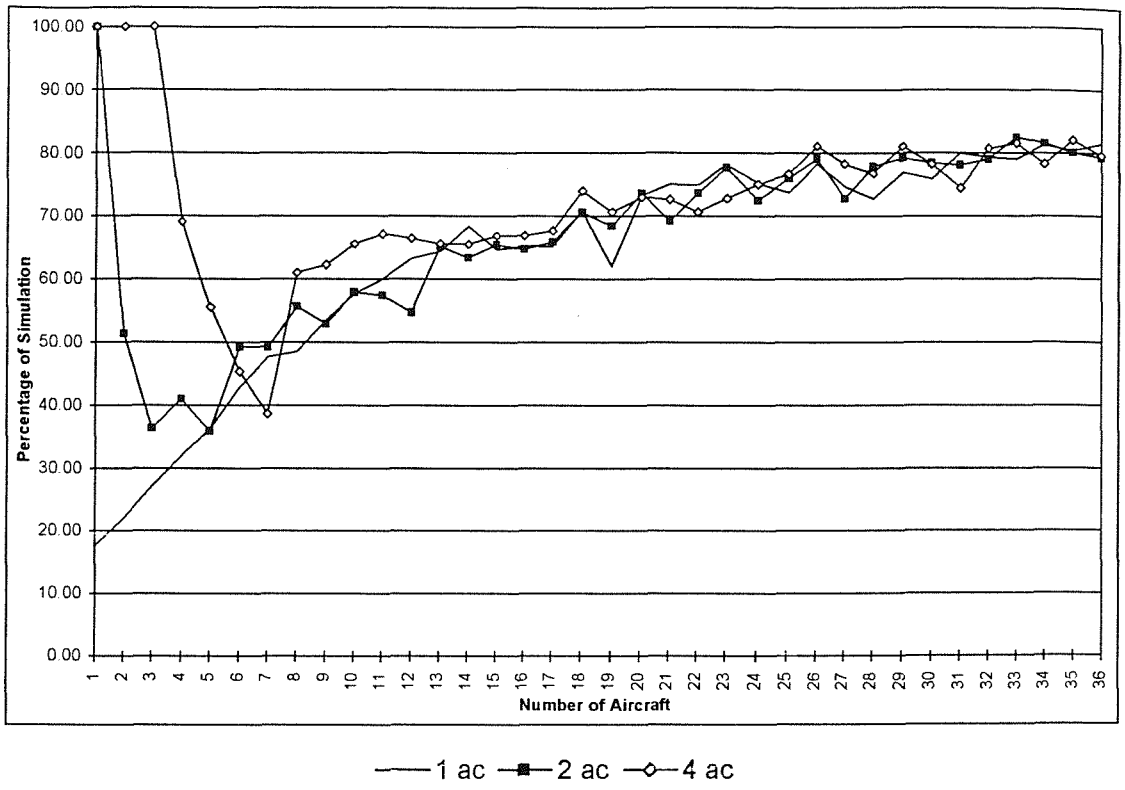


Figure 32. Mean Aircraft Availability for the 1920 Flights Option.

Whilst figures 30 to 32 are useful for examining the overall effect of the 3 alternative flying programmes, they are not suitable for identifying whether the mission profile is sustainable beyond the 166th day. Focusing on a single case allows a deeper examination of the 3 options in order to make this judgement and it appropriate to consider 2 measures: the number of flights flown per day and the aircraft availability per day. Clearly, this could be done for each of the number of aircraft available runs, however, this is both time consuming and potentially confusing. The 36 aircraft results reflect the number of aircraft against which the original scale was derived and for this reason was chosen as the case to be examined.

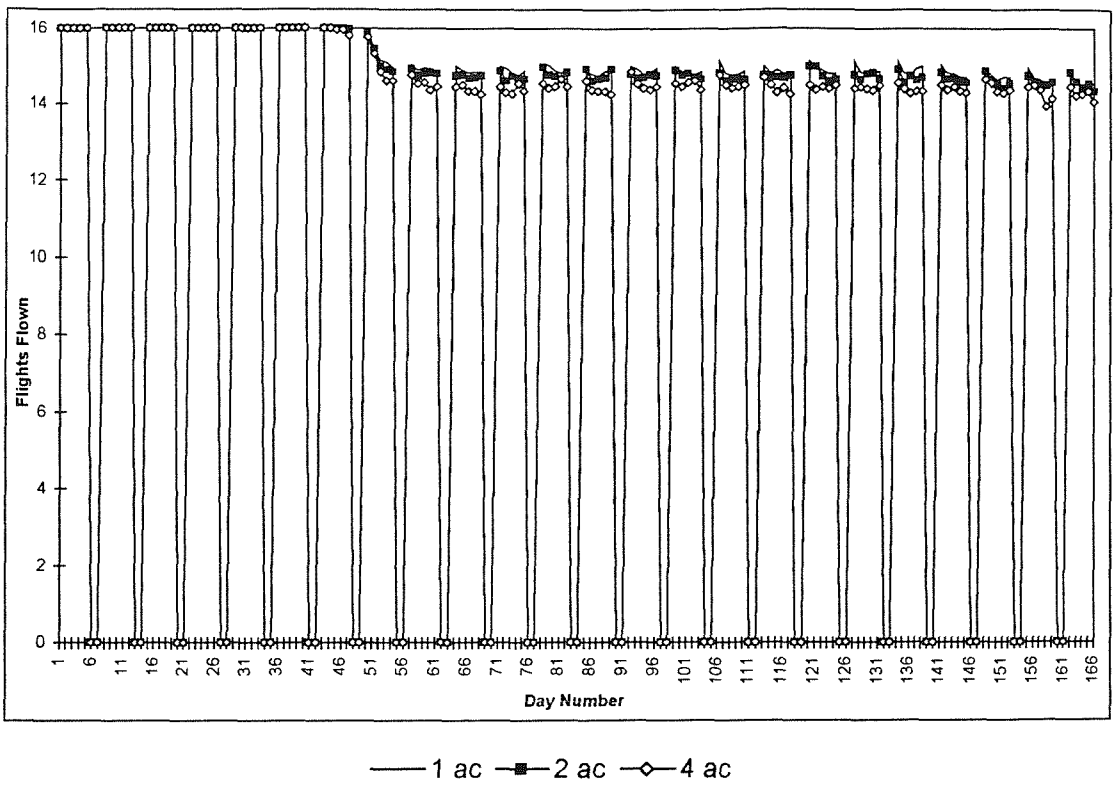


Figure 33. 36 Aircraft Mean Daily Flights Flown for the 1920 Flights Option.

Figure 33 shows the number of flights flown per day for each of the aircraft per mission options. It is immediately noticeable that there is a regular drop to 0 which reflects the fact that missions are only tasked for the first 5 days of each week. In all cases the number of missions flown is initially 100% until day 50 at which point there is a step down to a new steady state figure of approximately 90%. The small peaks at the beginning of each week reflects the effect of aircraft recovery on days 6 and 7 without any tasking.

Examination of figure 34 below reveals several interesting features. Firstly, it can be confirmed that the availability increases every day 6 and 7 as was surmised from the increased flight achievement at the start of each week. Secondly, the 1 aircraft per mission tasking results in an improved aircraft availability when compared with the baseline case as each aircraft becoming available allows a mission to be flown. This results in aircraft failing earlier than the baseline case and consequently becoming available sooner. It would be expected, using the same logic, that the 4 aircraft per mission case would result in a reduced availability, however, this is not the case as can be seen in figure 34.

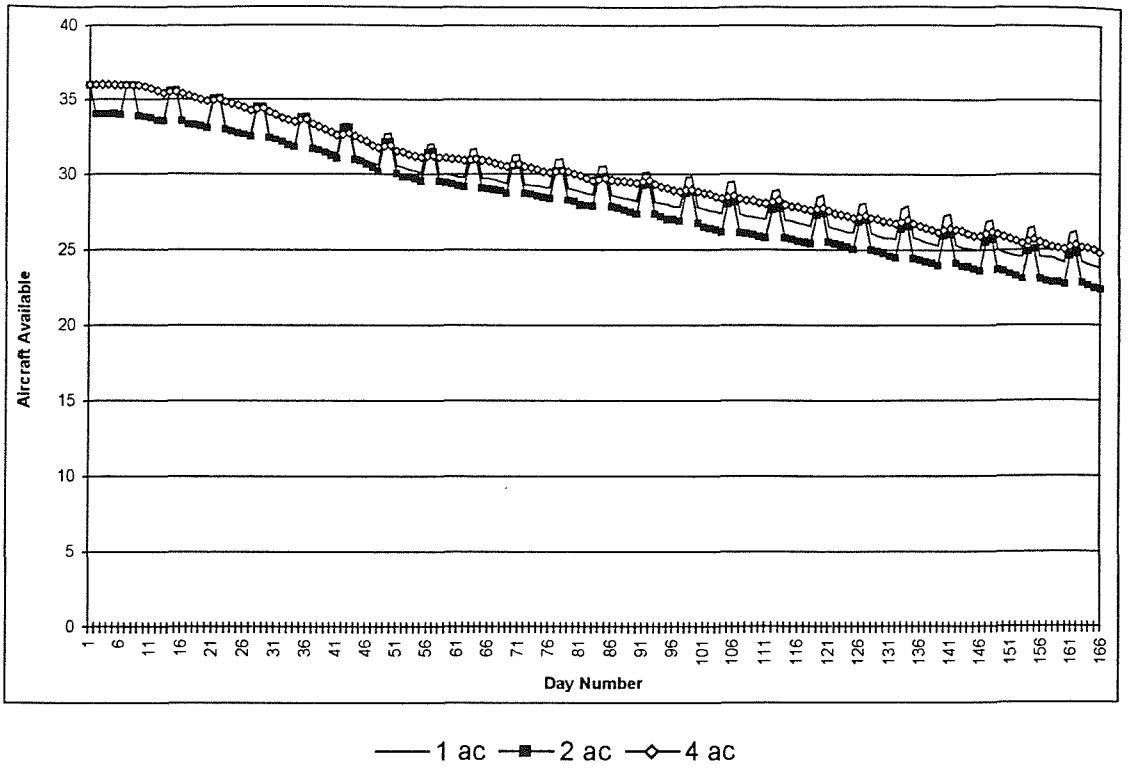


Figure 34. 36 Aircraft Mean Daily Aircraft Availability for the 1920 Flights Option.

This reverse of the expectation is as a consequence of the number of aircraft required to be available before a mission is launched resulting in more aircraft being available than for the baseline case whilst still insufficient to meet the minimum of 4 required for a mission. It is also of interest to note that, although the number of missions launched per day has a simple step function, the number of aircraft available each day for all the alternative options decays throughout the simulation and has yet to reach a steady state. Thus it can reasonably be concluded that there will come a point at which the flying programme becomes unviable but, that the information available does not permit that time to be calculated.

Variation in number of aircraft per mission with total number of flights also varying

Examination of alternative factor changes can be carried out in a similar manner to that of varying the number of flights per mission detailed above. The baseline case was derived using 8 missions of 2 flights per mission per day with a flying week of 5 days, equivalent to Monday to Friday. Thus a total of 1920 flights were tasked over a period of 166 days. By applying the same mission timings as for the baseline case but changing the number of flights required per mission examination of the effect of either halving or doubling the total flights to be flown can be undertaken.

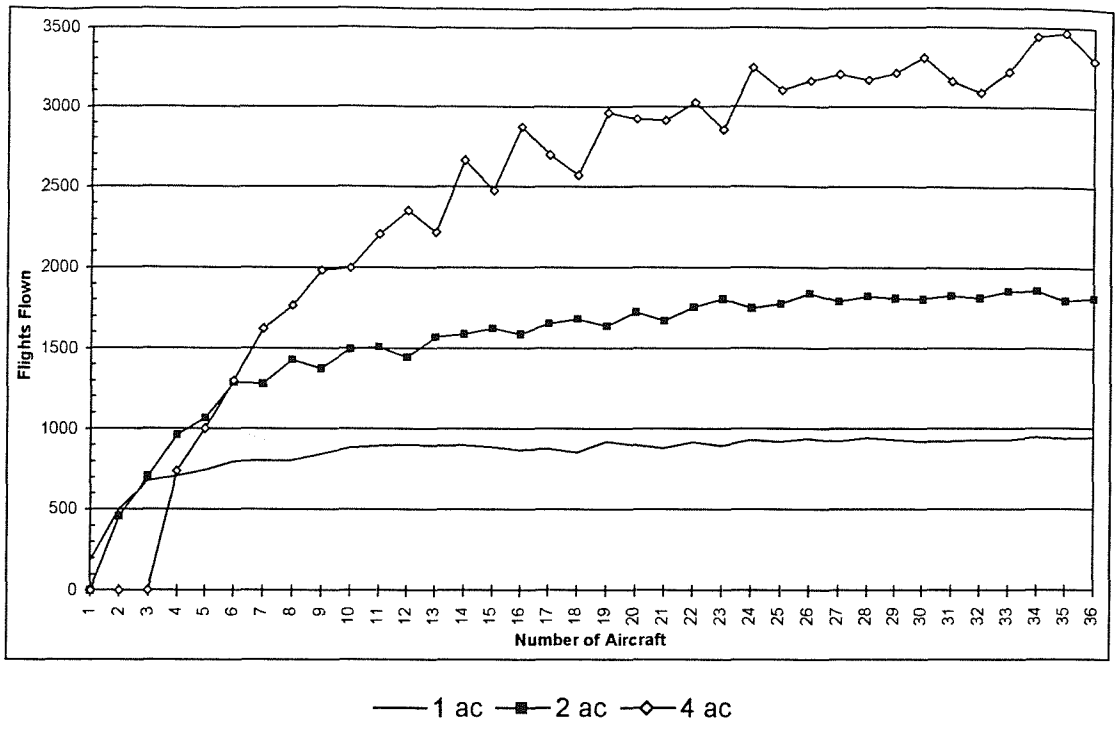


Figure 35. Flights Flown for the Variable Flights Option.

Figure 35 shows the flight achievements for the 3 options. However, as the target number of flights to be flown is different for each of the 3 options further manipulation of the data is required to allow comparison of the various achievements.

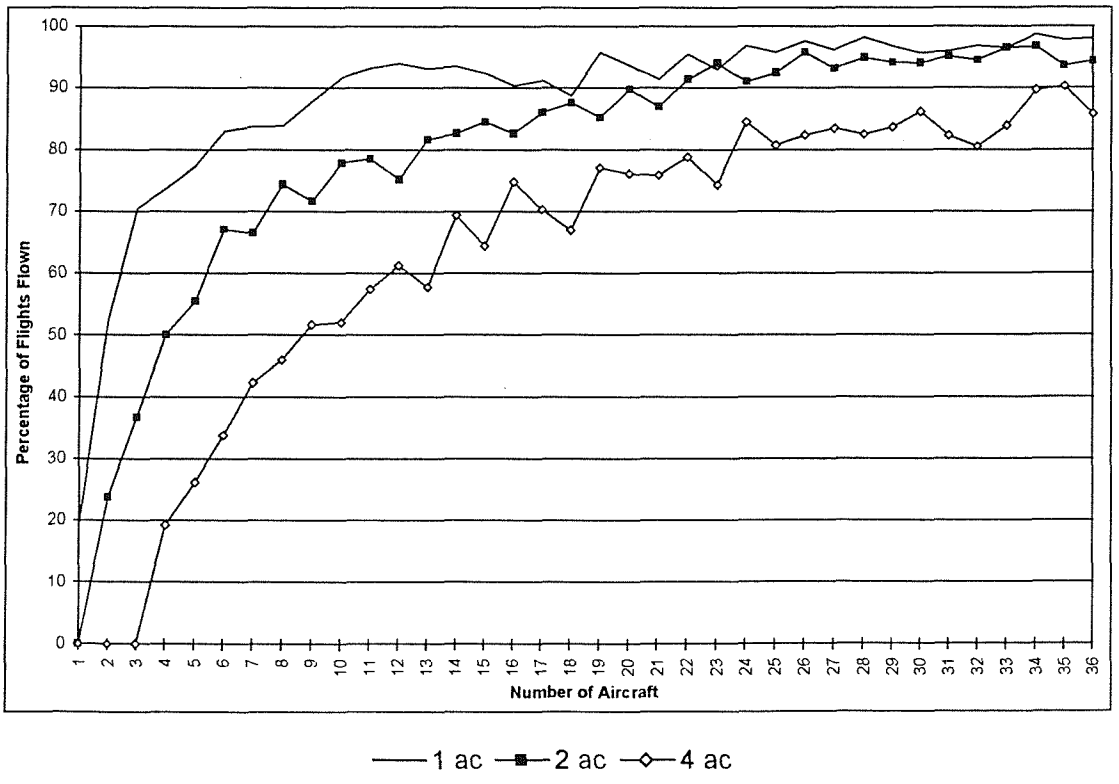


Figure 36. Percentage of Flights Flown for the Variable Flights Option.

In order to accomplish an effective comparison it is necessary to assess the achievement against the number of flights actually tasked and that is shown in figure 36.This comparison reveals a set of results which accords with what it would be reasonable to expect. The 1 aircraft option reflects half as many flights required as the baseline, 960 rather than 1920, and results in a much quicker increase to a maximum. What is interesting to note is to observe that the reduction in flights is still insufficient to reach 100% showing that aircraft unserviceability still reduces the achievement, albeit only in the order of 2% rather than 6% for the baseline. Doubling the number of flights required to 3840 results in a marked reduction in achievement throughout the range. This is as expected as the harder the fleet is worked the greater the number of unservicabilities. As the repair time for components is significant for a number of LRIs this result suggests a fleet with a declining number of aircraft available due to more LRIs contained within the repair chain than was intended when the scale was derived. As the scaling was undertaken for a fleet of 36 aircraft flying 1920 flights this result is not unexpected.

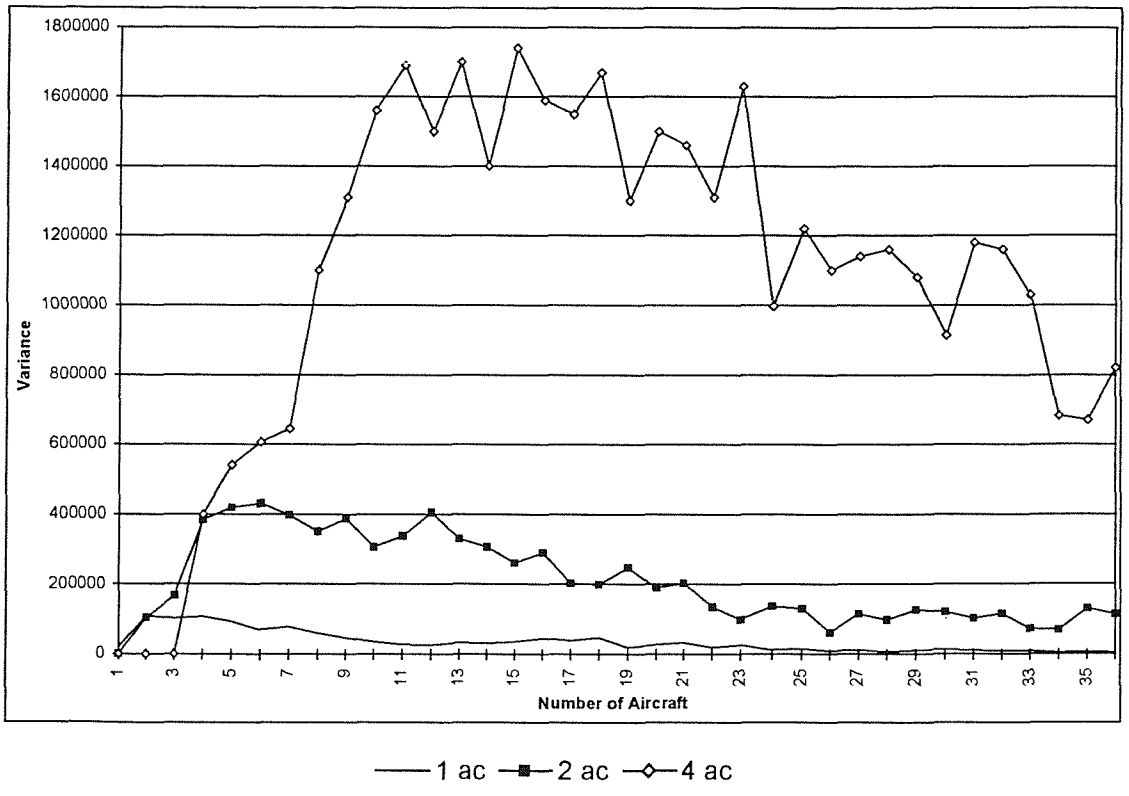


Figure 37. Variance for the Variable Flights Option.

Examination of the figure does show that there is a very large difference in the variance and the shape of the variance curves particularly for the 4 aircraft per mission option. However, as with the examination of flight achievement the direct comparison of variance

as portrayed in figure 37 is not useful as each relates to a different target and hence a different outcome. There is, therefore, a need to use another criteria against which the examination can be undertaken. An examination of the 90% confidence range would give an indication of the acceptability of the results as a means of judging the effect of altering the number of flights to be targeted.

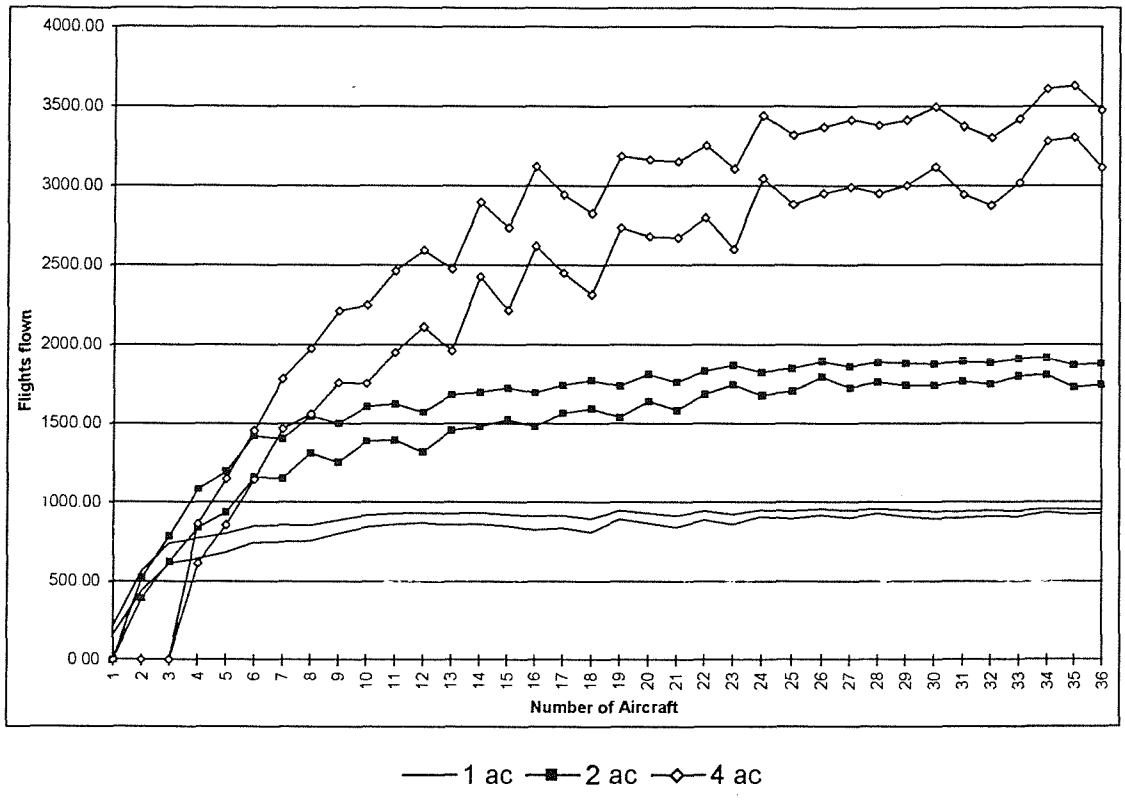


Figure 38. Upper and Lower 90% Confidence Boundaries for the Variable Flights Option.

Again this graph, whilst giving an impression of the width of the various bands, fails to give sufficient information to make a judgement at to how the final result compares. In order to achieve that the results should be converted into a percentage of the number of flights tasked which would give an idea of the spread compared against a common base. Figure 39 goes one stage further in that it shows the distance in from the achieved mean of the lower confidence limit compared with the target number of flights. Thus an achievement of 87% with a lower confidence band value of 80% will have a distance of 7%. Clearly this is the same value for the upper confidence limit as this lies as far above the achieved value as the lower limit lies below it.

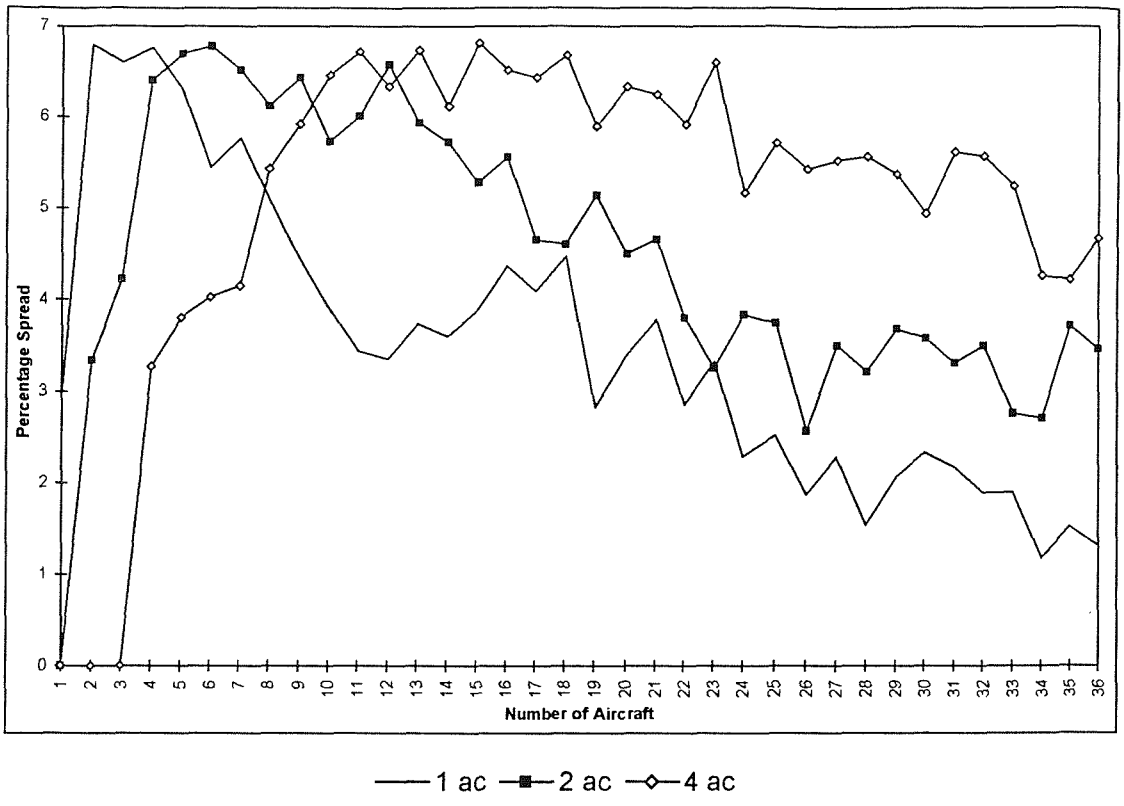


Figure 39. Distance of the Lower Confidence Limit from the Mean for the Variable Flights Option.

It can be seen that the larger the number of flights required to be mounted the greater the Confidence spread and, therefore, the wider the range of answers that would be statistically acceptable. Thus, as the number of flights is increased the probability that a satisfactory number of flights would be flown becomes increasingly unlikely and action would need to be taken to ensure that the achievement does not fall to an unacceptably low

CONCLUSIONS

The use of deterministic models to derive spares scales to support an operational task has been a key component of the RAF's modelling toolset for some years although, it was recognised at an early stage that, whilst the output was of use to maintenance staffs, the model failed to give the operational customers confidence that their needs would be met. This need could best be met by the development of a simulation application which would the effectiveness of a scale derived by a deterministic spares scaling application to be examined against a set flying programme. Whilst this has been technically feasible for some time the only attempt made by the RAF was both slow in operation and limited in functionality restricting its use to relatively simple questions of the total number of

missions that could be flown. The introduction of modern programming languages with built in functionality and the increasing power of desktop computers has allowed the development of a simulation that is both quick in operation, minutes rather than hours, and powerful in terms of the functionality that can be included. By taking advantage of these improvements, the development of ALSSim has allowed a wider range of questions to be asked ranging from the number of flights that are launched through to the mean aircraft availability. Simulation allows a greater fidelity of answer than is easily achieved with a deterministic model and allows the researcher to identify the effect of reducing the overall number of aircraft available through to the effect on a particular number of aircraft of varying the flying programme. Moreover, examination of the effect of changing the target flying rate can be easily studied as shown above allowing a judgement to be made as to whether it is appropriate or, indeed, possible to do so. Furthermore, the use of a simulation also provides the data necessary to undertake deeper investigation into the reasons why particular trends are taking place, something that cannot be done with the more simple deterministic models. Therefore, simulation has a place in the toolset of the modern aircraft logistic support analyst and provides a powerful tool capable of both providing the operational customer with a greater level of confidence in the proposed scales and allowing better ways of managing particularly scarce assets to be proposed.

FURTHER WORK

This study has resulted in an effective simulation programme which allows a number of fundamental questions to be asked relating to the ability of a given aircraft type with known equipment and reliability to achieve a stated flying programme with a particular scale which has been optimised using a deterministic application. To date the operation of the aircraft at a single base supported by a single repair depot has been considered. This could now be expanded to take account of multiple operating bases and/or repair sites allowing a wider range of questions to be asked. The model has been designed around the need to develop a simulation for aircraft and could, therefore, be used to support any organisation operating aircraft although, for fleets such as airlines, which take off and land at different locations, further development of the model would be necessary. Equally the model could be used with relatively minor alteration to examine any vehicle or piece of equipment which was subject to failures related to operating hours.

APPENDIX ONE ALSSIM COMPUTER CODE

Overview

This appendix contains all the source code for the ALSSim application except for that which is generated by the Microsoft C++ compiler as the underlying application control code. Thus only those files that have been modified by myself to incorporate additional code are included in this appendix. For each component of the simulation application there are 2 files, the header file in which all the parameters and operations are declared and the program file which contains the application code for the various operations. The first section of this appendix lists the code files in alphabetical order and gives a brief statement as to their purpose. The second section gives the code for each file in turn.

Aircraft - The code specific to the aircraft entity and the code to control LRIs fitted to the aircraft.

ALSSimDoc - The code that controls the simulation. These files contain the event controller and all the control code for each event.

DailyResults - The code used to calculate aircraft availability at the start of each day and the flight achievements for each day for each run. The results are kept in a linked list and used at the end of each set of simulation to calculate the overall results.

DelayedFlight - The code relating to the delayed flight list.

DlgSimulationFinished - The code for the dialog box that is generated at the end of each run to give an overview of the run achievements.

Event - The code required to populate and obtain data from the simulation event linked list.

RandomNumber - The code for the pseudo random number generator. This code is NOT my own. It is the code provided by L'Ecoyer and Andres in their paper on A Random Number Generator Based on the Combination of Four LCGs.

SBarSimProgress - The code to generate a progress bar at the bottom of the screen whilst the simulation is running. This bar serves no purpose other than to indicate how far the application is through the runs.

Stock - The code for each LRI type dealing with the off aircraft aspects. This file contains details of serviceable stock levels at both unit and depot, repair times, numbers required for aircraft and numbers unserviceable at both the unit and depot.

Aircraft

```
//Aircraft.h interface of the CAircraft class
class CAircraft : public CObject
{
//Attributes
private:
    bool bFirstLRI;
    bool bFirstTime;
    bool bInFlightAbortState;
    bool bLRIFailureFound;
    bool bLRIRequired;
    float fAircraftFailureTime;
    float fAircraftFlyingHours;
    float fAircraftPlannedLandingTime;
    float fClockTime;
    float fLastEventTime;
    float fLRIFailureTime;
    float fTimeAwaitingPreFlightServicing;
    float fTimeFlying;
    float fTimeInPreFlightServicing;
    float fTimeServiceable;
    float fTimeUnserviceable;
    float fTotalTimeAwaitingPreFlightServicing;
    float fTotalTimeFlying;
    float fTotalTimeInPreFlightServicing;
    float fTotalTimeServiceable;
    float fTotalTimeUnserviceable;
    int iAircraftNumber;
    int iAircraftState;
    int iLRIPosition;
    int iLRIState;
    int iLRIType;
    int iNumberOfFailedEssentialLRIs;
    int iNumberOfFailedNonEssentialLRIs;
    int iNumberOfLRIsRemoved;
    int iNumberOfLRIsRequired;
    int iNumberOfLRIsUnserviceable;
    // Class declaration for LRIs
    class CLRI : public CObject
    {
    //Attributes
    private:
        float fLRIFailureTime;
        int iLRIPosition;
        int iLRIState;
        int iLRIType;
    //Operations
    public:
        CLRI(){}
        CLRI(int iPosition,int iType,float fFailureTime)
        {
```

```
            iLRIPosition = iPosition;
            iLRIState = 1;
            iLRIType = iType;
            fLRIFailureTime = fFailureTime;
        }
        float GetLRIFailureTime();
        int GetLRIPosition();
        int GetLRIState();
        int GetLRIType();
        void ResetLRI(float fFailureTime);
        void SetLRIFailureTime(float fAircraftFlyingHours,float fMeanFailureTime);
        void SetLRIState(int iState);
        ~CLRI(){}
    };
    CObList lrilist;
    CLRI* pLRI;
    POSITION LRIPos;
//Operations for the Aircraft Class
public:
    CAircraft(){}
    CAircraft(int iNumber)
    {
        iAircraftNumber = iNumber;
        iAircraftState = 1;
        iNumberOfFailedEssentialLRIs = 0;
        iNumberOfFailedNonEssentialLRIs = 0;
        iNumberOfLRIsRemoved = 0;
        iNumberOfLRIsRequired = 0;
        iNumberOfLRIsUnserviceable = 0;
        fAircraftFlyingHours = 0.0;
        fTimeAwaitingPreFlightServicing = 0.0;
        fLastEventTime = 0.0;
        fTimeFlying = 0.0;
        fTimeInPreFlightServicing = 0.0;
        fTimeUnserviceable = 0.0;
        fTimeServiceable = 0.0;
    }
    bool CheckLRITypeRequired(int iLRIType);
    bool GetInFlightAbortState();
    bool ResetLRI(float fFailureTime);
    float GetAircraftFlyingHours();
    float GetAircraftFailureTime();
    float GetAircraftPlannedLandingTime();
    float GetLRIFailureTime(int iLRIPosition);
    float GetTimeAwaitingPreFlightServicing(int iNumberOfRuns);
    float GetTimeFlying(int iNumberOfRuns);
    float GetTimeInPreFlightServicing(int iNumberOfRuns);
    float GetTimeServiceable(int iNumberOfRuns);
    float GetTimeUnserviceable(int iNumberOfRuns);
    int AddNewLRI(int iPosition,int iType,float fFailureTime);
    int GetAircraftNumber();
```

```
    int GetAircraftState();
    int GetNumberOfFailedEssentialLRIs();
    int GetNumberOfFailedNonEssentialLRIs();
    int GetNumberOfLRIsRequired();
    int GetNumberOfLRIsUnserviceable();
    int GetLRIFailedInFlight(bool bFirstTime,float fEventDuration);
    int GetLRIState();
    int GetLRIType(bool bFirstTime);
    int GetUnserviceableLRIType();
    int RemoveUnserviceableLRIs(bool bFirstTime);
    int UpdateLRIRequired(int iType,int iState);
    void DeleteLRIs();
    void IncrementNumberOfEssentialLRIFailures();
    void IncrementNumberOfNonEssentialLRIFailures();
    void ReduceNumberOfLRIsRequired();
    void ResetAircraft();
    void SavePlannedLandingTime(float fClockTime,float fEventDuration);
    void SetAircraftState(int iAircraftState);
    void SetEndOfRunTotals(float fClockTime);
    void SetLRIState(int iLRIPosition,int iLRIState);
    void SetInFlightAbortState(bool bInFlightAbort);
    void SetTimeAwaitingPreFlightServicing(float fClockTime);
    void SetTimeFlying(float fClockTime);
    void SetTimeInPreFlightServicing(float fClockTime);
    void SetTimeServiceable(float fClockTime);
    void SetTimeUnserviceable(float fClockTime);
    void UpdateAircraftOnLanding(float fClockTime,float fFlightDuration,
        int iMaximumFailuresNonEssentialLRIs);
    void UpdateRemovedLRI(int iPosition,int iState);
    void UpdateReplacedLRI(int iPosition,float fFailureTime);
    ~CAircraft(){}
};
//Aircraft.cpp :implementation of the CAircraft class
#include "stdafx.h"
#include "Aircraft.h"
// CAircraft commands
bool CAircraft::CheckLRITypeRequired(int iType)
{
    bLRIRequired = false;
    for (LRIPos = lrilist.GetHeadPosition();LRIPos != NULL;)
    {
        pLRI = (CLRI*)lrilist.GetNext(LRIPos);
        iLRIType = pLRI->GetLRIType();
        if (iLRIType == iType)
        {
            if ( pLRI->GetLRIState() == 4)
            {
                bLRIRequired = true;
                break;
            }
        }
```

```
            else
            {
                if (iLRIType > iType)
                    break;
            }
        }
        return bLRIRequired;
}
bool CAircraft::GetInFlightAbortState()
{
    return bInFlightAbortState;
}
bool CAircraft::ResetLRI(float fFailureTime)
{
            pLRI = (CLRI*)lrilist.GetAt(LRIPos);
            pLRI->ResetLRI(fFailureTime);
            pLRI = (CLRI*)lrilist.GetNext(LRIPos);
            if (LRIPos == NULL)
                return true;
            else
                return false;
}
float CAircraft::GetAircraftFailureTime()
{
    return fAircraftFailureTime;
}
float CAircraft::GetAircraftFlyingHours()
{
    return fAircraftFlyingHours;
}
float CAircraft::GetAircraftPlannedLandingTime()
{
    return fAircraftPlannedLandingTime;
}
float CAircraft::GetLRIFailureTime(int iLRIPosition)
{
    fLRIFailureTime = pLRI->GetLRIFailureTime();
    return fLRIFailureTime;
}
float CAircraft::GetTimeAwaitingPreFlightServicing(int iNumberOfRuns)
{
    return fTimeAwaitingPreFlightServicing/iNumberOfRuns;
}
float CAircraft::GetTimeFlying(int iNumberOfRuns)
{
    return fTimeFlying/iNumberOfRuns;
}
float CAircraft::GetTimeInPreFlightServicing(int iNumberOfRuns)
{
    return fTimeInPreFlightServicing/iNumberOfRuns;
}
```

```
float CAircraft::GetTimeServiceable(int iNumberOfRuns)
{
    return fTimeServiceable/iNumberOfRuns;
}
float CAircraft::GetTimeUnserviceable(int iNumberOfRuns)
{
    return fTimeUnserviceable/iNumberOfRuns;
}
int CAircraft::AddNewLRI(int iLRIPosition,int iLRIType,float fFailureTime)
{
    if (iLRIPosition == 1|| fFailureTime < fAircraftFailureTime)
    {
        fAircraftFailureTime = fFailureTime;
    }
    lrilist.AddTail(new CLRI(iLRIPosition,iLRIType,fFailureTime));
    iLRIPosition++;
    return iLRIPosition;
}
int CAircraft::GetAircraftNumber()
{
    return iAircraftNumber;
}
int CAircraft::GetAircraftState()
{
    return iAircraftState;
}
int CAircraft::GetNumberOfFailedEssentialLRIs()
{
    return iNumberOfFailedEssentialLRIs;
}
int CAircraft::GetNumberOfFailedNonEssentialLRIs()
{
    return iNumberOfFailedNonEssentialLRIs;
}
int CAircraft::GetLRIFailedInFlight(bool bFirstTime,float fEventDuration)
{
    bLRIFailureFound = false;
    if (bFirstTime == true)
    {
        LRIPos = lrilist.GetHeadPosition();
    }
    pLRI = (CLRI*)lrilist.GetNext(LRIPos);
    while (LRIPos != NULL)
    {
        // Check if the LRI fails this flight
        if (pLRI->GetLRIFailureTime() <= fAircraftFlyingHours + fEventDuration
            && pLRI->GetLRIState() == 1)
        {
            bLRIFailureFound = true;
            //get LRI position
            iLRIPosition = pLRI->GetLRIPosition();
```

```
                break;
            }
            else
            {
                // step on to next LRI in the list
                pLRI = (CLRI*)lrilist.GetNext(LRIPos);
            }
        }
        //return LRI position
        if (bLRIFailureFound)
        {
            return iLRIPosition;
        }
        else
        {
            return (0);
        }
}
int CAircraft::GetLRIState()
{
    return iLRIState;
}
int CAircraft::GetLRIType(bool bFirstTime)
{
    if (bFirstTime)
    {
        LRIPos = lrilist.GetHeadPosition();
    }
    pLRI = (CLRI*)lrilist.GetAt(LRIPos);
    iLRIType = pLRI->GetLRIType();
    return iLRIType;
}
int CAircraft::GetNumberOfLRIsRequired()
{
    return iNumberOfLRIsRequired;
}
int CAircraft::GetNumberOfLRIsUnserviceable()
{
    return iNumberOfLRIsUnserviceable;
}
int CAircraft::GetUnserviceableLRIType()
{
    iLRIType = pLRI->GetLRIType();
    return iLRIType;
}
int CAircraft::RemoveUnserviceableLRIs(bool bFirstTime)
{
    if (bFirstTime)
    {
        LRIPos = lrilist.GetHeadPosition();
    }
```

```
    pLRI = (CLRI*)lrilist.GetAt(LRIPos);
    if (pLRI->GetLRIState() == 2)
    {
        // change LRI State to Being removed(3)
        iLRIState = 3;
        pLRI->SetLRIState(iLRIState);
        iNumberOfLRIsUnserviceable--;
        iNumberOfLRIsRemoved++;
        //return LRI position
        iLRIPosition = pLRI->GetLRIPosition();
        return iLRIPosition;
    }
    else
    {
        // step on to next LRI in the list
        pLRI = (CLRI*)lrilist.GetNext(LRIPos);
        return (0);
    }
}
int CAircraft::UpdateLRIRequired(int iType,int iState)
{
    for (LRIPos = lrilist.GetHeadPosition();LRIPos != NULL;)
    {
        pLRI = (CLRI*)lrilist.GetNext(LRIPos);
        if ((pLRI->GetLRIType() == iType) && (pLRI->GetLRIState() == 4))
        {
            pLRI->SetLRIState(iState);
            iLRIPosition = pLRI->GetLRIPosition();
            break;
        }
    }
    return iLRIPosition;
}
void CAircraft::DeleteLRIs()
{
    POSITION LRIPos = lrilist.GetHeadPosition();
    // Delete the LRI objects
    while (LRIPos != NULL)
    {
        delete lrilist.GetNext(LRIPos);
    }
    lrilist.RemoveAll();
}
void CAircraft::IncrementNumberOfEssentialLRIFailures()
{
    iNumberOfFailedEssentialLRIs++;
}
void CAircraft::IncrementNumberOfNonEssentialLRIFailures()
{
    iNumberOfFailedNonEssentialLRIs++;
}
```

```
void CAircraft::ReduceNumberOfLRIsRequired()
{
    iNumberOfLRIsRequired--;
}
void CAircraft::ResetAircraft()
{
    iAircraftState = 1;
    iNumberOfFailedEssentialLRIs = 0;
    iNumberOfFailedNonEssentialLRIs = 0;
    iNumberOfLRIsRequired = 0;
    iNumberOfLRIsRemoved = 0;
    iNumberOfLRIsUnserviceable = 0;
    fAircraftFlyingHours = 0.0;
    fLastEventTime = 0.0;
    // Reset first failure time for each installed LRI
    LRIPos = lrilist.GetHeadPosition();
    pLRI = (CLRI*)lrilist.GetNext(LRIPos);
    fAircraftFailureTime = pLRI->GetLRIFailureTime();
    while (LRIPos != NULL)
    {
        pLRI = (CLRI*)lrilist.GetNext(LRIPos);
        fLRIFailureTime = pLRI->GetLRIFailureTime();
        if (fLRIFailureTime < fAircraftFailureTime)
            fAircraftFailureTime = fLRIFailureTime;
    }
}
void CAircraft::SavePlannedLandingTime(float fClockTime,float fEventDuration)
{
    fAircraftPlannedLandingTime = fClockTime + fEventDuration;
}
void CAircraft::SetAircraftState(int iState)
{
    iAircraftState = iState;
}
void CAircraft::SetEndOfRunTotals(float fClockTime)
{
    // Update counter for state at the end of the run
    switch (iAircraftState)
    {
    case 1:
        SetTimeServiceable(fClockTime);
        break;
    case 2:
        SetTimeFlying(fClockTime);
        break;
    case 3:
        SetTimeUnserviceable(fClockTime);
        break;
    case 4:
        SetTimeInPreFlightServicing(fClockTime);
        break;
```

```
        case 5:
            SetTimeAwaitingPreFlightServicing(fClockTime);
            break;
        }
}
void CAircraft::SetInFlightAbortState(bool bAbortState)
{
    bInFlightAbortState = bAbortState;
}
void CAircraft::SetLRIState(int iPosition,int iState)
{
    for (LRIPos = lrilist.GetHeadPosition();LRIPos != NULL;)
    {
        pLRI = (CLRI*)lrilist.GetNext(LRIPos);
        if (pLRI->GetLRIPosition() == iPosition)
        {
            iLRIState = iState;
            pLRI->SetLRIState(iLRIState);
            break;
        }
    }
}
void CAircraft::SetTimeAwaitingPreFlightServicing(float fClockTime)
{
    fTimeAwaitingPreFlightServicing += fClockTime - fLastEventTime;
    fLastEventTime = fClockTime;
}
void CAircraft::SetTimeFlying(float fClockTime)
{
    fTimeFlying += fClockTime - fLastEventTime;
    fLastEventTime = fClockTime;
}
void CAircraft::SetTimeInPreFlightServicing(float fClockTime)
{
    fTimeInPreFlightServicing += fClockTime - fLastEventTime;
    fLastEventTime = fClockTime;
}
void CAircraft::SetTimeServiceable(float fClockTime)
{
    fTimeServiceable += fClockTime - fLastEventTime;
    fLastEventTime = fClockTime;
}
void CAircraft::SetTimeUnserviceable(float fClockTime)
{
    fTimeUnserviceable += fClockTime - fLastEventTime;
    fLastEventTime = fClockTime;
}
void CAircraft::UpdateAircraftOnLanding(float fClockTime,float fFlightDuration,
    int iMaxNonEssentialFailures)
{
    fAircraftFlyingHours += fFlightDuration;
```

```
    // Update Time Flying Counter
    SetTimeFlying(fClockTime);
    // Check for failures and set aircraft state accordingly
    if(iNumberOfFailedEssentialLRIs > 0 ||
        iNumberOfFailedNonEssentialLRIs > iMaxNonEssentialFailures)
    {
        // Change aircraft state to Unserviceable(3)
        iAircraftState = 3;
        iNumberOfLRIsUnserviceable = iNumberOfFailedEssentialLRIs +
            iNumberOfFailedNonEssentialLRIs;
        iNumberOfFailedEssentialLRIs = 0;
        iNumberOfFailedNonEssentialLRIs = 0;
    }
    else
    {
        // Change aircraft state to In Flight Servicing(4)
        iAircraftState = 4;
    }
}
void CAircraft::UpdateRemovedLRI(int iPosition,int iState)
{
    if (iState == 4)
        iNumberOfLRIsRequired++;
    for (LRIPos = lrilist.GetHeadPosition();LRIPos != NULL;)
    {
        pLRI = (CLRI*)lrilist.GetNext(LRIPos);
        if (pLRI->GetLRIPosition() == iPosition)
        {
            pLRI->SetLRIState(iState);
            break;
        }
    }
}
void CAircraft::UpdateReplacedLRI(int iPosition,float fFailureTime)
{
    for (LRIPos = lrilist.GetHeadPosition();LRIPos != NULL;)
    {
        pLRI = (CLRI*)lrilist.GetNext(LRIPos);
        iLRIPosition = pLRI->GetLRIPosition();
        if (iLRIPosition == iPosition)
        {
            // Change LRI state to Serviceable Fitted(1)
            iLRIState = 1;
            pLRI->SetLRIState(iLRIState);
            pLRI->SetLRIFailureTime(fAircraftFlyingHours,fFailureTime);
            iNumberOfLRIsRemoved--;
            // All LRIs fitted and Serviceable
            if (iNumberOfLRIsRemoved == 0)
            {
                // Update the next failure time for the aircraft
                LRIPos = lrilist.GetHeadPosition();
```

```
                pLRI = (CLRI*)lrilist.GetNext(LRIPos);
                fAircraftFailureTime = pLRI->GetLRIFailureTime();
                while (LRIPos != NULL)
                {
                    pLRI = (CLRI*)lrilist.GetNext(LRIPos);
                    fLRIFailureTime = pLRI->GetLRIFailureTime();
                    if (fLRIFailureTime < fAircraftFailureTime)
                        fAircraftFailureTime = fLRIFailureTime;
                }
                // Change Aircraft state to in Flight Servicing
                iAircraftState = 4;
                break;
            }
        }
        else
        {
            if (iLRIPosition > iPosition)
                break;
        }
    }
}
// CLRI commands
int CAircraft::CLRI::GetLRIPosition()
{
    return iLRIPosition;
}
int CAircraft::CLRI::GetLRIState()
{
    return iLRIState;
}
int CAircraft::CLRI::GetLRIType()
{
    return iLRIType;
}
float CAircraft::CLRI::GetLRIFailureTime()
{
    return fLRIFailureTime;
}
void CAircraft::CLRI::ResetLRI(float fFailureTime)
{
    iLRIState = 1;
    fLRIFailureTime = fFailureTime;
}
void CAircraft::CLRI::SetLRIFailureTime(float fAircraftFlyingHours,float fFailureTime)
{
    fLRIFailureTime = fAircraftFlyingHours + fFailureTime;
}
void CAircraft::CLRI::SetLRIState(int iState)
{
    iLRIState = iState;
}
```

ALSSimDoc

```
// ALSSimDoc.h : interface of the CALSSimDoc class
#if
!defined(AFX_ALSSIMDOC_H__AC08B32B_CF11_11D1_A97A_44455354616F__IN
CLUDED_)
#define
AFX_ALSSIMDOC_H__AC08B32B_CF11_11D1_A97A_44455354616F__INCLUDED
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// Forward References
class CSimProgressStatusBar;
class CALSSimDoc : public CDocument
{
protected: // create from serialization only
    CALSSimDoc();
    DECLARE_DYNCREATE(CALSSimDoc)
// Attributes
private:
    bool bAcceptableResult;
    bool bAircraftAllocatedToFlight;
    bool bEndRun;
    bool bEvenNumberRun;
    bool bFirstTime;
    bool bFlyingProgrammeFileLoaded;
    bool bInFlightAbortState;
    bool bLastLRI;
    bool bLRIDataFileLoaded;
    bool bLRIRepairOnUnit;
    bool bLRIStockFileLoaded;
    bool bResultsFileLoaded;
    bool bSecondNormalAvailable;
    bool bSimulationParametersFileLoaded;
    bool bStoringEvents;
    CString sFlyingProgrammeFile;
    CString sLRIDataFile;
    CString sLRIStockFile;
    CString sResultsFile;
    CString sSimParametersFile;
    float fAircraftFailureTime;
    float fAircraftFlyingHours;
    float fAircraftPlannedLandingTime;
    float fClockTime;
    float fDepotUnitTransferTime;
    float feEventDuration;
    float feEventTime;
    float fePreviousEventTime;
    float fEventDuration;
    float fEventTime;
    float fFailureGamma;
```

```
float fFailureMin;
float fFailureMax;
float fFailureShape;
float fFailureVariance;
float fLastTakeOffTime;
float fLogMean;
float fLogMu;
float fLogVariance;
float fLRIDepotRepairTime;
float fLRIFailureMax;
float fLRIFailureMin;
float fLRIFailureTime;
float fLRIMeanFailureTime;
float fLRIMeanRepairTime;
float fLRIProportionRepairedAtUnit;
float fLRIRefitTime;
float fLRIRemovalTime;
float fLRIRepairTime;
float fLRIUnitRepairTime;
float fLRIVariance;
float fMaximumFlightDelay;
float fMeanFlightsCancelled;
float fMeanFlightsFail;
float fMeanFlightsFirstHalfDelay;
float fMeanFlightsInFlightAbort;
float fMeanFlightsOnTime;
float fMeanFlightsSecondHalfDelay;
float fMeanFlightsSucceed;
float fMeanFlightsTakeOff;
float fMeanFlightsTasked;
float fMeanNumberOfAircraftAwaitingPreFlightServicing;
float fMeanNumberOfAircraftFlying;
float fMeanNumberOfAircraftInPreFlightServicing;
float fMeanNumberOfAircraftServiceable;
float fMeanNumberOfAircraftUnserviceable;
float fMeanNumberOfDailyFlightsCancelled;
float fMeanNumberOfDailyFlightsFail;
float fMeanNumberOfDailyFlightsFirstHalfDelay;
float fMeanNumberOfDailyFlightsInFlightAbort;
float fMeanNumberOfDailyFlightsOnTime;
float fMeanNumberOfDailyFlightsSecondHalfDelay;
float fMeanNumberOfDailyFlightsSucceed;
float fMeanNumberOfDailyFlightsTakeOff;
float fMeanNumberOfDailyFlightsTasked;
float fMeanTimeUnserviceable;
float fMeanTimeAwaitingPreFlightServicing;
float fMeanTimeInPreFlightServicing;
float fMeanTimeServiceable;
float fMeanTimeFlying;
float fMissionSuccessPoint;
float fNoFaultFoundAtDepotFactor;
```

```
float fNoFaultFoundAtUnitFactor;
float fNormalCalculationValue;
float fNormalCheckValue;
float fNormalFinalValue1;
float fNormalFinalValue2;
float fNormalNumber;
float fNormalValue1;
float fNormalValue2;
float fNumberOfAircraftAwaitingPreFlightServicing;
float fNumberOfAircraftFlying;
float fNumberOfAircraftInPreFlightServicing;
float fNumberOfAircraftServiceable;
float fNumberOfAircraftUnserviceable;
float fNumberOfDailyFlightsCancelled;
float fNumberOfDailyFlightsFail;
float fNumberOfDailyFlightsFirstHalfDelay;
float fNumberOfDailyFlightsInFlightAbort;
float fNumberOfDailyFlightsOnTime;
float fNumberOfDailyFlightsSecondHalfDelay;
float fNumberOfDailyFlightsSucceed;
float fNumberOfDailyFlightsTakeOff;
float fNumberOfDailyFlightsTasked;
float fNumberOfFlightsCancelled;
float fNumberOfFlightsFail;
float fNumberOfFlightsSucceed;
float fNumberOfFlightsFirstHalfDelay;
float fNumberOfFlightsInFlightAbort;
float fNumberOfFlightsOnTime;
float fNumberOfFlightsSecondHalfDelay;
float fNumberOfFlightsTakeOff;
float fNumberOfFlightsTasked;
float fPercentageFlightsCancelled;
float fPercentageFlightsFail;
float fPercentageFlightsFirstHalfDelay;
float fPercentageFlightsInFlightAbort;
float fPercentageFlightsOnTime;
float fPercentageFlightsSecondHalfDelay;
float fPercentageFlightsSucceed;
float fPercentageFlightsTakeOff;
float fPreFlightServicingDuration;
float fPreviousEventTime;
float fRandomNumber;
float fRandomNumber1;
float fRandomNumber2;
float fReliabilityFactor;
float fRepairFactor;
float fRepairGamma;
float fRepairMax;
float fRepairMin;
float fRepairShape;
float fRepairVariance;
```

```
float fResult;
float fScale;
float fTimeUnserviceable;
float fTimeAwaitingPreFlightServicing;
float fTimeInPreFlightServicing;
float fTimeServiceable;
float fTimeFlying;
float fTriangleDiv;
float fTriangleMin;
float fTriangleMax;
float fTriangleMode;
float fTriangleRange;
float fVarianceFlightsCancelled;
float fVarianceFlightsFail;
float fVarianceFlightsSucceed;
float fVarianceFlightsFirstHalfDelay;
float fVarianceFlightsInFlightAbort;
float fVarianceFlightsOnTime;
float fVarianceFlightsSecondHalfDelay;
float fVarianceFlightsTakeOff;
HANDLE hFile;
hyper hRandomNumber;
int iAircraftNumber;
int iAircraftState;
int iCount;
int iDayNumber;
int ieAircraftNumber;
int ieEventNumber;
int ieLRIPosition;
int ieLRIType;
int ieNumberOfAircraft;
int ieNumberOfLRIs;
int iEventNumber;
int iFailureDistributionUsed;
int iFailureDivisor;
int  iLastRunFlightsCancelled;
int  iLastRunFlightsFirstHalfDelay;
int  iLastRunFlightsOnTime;
int  iLastRunFlightsSecondHalfDelay;
int  iLastRunFlightsTasked;
int iLRIDepotStock;
int iLRIEssential;
int iLRINumberFitted;
int iLRIPosition;
int iLRIState;
int iLRIType;
int iLRIUnitStock;
int iNumberOfAircraft;
int iNumberOfAircraftAvailable;
int iNumberOfAircraftAwaitingPreFlightServicing;
int iNumberOfAircraftFlying;
```

```
int iNumberOfAircraftInPreFlightServicing;
int iNumberOfAircraftServiceable;
int  iNumberOfAircraftUnserviceable;
int iNumberOfDailyFlightsCancelled;
int iNumberOfDailyFlightsFail;
int iNumberOfDailyFlightsFirstHalfDelay;
int iNumberOfDailyFlightsInFlightAbort;
int iNumberOfDailyFlightsOnTime;
int iNumberOfDailyFlightsSecondHalfDelay;
int iNumberOfDailyFlightsSucceed;
int iNumberOfDailyFlightsTasked;
int iNumberOfDays;
int iNumberOfFailedEssentialLRIs;
int iNumberOfFailedNonEssentialLRIs;
int iMaximumFailuresNonEssentialLRIs;
int iNumberOfFlightsCancelled;
int iNumberOfFlightsDelayed;
int iNumberOfFlightsFail;
int iNumberOfFlightsFirstHalfDelay;
int iNumberOfFlightsInFlightAbort;
int iNumberOfFlightsOnTime;
int iNumberOfFlightsSecondHalfDelay;
int iNumberOfFlightsSucceed;
int iNumberOfFlightsTakeOff;
int iNumberOfFlightsTasked;
int iNumberOfLRIs;
int iNumberOfLRIsUnserviceable;
int iNumberOfPreFlightServicingTeams;
int iNumberOfRuns;
int iRandomNumberDivisionConstant;
int iRandomNumberSeed;
int iRandomNumberStream;
int iRepairDistributionUsed;
int iRepairDivisor;
int iRunNumber;
int iSquareNumberOfFlightsCancelled;
int iSquareNumberOfFlightsFail;
int iSquareNumberOfFlightsSucceed;
int iSquareNumberOfFlightsFirstHalfDelay;
int iSquareNumberOfFlightsInFlightAbort;
int iSquareNumberOfFlightsOnTime;
int iSquareNumberOfFlightsSecondHalfDelay;
int iSquareNumberOfFlightsTakeOff;
int iStartNumberOfAircraft;
int iStartRunNumberOfPreFlightServicingTeams;
int iStockType;
int iTotalFlightsCancelled;
int iTotalFlightsFail;
int iTotalFlightsFirstHalfDelay;
int iTotalFlightsInFlightAbort;
int iTotalFlightsOnTime;
```

```
    int iTotalFlightsSecondHalfDelay;
    int iTotalFlightsSucceed;
    int iTotalFlightsTakeOff;
    int iTotalFlightsTasked;
// Operations
public:
    bool IsValidFileSpec (LPCSTR lpszFileSpec);
    float GetBoundedNormal(CObList& randomnumberlist,
        int iRandomNumberStream,float fLRIMean,float fMin,float fMax,
        float fLRIVariance);
    float GetExponentialResult(CObList& randomnumberlist,int iStreamNumber,
        float fMean);
    float GetLognormalResult(CObList& randomnumberlist,int iStreamNumber,
        float fMean,float fVariance);
    float GetNormalResult(CObList& randomnumberlist,int iStreamNumber,float fMean,
        float fVariance);
    float GetRandomNumber(CObList& randomnumberlist,int iStreamNumber);
    float GetTriangularResult(CObList& randomnumberlist,int iStreamNumber,
        float fMean,float fMin,float fMax, float fDivisor);
    float GetWeibullResult(CObList& randomnumberlist,int iStreamNumber,
        float fMean,float fShape,float fGamma);
    int GetNextEvent(CObList& aircraftlist,CObList& delayedflightlist,
        CObList& eventlist,CObList& randomnumberlist, CObList& runresultlist,
        CObList& stocklist);
    void AllocateLRIToAircraft(CObList& aircraftlist,CObList& eventlist,
        CObList& stocklist,int iLRIType);
    void CreateAircraft(CObList& aircraftlist,CObList& randomnumberlist,
        CObList& stocklist);
    void CreateDailyResults(CObList& dailyresultslist,int iNumberOfDays);
    void CreateRandomNumberStreams(CObList& randomnumberlist);
    void EndSimulation(CObList& aircraftlist,CObList& runresultlist,
        CObList& randomnumberlist,CObList& stocklist);
    void InsertEvent(CObList& eventlist, int iEventNumber, float fEventTime,
        float fPreviousEventTime,int iAircraftNumber,int iNumberOfAircraft,
        int iLRIPosition,int iLRIType,int iNumberOfLRIs,float fEventDuration);
    void LoadSimulationScenarioDetails();
    void ReleaseMemory(CObList& aircraftlist,CObList& randomnumberlist,
        CObList& runresultlist,CObList& stocklist);
    void SaveDailyFlyingStats(CObList& dailyresultslist,int iDayNumber,
        int iNumberOfDailyFlightsTasked,int iNumberOfDailyFlightsOnTime,
        int iNumberOfDailyFlightsFirstHalfDelay,
        int iNumberOfDailyFlightsSecondHalfDelay,
        int iNumberOfDailyFlightsCancelled,int iNumberOfDailyFlightsInFlightAbort,
        int iNumberOfDailyFlightsFail,int iNumberOfDailyFlightsSucceed);
    void SaveEvent(int iEventNumber,float fClockTime,int iAircraftNumber,
        int iNumberOfAircraft,int iLRIPosition,int iLRIType,int iNumberOfLRIs,
        float fEventDuration);
    void SetUpEventList(CObList& eventlist);
    void ZeroDailyFlightsCounters();
    void ZeroDailyStatesCounters();
    void ZeroRunCounters();
```

```
    void ZeroSimulationCounters();
// Event Operations
public:
    void EventEndRun(CObList& aircraftlist,CObList& delayedflightlist,
        CObList& eventlist,CObList& randomnumberlist,CObList& runresultlist,
        CObList& stocklist);
    void EventInFlightFailure(CObList& aircraftlist,CObList& eventlist,
        CObList& stocklist);
    void EventLanding(CObList& aircraftlist,CObList& eventlist,CObList& stocklist);
    void EventLRIArrival(CObList& aircraftlist,CObList& eventlist,
        CObList& randomnumberlist,CObList& stocklist);
    void EventLRIRemovalComplete(CObList& aircraftlist,CObList& eventlist,
        CObList& randomnumberlist,CObList& stocklist);
    void EventLRIRepairComplete(CObList& aircraftlist,CObList& eventlist,
        CObList& stocklist);
    void EventLRIReplacementComplete(CObList& aircraftlist,CObList& eventlist,
        CObList& randomnumberlist,CObList& stocklist);
    void EventMissionRequired(CObList& aircraftlist,CObList& delayedflightlist,
        CObList& eventlist,CObList& stocklist);
    void EventNewDay(CObList& aircraftlist,CObList& eventlist,CObList& runresultlist,
        CObList& stocklist,bool bEndRun);
    void EventPreFlightServicingComplete(CObList& aircraftlist,
        CObList& delayedflightlist, CObList& eventlist,CObList& stocklist);
    void EventPreFlightServicingStart(CObList& eventlist);
    void EventTakeOff(CObList& aircraftlist,CObList& eventlist,CObList& stocklist,
        int iAircraftNumber,int iEventNumber);
    // Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CALSSimDoc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    //}}AFX_VIRTUAL
// Implementation
public:
    virtual ~CALSSimDoc();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
// Generated message map functions
protected:
    //{{AFX_MSG(CALSSimDoc)
    afx_msg void OnSimulationRun();
    afx_msg void OnUpdateSimulationRun(CCmdUI* pCmdUI);
    afx_msg void OnFileOpenSimulationParameters();
    afx_msg void OnUpdateFileOpenSimulationParameters(CCmdUI* pCmdUI);
    afx_msg void OnFileOpenFlyingProgramme();
    afx_msg void OnUpdateFileOpenFlyingProgramme(CCmdUI* pCmdUI);
    afx_msg void OnFileOpenLRIDataFile();
    afx_msg void OnUpdateFileOpenLRIDataFile(CCmdUI* pCmdUI);
```

```
    afx_msg void OnFileOpenLRIStockFile();
    afx_msg void OnUpdateFileOpenLRIStockFile(CCmdUI* pCmdUI);
    afx_msg void OnFileOpenResultsFile();
    afx_msg void OnUpdateFileOpenResultsFile(CCmdUI* pCmdUI);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the
previous line.
#endif //
!defined(AFX_ALSSIMDOC_H__AC08B32B_CF11_11D1_A97A_44455354616F__IN
CLUDED_)
// ALSSimDoc.cpp : implementation of the CALSSimDoc class
#include "stdafx.h"
#include "ALSSim.h"
#include "ALSSimDoc.h"
#include "ALSSimView.h"
#include "mainfrm.h"
//Include file handling and Maths classes
#include "fstream.h"
#include "math.h"
// Include the Dialog class
#include "DlgSimulationFinished.h"
#include "DlgClearInputParameters.h"
// Include the Simulation Progress Bar class
#include "SBarSimProgress.h"
// Include the list classes
#include "Aircraft.h"
#include "DelayedFlight.h"
#include "Event.h"
#include "RandomNumber.h"
#include "DailyResults.h"
#include "Stock.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
// CALSSimDoc
IMPLEMENT_DYNCREATE(CALSSimDoc, CDocument)
BEGIN_MESSAGE_MAP(CALSSimDoc, CDocument)
    //{{AFX_MSG_MAP(CALSSimDoc)
    ON_COMMAND(ID_SIMULATION_RUN, OnSimulationRun)
    ON_UPDATE_COMMAND_UI(ID_SIMULATION_RUN,
        OnUpdateSimulationRun)
    ON_COMMAND(ID_FILE_OPEN_SIMULATIONPARAMETERS,
        OnFileOpenSimulationParameters)
    ON_UPDATE_COMMAND_UI(ID_FILE_OPEN_SIMULATIONPARAMETERS,
        OnUpdateFileOpenSimulationParameters)
    ON_COMMAND(ID_FILE_OPEN_FLYINGPROGRAMME,
```

```
        OnFileOpenFlyingProgramme)
    ON_UPDATE_COMMAND_UI(ID_FILE_OPEN_FLYINGPROGRAMME,
        OnUpdateFileOpenFlyingProgramme)
    ON_COMMAND(ID_FILE_OPEN_LRIDATAFILE, OnFileOpenLRIDataFile)
    ON_UPDATE_COMMAND_UI(ID_FILE_OPEN_LRIDATAFILE,
        OnUpdateFileOpenLRIDataFile)
    ON_COMMAND(ID_FILE_OPEN_LRISTOCKFILE, OnFileOpenLRIStockFile)
    ON_UPDATE_COMMAND_UI(ID_FILE_OPEN_LRISTOCKFILE,
        OnUpdateFileOpenLRIStockFile)
    ON_COMMAND(ID_FILE_OPEN_RESULTSFILE, OnFileOpenResultsFile)
    ON_UPDATE_COMMAND_UI(ID_FILE_OPEN_RESULTSFILE,
        OnUpdateFileOpenResultsFile)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
// CALSSimDoc construction/destruction
CALSSimDoc::CALSSimDoc()
{
}
CALSSimDoc::~CALSSimDoc()
{
}
BOOL CALSSimDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;
    bSimulationParametersFileLoaded = false;
    bFlyingProgrammeFileLoaded = false;
    bLRIDataFileLoaded = false;
    bLRIStockFileLoaded = false;
    bResultsFileLoaded = false;
    return TRUE;
}
// CALSSimDoc serialization
void CALSSimDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
    }
    else
    {
    }
}
// CALSSimDoc diagnostics
#ifdef _DEBUG
void CALSSimDoc::AssertValid() const
{
    CDocument::AssertValid();
}
void CALSSimDoc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
```

```
}
#endif //_DEBUG
// CALSSimDoc Main Simulation control Command
void CALSSimDoc::OnSimulationRun()
{
    //Zero simulation Counters
    ZeroSimulationCounters();
    // Create the Object lists
    CObList aircraftlist;
    CObList delayedflightlist;
    CObList eventlist;
    CObList randomnumberlist;
    CObList dailyresultslist;
    CObList stocklist;
    // Open the Events storage file
    ofstream EventOF("Event.txt");
    // load in objects and set them up
    LoadSimulationScenarioDetails();
    // Create the progress control for the simulation
    CSimProgressStatusBar* pSimStatus = CALSSimApp::GetApp()->
        GetMainFrame()->GetStatusBar();
    if(pSimStatus)
    {
        CString Label;
        Label.LoadString(IDS_SIMULATIONPROGRESS);
        pSimStatus->SetSimProgressLabel(Label);
        // Flip the status bar to progress mode
        pSimStatus->ShowSimProgressDisplay(true);
        CProgressCtrl* pSimProgress = pSimStatus->GetProgressCtrl();
        if (pSimProgress)
        {
            pSimProgress->SetRange(0,iNumberOfRuns);
            pSimProgress->SetStep(1);
        }
    }
    // Create the Random Number Streams
    CreateRandomNumberStreams(randomnumberlist);
    //create the aircraft objects including fitted LRIs and stocks
    CreateAircraft(aircraftlist, randomnumberlist, stocklist);
    // Run the Simulation
    for (iRunNumber = 1;iRunNumber <= iNumberOfRuns;iRunNumber++)
    {
        // Update the progress bar
        CProgressCtrl* pSimProgress = pSimStatus->GetProgressCtrl();
        if (pSimProgress)
        {
            pSimProgress->StepIt();
        }
        // Reset End Run Boolean Variable
        bEndRun = false;
        // Reset Normal Number Calculated Boolean Variable
```

```
bSecondNormalAvailable = false;
//Reset the Number Of Aircraft Available Counter
iNumberOfAircraftAvailable = iStartNumberOfAircraft;
// Zero run counters
ZeroRunCounters();
// Zero the Daily Flight Achievement Counters
ZeroDailyFlightsCounters();
// Set Store Events switch for the first run
if (iRunNumber == 1)
    bStoringEvents = true;
else
    bStoringEvents = false;
// Populate the event list
SetUpEventList(eventlist);
// Zero Day Number
iDayNumber = 0;
// Set Event number to ensure that simulation runs
iEventNumber = 1;
while (iEventNumber != 0)
{
    // Run the simulation event controller
    iEventNumber = GetNextEvent(aircraftlist, delayedflightlist, eventlist,
        randomnumberlist,
    dailyresultslist, stocklist);
    //Action the event
    switch (iEventNumber)
    {
    case 0:
        EventEndRun(aircraftlist,delayedflightlist,eventlist,randomnumberlist,
            dailyresultslist,stocklist);
        break;
    case 1:
        EventNewDay(aircraftlist,eventlist,dailyresultslist,stocklist,bEndRun);
        break;
    case 2:
        EventMissionRequired(aircraftlist,delayedflightlist,eventlist,stocklist);
        break;
    case 3:
        EventLanding(aircraftlist,eventlist,stocklist);
        break;
    case 4:
        EventPreFlightServicingComplete(aircraftlist,delayedflightlist,eventlist,
            stocklist);
        break;
    case 5:
        EventLRIRemovalComplete(aircraftlist,eventlist,randomnumberlist,
            stocklist);
        break;
    case 6:
        EventLRIReplacementComplete(aircraftlist,eventlist,randomnumberlist,
            stocklist);
```

```
                break;
            case 7:
                EventLRIRepairComplete(aircraftlist,eventlist,stocklist);
                break;
            case 8:
                EventLRIArrival(aircraftlist,eventlist,randomnumberlist,stocklist);
                break;
            case 11:
                EventInFlightFailure(aircraftlist,eventlist,stocklist);
                break;
            }
        }
    }
    // End of simulation
    // Turn Off the progress bar
    pSimStatus->ShowSimProgressDisplay(false);
    EndSimulation(aircraftlist,dailyresultslist,randomnumberlist,stocklist);
}
// CALSSimDoc Simulation Controller
int CALSSimDoc::GetNextEvent(CObList& aircraftlist, CObList& delayedflightlist,
    CObList& eventlist, CObList& randomnumberlist, CObList& dailyresultslist,
    CObList& stocklist)
{
    //Get next event to be actioned
    CEvent* pEvent;
    POSITION EventPos = eventlist.GetHeadPosition();
    pEvent = (CEvent*)eventlist.GetAt(EventPos);
    iEventNumber = pEvent->GetEventNumber();
    iAircraftNumber = pEvent->GetAircraftNumber();
    iNumberOfAircraft = pEvent->GetNumberOfAircraft();
    iLRIPosition = pEvent->GetLRIPosition();
    iLRIType = pEvent->GetLRIType();
    iNumberOfLRIs = pEvent->GetNumberOfLRIs();
    fClockTime = pEvent->GetEventTime();
    fPreviousEventTime = pEvent->GetPreviousEventTime();
    fEventDuration = pEvent->GetEventDuration();
    //Release the memory occupied by the event object
    delete eventlist.GetNext(EventPos);
    eventlist.RemoveHead();
    // Return the event Number
    return iEventNumber;
}
// CALSSimDoc Events
// Event 0 - Event End Run
void CALSSimDoc::EventEndRun(CObList& aircraftlist,CObList& delayedflightlist,
    CObList& eventlist, CObList& randomnumberlist,CObList& dailyresultslist,
    CObList& stocklist)
{
    // Set end run boolean variable
    bEndRun = true;
    // list control variables
```

```
CAircraft* pAircraft;
POSITION AircraftPos;
CDelayedFlight* pDelayedFlight;
CRandomNumber* pRandomNumber;
POSITION RandomNumberPos;
CStock* pStock;
POSITION StockPos;
// If storing events, save the End Run Event to the events filefile
if (bStoringEvents)
{
    iAircraftNumber = 0;
    iLRIPosition = 0;
    iLRIType = 0;
    SaveEvent(iEventNumber,fClockTime,iAircraftNumber,iNumberOfAircraft,
        iLRIPosition,iLRIType,iNumberOfLRIs,fEventDuration);
}
// Get all outstanding flights and cancel them
iNumberOfFlightsCancelled += iNumberOfFlightsDelayed;
iNumberOfDailyFlightsCancelled += iNumberOfFlightsDelayed;
// Call event new day to store aircraft states and daily flight achievements
EventNewDay(aircraftlist,eventlist,dailyresultslist,stocklist,bEndRun);
// Save Flight counters for this run
// Total take offs this run
iNumberOfFlightsTakeOff = iNumberOfFlightsOnTime +
    iNumberOfFlightsFirstHalfDelay + iNumberOfFlightsSecondHalfDelay;
// The totals of the run results so far
iTotalFlightsTasked += iNumberOfFlightsTasked;
iTotalFlightsOnTime += iNumberOfFlightsOnTime;
iTotalFlightsFirstHalfDelay += iNumberOfFlightsFirstHalfDelay;
iTotalFlightsSecondHalfDelay += iNumberOfFlightsSecondHalfDelay;
iTotalFlightsCancelled += iNumberOfFlightsCancelled;
iTotalFlightsInFlightAbort += iNumberOfFlightsInFlightAbort;
iTotalFlightsFail += iNumberOfFlightsFail;
iTotalFlightsSucceed += iNumberOfFlightsSucceed;
iTotalFlightsTakeOff += iNumberOfFlightsTakeOff;
// Squares of the results so far
// Calculate the square of the sums of the recorded results
iSquareNumberOfFlightsOnTime += pow(iNumberOfFlightsOnTime,2);
iSquareNumberOfFlightsFirstHalfDelay += pow(iNumberOfFlightsFirstHalfDelay,2);
iSquareNumberOfFlightsSecondHalfDelay +=
    pow( iNumberOfFlightsSecondHalfDelay,2);
iSquareNumberOfFlightsCancelled += pow(iNumberOfFlightsCancelled,2);
iSquareNumberOfFlightsInFlightAbort += pow(iNumberOfFlightsInFlightAbort,2);
iSquareNumberOfFlightsTakeOff += pow(iNumberOfFlightsTakeOff,2);
iSquareNumberOfFlightsFail += pow(iNumberOfFlightsFail,2);
iSquareNumberOfFlightsSucceed += pow(iNumberOfFlightsSucceed,2);
// Clear the Delayed Flights list and release the memory
POSITION DelayedFlightPos = delayedflightlist.GetHeadPosition();
// Delete the Delayed Flight objects
while (DelayedFlightPos != NULL)
{
```

```
    delete aircraftlist.GetNext(DelayedFlightPos);
}
delayedflightlist.RemoveAll();
// Calculate new seeds for the streams
pRandomNumber = (CRandomNumber*)randomnumberlist.GetHead();
pRandomNumber->SetNewRandomNumberSeed();
// Reset the LRIs fitted to the aircraft
for (StockPos = stocklist.GetHeadPosition();StockPos != NULL;)
{
    pStock = (CStock*)stocklist.GetNext(StockPos);
    pStock->ResetStock();
    iLRIType = pStock->GetStockType();
    fLRIFailureTime = pStock->GetMeanFailureTime();
    for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
    {
        if (iLRIType == 1)
            bFirstTime = true;
        else
            bFirstTime = false;
        pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
        // Get Failure Time For LRU
        while ( pAircraft->GetLRIType(bFirstTime) == iLRIType)
        {
            // action on failure distribution used
            switch(iFailureDistributionUsed)
            {
            case 0:  // none
                fLRIFailureTime = fReliabilityFactor * fLRIMeanFailureTime;
                break;
            case 1:  // exponential
                fLRIFailureTime = fReliabilityFactor *
                    GetExponentialResult( randomnumberlist,
                    iRandomNumberStream,fLRIMeanFailureTime);
                break;
            case 2:  // Lognormal
                fLRIFailureTime = fReliabilityFactor *
                    GetLognormalResult(randomnumberlist,iRandomNumberStream,
                    fLRIMeanFailureTime,fFailureVariance);
                break;
            case 3:  // Normal
                // Return is bounded to ensure no negative values
                // Max, Min and Variance read in from Simulation parameters File
                fLRIFailureTime = fReliabilityFactor *
                    GetBoundedNormal(randomnumberlist,iRandomNumberStream,
                    fLRIMeanFailureTime,fFailureMin,fFailureMax,fFailureVariance);
                break;
            case 4:  //  Triangular
                fLRIFailureTime = fReliabilityFactor *
                    GetTriangularResult(randomnumberlist,iRandomNumberStream,
                    fLRIMeanFailureTime,fFailureMin,fFailureMax,iFailureDivisor);
                break;
```

```
                case 5:  // Weibull
                    fLRIFailureTime = fReliabilityFactor *
                        GetWeibullResult(randomnumberlist,iRandomNumberStream,
                        fLRIMeanFailureTime,fFailureShape,fFailureGamma);
                    break;
                }
            // Reset LRI with new failure time for next run
                bLastLRI = pAircraft->ResetLRI(fLRIFailureTime);
                if (bLastLRI)
                    break;
                bFirstTime = false;
            }
        }
    }
    // Reset the aircraft
    for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos!= NULL;)
    {
        pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
        pAircraft->SetEndOfRunTotals(fClockTime);
        pAircraft->ResetAircraft();
    }
    // Reset the number of Pre-flight servicing teams
    iNumberOfPreFlightServicingTeams = iStartRunNumberOfPreFlightServicingTeams;
}
// Event 11 - Event In Flight Failure
void CALSSimDoc::EventInFlightFailure(CObList& aircraftlist,CObList& eventlist,
    CObList& stocklist)
{
    CAircraft* pAircraft;
    POSITION AircraftPos;
    CStock* pStock;
    POSITION StockPos;
    // If storing events, save the In Flight Failure Event to the events file
    if (bStoringEvents)
    {
        SaveEvent(iEventNumber,fClockTime,iAircraftNumber,iNumberOfAircraft,
                iLRIPosition,iLRIType,iNumberOfLRIs,fEventDuration);
    }
    // Get Aircraft pointer
    for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
    {
        pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
        if (pAircraft->GetAircraftNumber() == iAircraftNumber)
        {
            // Change LRI state to unserviceable fitted (State 2)
            iLRIState = 2;
            pAircraft->SetLRIState(iLRIPosition,iLRIState);
            // Check if the failure is to an essential LRI
            for (StockPos = stocklist.GetHeadPosition();StockPos != NULL;)
            {
                pStock = (CStock*)stocklist.GetNext(StockPos);
```

```
if (pStock->GetStockType() == iLRIType)
{
    if (pStock->GetLRIEssentiality() == 1)
    {
        // Increment the number of Essential failures
        pAircraft->IncrementNumberOfEssentialLRIFailures();
        if (pAircraft->GetInFlightAbortState() == false)
        {
            // In Flight Abort will occur
            bInFlightAbortState = true;
            pAircraft->SetInFlightAbortState(bInFlightAbortState);
            iNumberOfFlightsInFlightAbort++;
            iNumberOfDailyFlightsInFlightAbort++;

            if (fClockTime - fPreviousEventTime < fEventDuration *
                    fMissionSuccessPoint)
            {
                iNumberOfFlightsFail++;
                iNumberOfDailyFlightsFail++;
            }
            else
            {
                iNumberOfFlightsSucceed++;
                iNumberOfDailyFlightsSucceed++;
            }
            if (fClockTime - fPreviousEventTime < fEventDuration / 2)
            {
                // landing time = current time + flight time
                feEventTime = fClockTime + (fClockTime -
                    fPreviousEventTime);
                feEventDuration = 2 * (fClockTime -
                    fPreviousEventTime);
                // Insert landing event into the event queue
                ieEventNumber = 3;
                ieAircraftNumber = iAircraftNumber;
                ieNumberOfAircraft = 1;
                ieLRIPosition = 0;
                ieLRIType = 0;
                ieNumberOfLRIs = 0;
                fePreviousEventTime = fClockTime;
                InsertEvent(eventlist,ieEventNumber,feEventTime,
                    fePreviousEventTime,ieAircraftNumber,
                    ieNumberOfAircraft,ieLRIPosition,ieLRIType,
                    ieNumberOfLRIs,feEventDuration);
            }
        }
        break;
    }
    else
    {
        //Increment the number of non essential failures
```

```
                    pAircraft->IncrementNumberOfNonEssentialLRIFailures();
                }

            }
        }
        break;
    }
  }
}
//Event 3 - Event Landing
void CALSSimDoc::EventLanding(CObList& aircraftlist,CObList& eventlist,
    CObList& stocklist)
{
    // list control variables
    CAircraft* pAircraft;
    POSITION AircraftPos;
    CEvent* pEvent;
    POSITION EventPos1,EventPos2;
    CStock* pStock;
    POSITION StockPos;
    for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
    {
        pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
        if (pAircraft->GetAircraftNumber() == iAircraftNumber)
        {
        // If landing as a result of an in-flight Abort
            if (pAircraft->GetInFlightAbortState() == true)
            {
                // Get the planned landing time
                fAircraftPlannedLandingTime = pAircraft-
                    >GetAircraftPlannedLandingTime();
                for (EventPos1 = eventlist.GetHeadPosition();EventPos1 != NULL;)
                {
                    EventPos2 = EventPos1;
                    pEvent = (CEvent*)eventlist.GetNext(EventPos1);
                    if (pEvent->GetAircraftNumber() == iAircraftNumber
                        && pEvent->GetEventTime() <= fAircraftPlannedLandingTime)
                    {
                        eventlist.RemoveAt(EventPos2);
                    }
                    else
                    {
                        if (pEvent->GetEventTime() > fAircraftPlannedLandingTime)
                        {
                            break;
                        }
                    }
                }
            }
            //Update the aircraft flying hours and ascertain its serviceability state
            pAircraft->UpdateAircraftOnLanding(fClockTime,fEventDuration,
```

```
    iMaximumFailuresNonEssentialLRIs);
// If storing events, save the Landing Event to the events filefile
if (bStoringEvents)
{
    fEventDuration = 0;
    SaveEvent(iEventNumber,fClockTime,iAircraftNumber,
        iNumberOfAircraft,iLRIPosition,  iLRIType,iNumberOfLRIs,
        fEventDuration);
}
// All fitted LRIs serviceable
if (pAircraft->GetAircraftState() == 4)
{
    // And there is a pre-flight servicing team available
    if (iNumberOfPreFlightServicingTeams > 0)
    {
        EventPreFlightServicingStart(eventlist);
    }
    // otherwise change aircraft state to awaiting flight servicing(5)
    else
    {
        iAircraftState = 5;
        pAircraft->SetAircraftState(iAircraftState);
    }
}
// One or more LRIs unserviceable
else
{
    iNumberOfLRIsUnserviceable = pAircraft-
        >GetNumberOfLRIsUnserviceable();
    // Set boolean variable to ensure search starts at head of list
    bFirstTime = true;
    while (iNumberOfLRIsUnserviceable > 0)
    {
        iLRIPosition = pAircraft->RemoveUnserviceableLRIs(bFirstTime);
        // Get the LRI Type as well
        bFirstTime = false;
        if (iLRIPosition > 0)
        {
            iLRIType = pAircraft->GetUnserviceableLRIType();
            // Get duration of LRI removal from station stock file
            for (StockPos = stocklist.GetHeadPosition();StockPos != NULL;)
            {
                pStock = (CStock*)stocklist.GetNext(StockPos);
                iStockType = pStock->GetStockType();
                if (iStockType == iLRIType)
                {
                    fEventDuration = pStock->GetRemovalTime();
                    break;
                }
            }
            //Insert LRI Removal complete event for this LRI into event queue
```

```
                        ieEventNumber = 5;
                        ieAircraftNumber = iAircraftNumber;
                        ieNumberOfAircraft = 1;
                        ieLRIPosition = iLRIPosition;
                        ieLRIType = iLRIType;
                        ieNumberOfLRIs = 0;
                        feEventDuration = fEventDuration;
                        feEventTime = fClockTime +feEventDuration;
                        fePreviousEventTime = fClockTime;
                        InsertEvent(eventlist,ieEventNumber,feEventTime,
                            fePreviousEventTime,ieAircraftNumber,ieNumberOfAircraft,
                            ieLRIPosition,ieLRIType,ieNumberOfLRIs,feEventDuration);
                        // Update check controllers
                        iNumberOfLRIsUnserviceable--;
                    }
                }
            }
            break;
        }
    }
}
//Event 8 - Event LRI Arrival
void CALSSimDoc::EventLRIArrival(CObList& aircraftlist,CObList& eventlist,
    CObList& randomnumberlist,CObList& stocklist)
{
    // list control variables
    CStock* pStock;
    POSITION StockPos;
    if (bStoringEvents)
    {
        SaveEvent(iEventNumber,fClockTime,iAircraftNumber,iNumberOfAircraft,
            iLRIPosition,iLRIType,iNumberOfLRIs,fEventDuration);
    }
    // Find the LRI Type that has been delivered
    for (StockPos = stocklist.GetHeadPosition();StockPos != NULL;)
    {
        pStock = (CStock*)stocklist.GetNext(StockPos);
        if (pStock->GetStockType() == iLRIType)
        {
            // Arrival at the unit
            if (iAircraftNumber == 0)
            {
                iCount = iNumberOfLRIs;
                for (iCount;iCount > 0;iCount--)
                {
                    // LRI Required for an aircraft
                    if (pStock->GetNumberRequiredForAircraft() > 0)
                    {
                        pStock->ReduceNumberRequiredForAircraft();
                        AllocateLRIToAircraft(aircraftlist,eventlist,stocklist,iLRIType);
                    }
```

```
            // Otherwise increment the units serviceable stock
            else
            {
                iNumberOfLRIs = iCount;
                pStock->IncreaseUnitServiceableStock(iNumberOfLRIs);
                break;
            }
        }
        break;
    }
    // Arrival at the Depot
    else
    {
        // Calculate the repair time
        iCount = iNumberOfLRIs;
        for (iCount;iCount > 0;iCount--)
        {
            // Get Repair Time for Depot repair of LRI and
            // insert a LRI Repair complete event into the queue
            ieEventNumber = 7;
            ieAircraftNumber = iAircraftNumber;
            ieNumberOfAircraft = 0;
            ieLRIType = iLRIType;
            ieLRIPosition = 0;
            ieNumberOfLRIs = 1;
            fLRIMeanRepairTime = pStock->GetDepotRepairTime();
            // Add repair time randomness if appropriate
            // Switch depending on distribution used
            iRandomNumberStream = 2;
            // action on repair distribution used
            switch (iRepairDistributionUsed)
            {
            case 0: // none
                feEventDuration = fReliabilityFactor * fLRIMeanRepairTime;
                break;
            case 1: // exponential
                feEventDuration = fReliabilityFactor *
                    GetExponentialResult(randomnumberlist,
                    iRandomNumberStream,fLRIMeanRepairTime);
                break;
            case 2:  // Lognormal
                feEventDuration = fReliabilityFactor *
                    GetLognormalResult(randomnumberlist,
                    iRandomNumberStream,fLRIMeanRepairTime,
                    fRepairVariance);
                break;
            case 3:  // Normal
                // Return is bounded to ensure no negative values
                // Max, Min and Variance read in from Simulation parameters File
                feEventDuration = fReliabilityFactor *
                    GetBoundedNormal(randomnumberlist,
```

```
                        iRandomNumberStream,fLRIMeanRepairTime,fRepairMin,
                        fRepairMax,fRepairVariance);
                    break;
                case 4:  // Triangular
                    feEventDuration = fReliabilityFactor *
                        GetTriangularResult(randomnumberlist,
                        iRandomNumberStream,fLRIMeanRepairTime,fRepairMin,
                        fRepairMax,iRepairDivisor);
                    break;
                case 5:  // Weibull
                    feEventDuration = fReliabilityFactor *
                        GetWeibullResult(randomnumberlist,iRandomNumberStream,
                        fLRIMeanRepairTime,fRepairShape,fRepairGamma);
                    break;
                }
                // Check if the LRI has a spurious fault
                if (fNoFaultFoundAtDepotFactor > 0)
                {
                    iRandomNumberStream = 5;
                    // get random number
                    fRandomNumber = GetRandomNumber(randomnumberlist,
                        iRandomNumberStream);
                    // Depot repair time = test time = 0.2 * repair time
                    if (fRandomNumber <= fNoFaultFoundAtDepotFactor)
                    {
                        feEventDuration *= 0.2;
                    }
                }
                feEventTime = fClockTime +feEventDuration;
                fePreviousEventTime = fClockTime;
                InsertEvent(eventlist,ieEventNumber,feEventTime,
                    fePreviousEventTime,ieAircraftNumber,ieNumberOfAircraft,
                    ieLRIPosition,ieLRIType,ieNumberOfLRIs,feEventDuration);
            }
            break;
        }
    }
}
}
//Event 5 - Event LRI Removal Complete
void CALSSimDoc::EventLRIRemovalComplete(CObList& aircraftlist,
    CObList& eventlist,CObList& randomnumberlist,CObList& stocklist)
{
    // list control variables
    CAircraft* pAircraft;
    POSITION AircraftPos;
    CStock* pStock;
    POSITION StockPos;
    if (bStoringEvents)
    {
        SaveEvent(iEventNumber,fClockTime,iAircraftNumber,iNumberOfAircraft,
```

```
        iLRIPosition,iLRIType,iNumberOfLRIs,fEventDuration);
}
for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
{
    pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
    if (pAircraft->GetAircraftNumber() == iAircraftNumber)
    {
        // Action repair and replacement if spare available
        for (StockPos = stocklist.GetHeadPosition();StockPos != NULL;)
        {
            pStock = (CStock*)stocklist.GetNext(StockPos);
            if (pStock->GetStockType() == iLRIType)
            {
                // Get Repair location
                // get random number
                iRandomNumberStream = 3;
                fLRIProportionRepairedAtUnit =
                    GetRandomNumber(randomnumberlist,iRandomNumberStream);
                //send random number to pStock
                bLRIRepairOnUnit = (pStock->
                    LRIRepairedOnUnit(fLRIProportionRepairedAtUnit));
                if (bLRIRepairOnUnit)
                {
                    // Get Repair Time for Unit repair of LRI and insert a LRI Repair
                    // complete event into the queue
                    ieEventNumber = 7;
                    ieAircraftNumber = 0;
                    ieNumberOfAircraft = 0;
                    ieLRIPosition = 0;
                    ieLRIType = iLRIType;
                    ieNumberOfLRIs = 1;
                    fLRIMeanRepairTime = pStock->GetUnitRepairTime();
                    // add repair time randomness if appropriate
                    iRandomNumberStream = 2;
                    // action on repair distribution used
                    switch (iRepairDistributionUsed)
                    {
                        case 0: // none
                            feEventDuration = fReliabilityFactor *
                                fLRIMeanRepairTime;
                            break;
                        case 1: // exponential
                            feEventDuration = fReliabilityFactor *
                                GetExponentialResult(randomnumberlist,
                                iRandomNumberStream,fLRIMeanRepairTime);
                            break;
                        case 2:  // Lognormal
                                feEventDuration = fReliabilityFactor *
                                GetLognormalResult(randomnumberlist,
                                iRandomNumberStream,fLRIMeanRepairTime,
                                fRepairVariance);
```

```
                break;
            case 3:  // Normal
                // Return is bounded to ensure no negative values
                // Max, Minand Variance read in from Simulation
                // parameters File
                feEventDuration = fReliabilityFactor *
                    GetBoundedNormal(randomnumberlist,
                    iRandomNumberStream,fLRIMeanRepairTime,
                    fRepairMin,fRepairMax,fRepairVariance);
                break;
            case 4:  //  Triangular
                feEventDuration = fReliabilityFactor *
                    GetTriangularResult(randomnumberlist,
                    iRandomNumberStream,fLRIMeanRepairTime,
                    fRepairMin,fRepairMax,iRepairDivisor);
                break;
            case 5:  // Weibull
                feEventDuration = fReliabilityFactor *
                    GetWeibullResult(randomnumberlist,
                    iRandomNumberStream,fLRIMeanRepairTime,
                    fRepairShape,fRepairGamma);
                break;
        }
        // check for no fault found
        if (fNoFaultFoundAtUnitFactor > 0)
        {
            iRandomNumberStream = 4;
            // get random number
            fRandomNumber = GetRandomNumber(randomnumberlist,
                iRandomNumberStream);
            // if random number < fNoFaultFoundAtUnit Factor
            // Unit repair time = test time = .2 * repair time
            if (fRandomNumber <= fNoFaultFoundAtUnitFactor)
            {
                feEventDuration *= 0.2;
            }
        }
        feEventTime = fClockTime +feEventDuration;
        fePreviousEventTime = fClockTime;
        InsertEvent(eventlist,ieEventNumber,feEventTime,
            fePreviousEventTime,ieAircraftNumber,ieNumberOfAircraft,
            ieLRIPosition,ieLRIType,ieNumberOfLRIs,feEventDuration);
    }
    else
    {
        // Increment the number of unserviceable LRIs in stock
        pStock->IncreaseUnitUnserviceableStock();
    }
    // If stock is available at the unit to replace removed LRI
    if (pStock->GetUnitServiceableStock() > 0)
    {
```

```
                    // Reduce Serviceable Stock
                    pStock->ReduceUnitServiceableStock();
                    // Change LRI state to being fitted(5)
                    iLRIState = 5;
                    pAircraft->UpdateRemovedLRI(iLRIPosition,iLRIState);
                    // If LRI being replaced call lru replacement event
                    fEventDuration = pStock->GetReplacementTime();
                    // Insert a LRI replacement complete event into the event queue
                    ieEventNumber = 6;
                    ieAircraftNumber = iAircraftNumber;
                    ieNumberOfAircraft = 1;
                    ieLRIPosition = iLRIPosition;
                    ieLRIType = iLRIType;
                    ieNumberOfLRIs = 0;
                    feEventDuration = fEventDuration;
                    feEventTime = fClockTime +feEventDuration;
                    fePreviousEventTime = fClockTime;
                    InsertEvent(eventlist,ieEventNumber,feEventTime,
                        fePreviousEventTime,ieAircraftNumber,ieNumberOfAircraft,
                        ieLRIPosition,ieLRIType,ieNumberOfLRIs,feEventDuration);
                    break;
                }
                // No stock is available at the unit to replace removed LRI
                else
                {
                    // Change LRI State to removed and increment number of LRIs
                    // required for aircraft
                    iLRIState = 4;
                    pAircraft->UpdateRemovedLRI(iLRIPosition,iLRIState);
                    // Increment Number of spares of this type required for aircraft
                    pStock->IncreaseNumberRequiredForAircraft();
                    // Check if a spare available at the depot
                    if (pStock->GetDepotServiceableStock() > 0)
                    {
                        // Reduce Serviceable Stock
                        pStock->ReduceDepotServiceableStock();
                        // Increment Allocated Stock
                        pStock->IncreaseDepotAllocatedStock();
                    }
                    break;
                }
            }
        }
        break;
    }
  }
}
//Event 7 - Event LRI Repair Complete
void CALSSimDoc::EventLRIRepairComplete(CObList& aircraftlist,CObList& eventlist,
    CObList& stocklist)
{
```

```
        // list control variables
        CStock* pStock;
        POSITION StockPos;
        iLRIPosition = 0;
        if (bStoringEvents)
        {
            SaveEvent(iEventNumber,fClockTime,iAircraftNumber,iNumberOfAircraft,
                iLRIPosition,iLRIType,iNumberOfLRIs,fEventDuration);
        }
        for (StockPos = stocklist.GetHeadPosition();StockPos != NULL;)
        {
            pStock = (CStock*)stocklist.GetNext(StockPos);
            if (pStock->GetStockType() == iLRIType)
            {
                // LRI Repaired at the unit
                if (iAircraftNumber == 0)
                {
                    // If LRIs of this type required for aircraft
                    if (pStock->GetNumberRequiredForAircraft() > 0)
                    {
                        pStock->ReduceNumberRequiredForAircraft();
                        AllocateLRIToAircraft(aircraftlist,eventlist,stocklist,iLRIType);
                        break;
                    }
                    // Otherwise increase serviceable stock of this type
                    else
                    {
                        iNumberOfLRIs = 1;
                        pStock->IncreaseUnitServiceableStock(iNumberOfLRIs);
                        break;
                    }
                }
                // LRI Repaired at the Depot
                else
                {
                    // LRI required at the unit
                    if (pStock->MoreLRIsRequiredAtUnit() ||
                        (pStock->GetNumberRequiredForAircraft() > 0))
                    {
                        pStock->IncreaseDepotAllocatedStock();
                    }
                    // LRI not required at the unit
                    else
                    {
                        pStock->IncreaseDepotServiceableStock();
                        break;
                    }
                }
            }
        }
    }
```

```
//Event 6 - Event LRI Replacement Complete
void CALSSimDoc::EventLRIReplacementComplete(CObList& aircraftlist,
    CObList& eventlist,CObList& randomnumberlist,CObList& stocklist)
{
    // list control variables
    CAircraft* pAircraft;
    POSITION AircraftPos;
    CStock* pStock;
    POSITION StockPos;
    if (bStoringEvents)
    {
        SaveEvent(iEventNumber,fClockTime,iAircraftNumber,iNumberOfAircraft,
            iLRIPosition,iLRIType,iNumberOfLRIs,fEventDuration);
    }
    // Get mean time to failure for this type of LRI
    for (StockPos = stocklist.GetHeadPosition();StockPos != NULL;)
    {
        pStock = (CStock*)stocklist.GetNext(StockPos);
        if (pStock->GetStockType() == iLRIType)
        {
            fLRIMeanFailureTime = pStock->GetMeanFailureTime();
            break;
        }
    }
    // Calculate failure time for this LRI
    iRandomNumberStream = 1;
    // action on failure distribution used
    switch (iFailureDistributionUsed)
    {
        case 0:  // none
            fLRIFailureTime = fReliabilityFactor * fLRIMeanFailureTime;
            break;
        case 1:  // Exponential
                fLRIFailureTime = fReliabilityFactor *
                GetExponentialResult(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime);
            break;
        case 2:  // Lognormal
            fLRIFailureTime = fReliabilityFactor *
                GetLognormalResult(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime,fFailureVariance);
            break;
        case 3:  // Normal
            // Return is bounded to ensure no negative values
            // Max, Min and Variance read in from Simulation parameters File
            fLRIFailureTime = fReliabilityFactor *
                GetBoundedNormal(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime,fFailureMin,fFailureMax,fFailureVariance);
            break;
        case 4:  //  Triangular
```

```
            fLRIFailureTime = fReliabilityFactor *
                GetTriangularResult(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime,fFailureMin,fFailureMax,iFailureDivisor);
            break;
        case 5:  // Weibull
            fLRIFailureTime = fReliabilityFactor *
                GetWeibullResult(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime,fFailureShape,fFailureGamma);
            break;
    }
    // Update the aircraft and relevant LRI
    for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
    {
        pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
        if (pAircraft->GetAircraftNumber() == iAircraftNumber)
        {
            pAircraft->UpdateReplacedLRI(iLRIPosition,fLRIFailureTime);
            //If no unserviceable LRIs fitted
            if (pAircraft->GetAircraftState() == 4)
            {
                // Update the Unserviceable time counter
                pAircraft->SetTimeUnserviceable(fClockTime);
                // And there is a pre-flight servicing team available
                if (iNumberOfPreFlightServicingTeams > 0)
                {
                    EventPreFlightServicingStart(eventlist);
                    // Decrement number of servicing teams
                }
                else
                {
                    iAircraftState = 5;
                    pAircraft->SetAircraftState(iAircraftState);
                }
            }
            break;
        }
    }
}
//Event 2 - Event Mission Required
void CALSSimDoc::EventMissionRequired(CObList& aircraftlist,
    CObList& delayedflightlist,CObList& eventlist,CObList& stocklist)
{
    // list control variables
    CAircraft* pAircraft;
    POSITION AircraftPos;
    if (bStoringEvents)
    {
        SaveEvent(iEventNumber,fClockTime,iAircraftNumber,iNumberOfAircraft,
            iLRIPosition,iLRIType,iNumberOfLRIs,fEventDuration);
    }
    // Incease Daily Missions Tasked Counter
```

```
    iNumberOfDailyFlightsTasked += iNumberOfAircraft;
    if (iNumberOfAircraftAvailable >= iNumberOfAircraft)
    {
        iNumberOfAircraftAvailable -= iNumberOfAircraft;
        for (iNumberOfAircraft;iNumberOfAircraft > 0;iNumberOfAircraft--)
        {
            // Get first aircraft available and allocate it
            for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
            {
                pAircraft =(CAircraft*)aircraftlist.GetNext(AircraftPos);
                if (pAircraft->GetAircraftState() == 1)
                {
                    iAircraftNumber = pAircraft->GetAircraftNumber();
                    EventTakeOff(aircraftlist,eventlist,stocklist,iAircraftNumber,
                        iEventNumber);
                    break;
                }
            }
        }
    }
    else
    // Action if no aircraft allocated to the flight
    {
        //increment number of delayed flights
        iNumberOfFlightsDelayed+= iNumberOfAircraft;
        //Add the flight to the list of delayed flights
        delayedflightlist.AddTail(new CDelayedFlight(fClockTime,iNumberOfAircraft,
            fMaximumFlightDelay,fEventDuration));
    }
}
//Event 1 - Event New Day
void CALSSimDoc::EventNewDay(CObList& aircraftlist,CObList& eventlist,
    CObList& dailyresultslist,CObList& stocklist,bool bEndRun)
{
    // list control variables
    CAircraft* pAircraft;
    POSITION AircraftPos;
    CDailyResults* pDailyResults;
    POSITION DailyResultsPos;
    CStock* pStock;
    POSITION StockPos;
    // Increment day number;
    iDayNumber++;
    //Zero the daily results counters
    ZeroDailyStatesCounters();
    // create the daily results linked list
    if (iRunNumber == 1)
    {
        dailyresultslist.AddTail(new CDailyResults(iDayNumber));
    }
    // If storing events, save the End Run Event to the events filefile
```

```
if (bStoringEvents)
{
    SaveEvent(iEventNumber,fClockTime,iAircraftNumber,iNumberOfAircraft,
        iLRIPosition,iLRIType,iNumberOfLRIs,fEventDuration);
}
// If not called by Event End Run, get the aircraft states and store
//values in the daily results within the run results file
if (bEndRun == false)
{
    for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
    {
        pAircraft =(CAircraft*)aircraftlist.GetNext(AircraftPos);
        iAircraftState = pAircraft->GetAircraftState();
        switch (iAircraftState)
        {
        case 1:
            iNumberOfAircraftServiceable++;
            break;
        case 2:
            iNumberOfAircraftFlying++;
            break;
        case 3:
            iNumberOfAircraftUnserviceable++;
            break;
        case 4:
            iNumberOfAircraftInPreFlightServicing++;
            break;
        case 5:
            iNumberOfAircraftAwaitingPreFlightServicing++;
            break;
        }
    }
    // store aircraft states and daily flight achievements for the day
    // Sum of all runs
    for (DailyResultsPos = dailyresultslist.GetHeadPosition();
        DailyResultsPos != NULL;)
    {
        pDailyResults =(CDailyResults*)dailyresultslist.GetNext(DailyResultsPos);
        if (pDailyResults->GetDayNumber() == iDayNumber)
        {
            pDailyResults-
                >SaveDailyAircraftStates(iNumberOfAircraftServiceable,
                iNumberOfAircraftFlying,iNumberOfAircraftUnserviceable,
                iNumberOfAircraftInPreFlightServicing,
                iNumberOfAircraftAwaitingPreFlightServicing);
            break;
        }
    }
}
if (iDayNumber > 1)
{
```

```
        SaveDailyFlyingStats(dailyresultslist,iDayNumber,
            iNumberOfDailyFlightsTasked,iNumberOfDailyFlightsOnTime,
        iNumberOfDailyFlightsFirstHalfDelay,
            iNumberOfDailyFlightsSecondHalfDelay,iNumberOfDailyFlightsCancelled,
        iNumberOfDailyFlightsInFlightAbort,iNumberOfDailyFlightsFail,
            iNumberOfDailyFlightsSucceed);
}
//  Zero the Daily Flight Achievement Counters
ZeroDailyFlightsCounters();
// Check to see if any LRIs need moving from their current location
for (StockPos = stocklist.GetHeadPosition();StockPos != NULL;)
{
    pStock = (CStock*)stocklist.GetNext(StockPos);
    // Are there any Unserviceable LRIs in Stock at the unit
    iNumberOfLRIs = pStock->GetUnitUnserviceableStock();
    if (iNumberOfLRIs > 0)
    {
        // Zero the Number of Unserviceable LRIs in Stock
        pStock->ZeroUnitUnserviceableStock();
        // Insert LRI arrival event into event list
        ieEventNumber = 8;
        ieAircraftNumber = -1;
        ieNumberOfAircraft = 0;
        ieLRIPosition = 0;
        ieLRIType = pStock->GetStockType();
        ieNumberOfLRIs = iNumberOfLRIs;
        feEventDuration = fDepotUnitTransferTime;
        feEventTime = fClockTime +feEventDuration;
        fePreviousEventTime = fClockTime;
        InsertEvent(eventlist,ieEventNumber,feEventTime,fePreviousEventTime,
            ieAircraftNumber,ieNumberOfAircraft,ieLRIPosition,ieLRIType,
            ieNumberOfLRIs,feEventDuration);
    }
    // Are there any Serviceable LRIs at the depot to be transferred to the unit
    iNumberOfLRIs = pStock->GetDepotAllocatedStock();
    if (iNumberOfLRIs > 0)
    {
        // Zero the number of LRIs at the depot awaiting allocation
        pStock->ZeroDepotAllocatedStock();
        // Insert LRI arrival event into event list
        ieEventNumber = 8;
        ieAircraftNumber = 0;
        ieNumberOfAircraft = 0;
        ieLRIPosition = 0;
        ieLRIType = pStock->GetStockType();
        ieNumberOfLRIs = iNumberOfLRIs;
        feEventDuration = fDepotUnitTransferTime;
        feEventTime = fClockTime +feEventDuration;
        fePreviousEventTime = fClockTime;
        InsertEvent(eventlist,ieEventNumber,feEventTime,fePreviousEventTime,
            ieAircraftNumber,ieNumberOfAircraft,ieLRIPosition,ieLRIType,
```

```
                ieNumberOfLRIs,feEventDuration);
        }
    }
}
//Event 4 - Event Pre Flight Servicing Complete
void CALSSimDoc::EventPreFlightServicingComplete(CObList& aircraftlist,
    CObList& delayedflightlist,CObList& eventlist,CObList& stocklist)
{
    // list control variables
    CAircraft* pAircraft;
    POSITION AircraftPos;
    CDelayedFlight* pDelayedFlight;
    POSITION DelayedFlightPos1;
    POSITION DelayedFlightPos2;
    // Increment the number of aircraft available for allocation to missions
    iNumberOfAircraftAvailable++;
    // Find the aircraft that has just had its pre-flight servicing completed
    iLRIPosition = 0;
    iLRIType = 0;
    for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
    {
        pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
        if (pAircraft->GetAircraftNumber() == iAircraftNumber)
        {
            iAircraftState = 1;
            pAircraft->SetAircraftState(iAircraftState);
            // Update the Flight Servicing time counter
            pAircraft->SetTimeInPreFlightServicing(fClockTime);
            // If storing events, save the Landing Event to the events filefile
            if (bStoringEvents)
            {
                SaveEvent(iEventNumber,fClockTime,iAircraftNumber,
                    iNumberOfAircraft,iLRIPosition,iLRIType,iNumberOfLRIs,
                    fEventDuration);
            }
            break;
        }
    }
    // If there are any delayed flights outstanding and aircraft not allocated to mission
    bAircraftAllocatedToFlight = false;
    for (DelayedFlightPos1 = delayedflightlist.GetHeadPosition();
            DelayedFlightPos1 != NULL;)
    {
        DelayedFlightPos2 = DelayedFlightPos1;
        pDelayedFlight =
            (CDelayedFlight*)delayedflightlist.GetNext(DelayedFlightPos1);
        fLastTakeOffTime = pDelayedFlight->GetLastTakeOffTime();
        iNumberOfAircraft = pDelayedFlight->GetNumberOfAircraftRequired();
        // If latest time has already passed cancel the flight
        if (fClockTime > fLastTakeOffTime)
        {
```

```
            delayedflightlist.RemoveAt(DelayedFlightPos2);
            iNumberOfFlightsCancelled+= iNumberOfAircraft;
            iNumberOfDailyFlightsCancelled+= iNumberOfAircraft;
            iNumberOfFlightsDelayed-= iNumberOfAircraft;
            // Release the memory occupied by the delayed flight object
            delete pDelayedFlight;
        }
        // If current time within acceptable delay range and sufficient aircraft available
        else
        {
            if (iNumberOfAircraftAvailable >= iNumberOfAircraft)
            {
                // Reduce the number of aircraft available
                iNumberOfAircraftAvailable -= iNumberOfAircraft;
                iNumberOfFlightsDelayed -= iNumberOfAircraft;
                fEventDuration = pDelayedFlight->GetFlightDuration();
                delayedflightlist.RemoveAt(DelayedFlightPos2);
                iNumberOfAircraft--;
                // Allocate aircraft to the mission
                bAircraftAllocatedToFlight = true;
                EventTakeOff(aircraftlist,eventlist,stocklist,iAircraftNumber,
                    iEventNumber);
                // Allocate any other aircraft required
                for (iNumberOfAircraft;iNumberOfAircraft > 0;iNumberOfAircraft--)
                {
                    // Get first aircraft available and allocate it
                    for (AircraftPos = aircraftlist.GetHeadPosition();
                        AircraftPos != NULL;)
                    {
                        pAircraft =(CAircraft*)aircraftlist.GetNext(AircraftPos);
                        if (pAircraft->GetAircraftState() == 1)
                        {
                            iAircraftNumber = pAircraft->GetAircraftNumber();
                            EventTakeOff(aircraftlist,eventlist,stocklist,iAircraftNumber,
                                iEventNumber);
                            break;
                        }
                    }
                }
                // Release the memory occupied by the delayed flight object
                delete pDelayedFlight;
            }
        }
        if (bAircraftAllocatedToFlight == true)
        {
            break;
        }
    }
    // increment the number of pre-flight servicing teams available
    iNumberOfPreFlightServicingTeams++;
    // check to see if any aircraft are awaiting pre-flight servicing
```

```
    for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
    {
        pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
        // Aircraft waiting for pre-flightservicing
        if(pAircraft->GetAircraftState() == 5)
        {
            iAircraftNumber = pAircraft->GetAircraftNumber();
            iAircraftState = 4;
            pAircraft->SetAircraftState(iAircraftState);
            // Update the awaiting Pre-Flight Servicing time counter
            pAircraft->SetTimeAwaitingPreFlightServicing(fClockTime);
            EventPreFlightServicingStart(eventlist);
            break;
        }
    }
}
//Event 10 - Event Pre Flight Servicing Start
void CALSSimDoc::EventPreFlightServicingStart(CObList& eventlist)
{
    iEventNumber = 10;
    // Decrement number of sevicing teams
    iNumberOfPreFlightServicingTeams--;
    // If storing events, save the Event to the events filefile
    if (bStoringEvents)
    {
        SaveEvent(iEventNumber,fClockTime,iAircraftNumber,iNumberOfAircraft,
            iLRIPosition,iLRIType,iNumberOfLRIs,fEventDuration);
    }
    //Insert pre-flight servicing complete event for this aircraft into the event queue
    ieEventNumber = 4;
    ieAircraftNumber = iAircraftNumber;
    ieNumberOfAircraft = 1;
    ieLRIPosition = 0;
    ieLRIType = 0;
    ieNumberOfLRIs = 0;
    feEventDuration = fPreFlightServicingDuration;
    feEventTime = fClockTime +feEventDuration;
    fePreviousEventTime = fClockTime;
    InsertEvent(eventlist,ieEventNumber,feEventTime,fePreviousEventTime,
        ieAircraftNumber,ieNumberOfAircraft,ieLRIPosition,ieLRIType,
        ieNumberOfLRIs,feEventDuration);
}
//Event 9 - Event Take Off
void CALSSimDoc::EventTakeOff(CObList& aircraftlist,CObList& eventlist,
    CObList& stocklist,int iAircraftNumber,int iEventNumber)
{
    CAircraft* pAircraft;
    POSITION AircraftPos;
    CStock* pStock;
    POSITION StockPos;
    //change aircraft state to flying
```

```
for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos !=NULL;)
{
    pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
    if (pAircraft->GetAircraftNumber() == iAircraftNumber)
    {
        iAircraftState = 2;
        pAircraft->SetAircraftState(iAircraftState);
        // Set In Flight Abort Boolean variable to false
        bInFlightAbortState = false;
        pAircraft->SetInFlightAbortState(bInFlightAbortState);
        if (iEventNumber == 2)
        {
            // Update the serviceable time counter and Change Aircraft State
            pAircraft->SetTimeServiceable(fClockTime);
            // Increment flights on time counters
            iNumberOfFlightsOnTime++;
            iNumberOfDailyFlightsOnTime++;
        }
        else
        {
            // increment the relevant flightdelay counter
            if (fLastTakeOffTime - fClockTime < fMaximumFlightDelay/2)
            {
                iNumberOfFlightsSecondHalfDelay++;
                iNumberOfDailyFlightsSecondHalfDelay++;
            }
            else
            {
                iNumberOfFlightsFirstHalfDelay++;
                iNumberOfDailyFlightsFirstHalfDelay++;
            }
        }
        bAircraftAllocatedToFlight = true;
        // If storing events, save the Take Off Event to the events filefile
        if (bStoringEvents)
        {
            iEventNumber = 9;
            iLRIPosition = 0;
            iLRIType = 0;
            SaveEvent(iEventNumber,fClockTime,iAircraftNumber,
                iNumberOfAircraft,iLRIPosition,iLRIType,iNumberOfLRIs,
                fEventDuration);
        }
        // If failure before planned landing set up in flight failure
        fAircraftFailureTime = pAircraft->GetAircraftFailureTime();
        fAircraftFlyingHours = pAircraft->GetAircraftFlyingHours();
        iNumberOfFailedEssentialLRIs = 0;
        if ( fAircraftFailureTime < fAircraftFlyingHours + fEventDuration)
        {
            bFirstTime = true;
            iLRIPosition = 1;
```

```
        while (iLRIPosition > 0)
        {
            iLRIPosition = pAircraft->
                GetLRIFailedInFlight(bFirstTime,fEventDuration);
            bFirstTime = false;
            // Get the LRI position
            if (iLRIPosition > 0)
            {
                iLRIType = pAircraft->GetUnserviceableLRIType();
                // Get LRI Essentiality
                for (StockPos = stocklist.GetHeadPosition();StockPos != NULL;)
                {
                    pStock = (CStock*)stocklist.GetNext(StockPos);
                    if (pStock->GetStockType() == iLRIType)
                    {
                        if (pStock->GetLRIEssentiality() == 1)
                        {
                            // Increment the number of essential failure
                            iNumberOfFailedEssentialLRIs++;
                            fLRIFailureTime = pAircraft->
                                GetLRIFailureTime(iLRIPosition);
                            // Insert an in flight failure event into the event queue
                            ieEventNumber = 11;
                            ieAircraftNumber = iAircraftNumber;
                            ieNumberOfAircraft = 1;
                            feEventTime = fClockTime +fLRIFailureTime -
                                fAircraftFlyingHours;
                            ieLRIPosition = iLRIPosition;
                            ieLRIType = iLRIType;
                            ieNumberOfLRIs = 1;
                            feEventDuration = fEventDuration;
                            fePreviousEventTime = fClockTime;
                            InsertEvent(eventlist,ieEventNumber,feEventTime,
                                fePreviousEventTime,ieAircraftNumber,
                                ieNumberOfAircraft,ieLRIPosition,ieLRIType,
                                ieNumberOfLRIs,feEventDuration);
                        }
                        break;
                    }
                }
            }
        }
    }
    // Check to see if essential LRIs have failed
    if (iNumberOfFailedEssentialLRIs == 0)
    {
        // The mission will be a success
        iNumberOfFlightsSucceed++;
        iNumberOfDailyFlightsSucceed++;
    }
    // Save planned mission times for landing cleanup
```

```
                pAircraft->SavePlannedLandingTime(fClockTime,fEventDuration);
                //Insert a landing event for this flight into the event queue
                ieEventNumber = 3;
                feEventTime = fClockTime +fEventDuration;
                ieAircraftNumber = iAircraftNumber;
                ieNumberOfAircraft = 1;
                ieLRIPosition = 0;
                ieLRIType = 0;
                ieNumberOfLRIs = 0;
                feEventDuration = fEventDuration;
                fePreviousEventTime = fClockTime;
                InsertEvent(eventlist,ieEventNumber,feEventTime,fePreviousEventTime,
                    ieAircraftNumber,ieNumberOfAircraft,ieLRIPosition,ieLRIType,
                    ieNumberOfLRIs,feEventDuration);
                break;
            }
        }
    }
// CALSSimDoc commands
bool CALSSimDoc::IsValidFileSpec (LPCSTR lpszFileSpec)
{
    OFSTRUCT of;
    if (OpenFile (lpszFileSpec, &of, OF_EXIST) == HFILE_ERROR)
    {
        return false;
    }
    else
    {
        return true;
    }
}
float CALSSimDoc::GetBoundedNormal(CObList& randomnumberlist,
    int iRandomNumberStream,float fLRIMeanFailTime,float fFailMin,
    float fFailMax,float fVariance)
{
    fLRIFailureMin = fLRIMeanFailTime * fFailMin;
    fLRIFailureMax = fLRIMeanFailTime * fFailMax;
    fLRIFailureTime = 0;
    while ((fLRIFailureTime < (fLRIFailureMin))||(fLRIFailureTime >
        (fLRIFailureMax)))
    {
        fLRIFailureTime = fReliabilityFactor *
            GetNormalResult(randomnumberlist,iRandomNumberStream,
            fLRIMeanFailTime,fVariance);
    }
    return fLRIFailureTime;
}
float CALSSimDoc::GetExponentialResult(CObList& randomnumberlist,
int iStreamNumber,float fMean)
{
    fRandomNumber = GetRandomNumber(randomnumberlist,iStreamNumber);
```

```
    fResult = (-fMean)*log(1 - fRandomNumber);
    return fResult;
}
float CALSSimDoc::GetLognormalResult(CObList& randomnumberlist,
    int iStreamNumber,float fMean,float fVariance)
{
    // use a Lognormal(1,fVariance) distribution
    fLogMu = 1;
    fLogVariance = fVariance;
    fLogMean = exp(fLogMu + fLogVariance/2);
    // get normal value then convert
    feEventDuration =GetNormalResult(randomnumberlist,iRandomNumberStream,
        fLogMu,fLogVariance);
    fResult = exp(feEventDuration);
    // Convert the value to take account of the actual mean
    fResult = (fResult * fMean)/fLogMean;
    return fResult;
}
float CALSSimDoc::GetNormalResult(CObList& randomnumberlist,
    int iStreamNumber,float fMean,float fVariance)
{
    // First generate a U(0,1) value
    // formula generates 2 numbers but only one is used
    bAcceptableResult = false;
    while (bAcceptableResult != true)
    {
        fRandomNumber1 = GetRandomNumber(randomnumberlist,iStreamNumber);
        fRandomNumber2 = GetRandomNumber(randomnumberlist,iStreamNumber);
        fNormalValue1 = 2*fRandomNumber1 - 1;
        fNormalValue2 = 2*fRandomNumber2 - 1;
        fNormalCheckValue = pow(fNormalValue1,2) + pow(fNormalValue2,2);
        if (fNormalCheckValue <=1)
        {
            bAcceptableResult = true;
            fNormalCalculationValue =
                sqrt((-2*log(fNormalCheckValue))/fNormalCheckValue);
            // calculate for N(1,fVariance)
            fNormalNumber = 1 + (sqrt(fVariance) * fNormalValue1 *
                fNormalCalculationValue);
            fNormalNumber = fNormalNumber * fMean;
            //   Second value not used thus not calculated or returned
        }
    }
    return fNormalNumber;
}
float CALSSimDoc::GetRandomNumber(CObList& randomnumberlist,
int iStreamNumber)
{
    // Set up pointers to the Random Number Class
    CRandomNumber* pRandomNumber;
```

```
    pRandomNumber = (CRandomNumber*)randomnumberlist.GetHead();
    fRandomNumber = pRandomNumber->GenVal(iStreamNumber);
    return fRandomNumber;
}
float CALSSimDoc::GetTriangularResult(CObList& randomnumberlist,
    int iStreamNumber,float fLRIMean,float fTriangleMin,float fTriangleMax,
    float fTriangleDiv)
{
    // shape parameters read in from file
    // mode = min +(max-min)/div
    float B,T,X;  // temporary variables used within the operation
    fTriangleRange = fTriangleMax - fTriangleMin;
    fTriangleMode =fTriangleMin + (fTriangleRange/fTriangleDiv);
    B = (fTriangleMode - fTriangleMin)/fTriangleRange;
    fRandomNumber = GetRandomNumber(randomnumberlist,iStreamNumber);
    if (fRandomNumber < B)
        T = sqrt(B * fRandomNumber);
    else
        T = 1 - sqrt((1 - B)*(1 - fRandomNumber));
    X = (fTriangleMin + (fTriangleRange*T));
    // Convert the result to take account of the LRI Mean
    fResult = (X * fLRIMean)/((fTriangleMin + fTriangleMax + fTriangleMode)/3);
    return fResult;
}
float CALSSimDoc::GetWeibullResult(CObList& randomnumberlist,int iStreamNumber,
    float fLRIMean,float fShape, float fGamma)
{
//  a = fScale;
//  b = fShape;
//  a = fLRIMean /fGamma where fGamma = gamma(1 + 1/b) entered as fixed value
    fScale = fLRIMean / fGamma;
    fRandomNumber = GetRandomNumber(randomnumberlist,iStreamNumber);
    fResult = fScale*(pow(-log(1-fRandomNumber), 1/fShape));
    return fResult;
}
/ Allocate the repaired LRI to an aircraft
void CALSSimDoc::AllocateLRIToAircraft(CObList& aircraftlist,CObList& eventlist,
    CObList& stocklist,int iLRIType)
{
    // Set up pointers to Classes
    CAircraft* pAircraft;
    POSITION AircraftPos;
    CStock* pStock;
    POSITION StockPos;
    for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos != NULL;)
    {
        pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
        // Aircraft requires LRI of some type
        if (pAircraft->GetNumberOfLRIsRequired() > 0)
        {
            // LRI of this type required on the aircraft
```

```
            if (pAircraft->CheckLRITypeRequired(iLRIType))
            {
                // Decrement the number of LRIs Required on the Aircraft
                pAircraft->ReduceNumberOfLRIsRequired();
                // Change LRI state to being fitted(5)
                iLRIState = 5;
                // Get the LRI position for the LRI to be fitted to
                iLRIPosition = pAircraft->UpdateLRIRequired(iLRIType,iLRIState);
                // Set up LRI replacement event
                for (StockPos = stocklist.GetHeadPosition();StockPos !=NULL;)
                {
                    pStock = (CStock*)stocklist.GetNext(StockPos);
                    if (pStock->GetStockType() == iLRIType)
                    {
                        fEventDuration = pStock->GetReplacementTime();
                        break;
                    }
                }
                //Insert a LRI replacement complete event into the event queue
                ieEventNumber = 6;
                ieAircraftNumber = pAircraft->GetAircraftNumber();
                ieNumberOfAircraft = 1;
                ieLRIPosition = iLRIPosition;
                ieLRIType = iLRIType;
                ieNumberOfLRIs = 0;
                feEventDuration = fEventDuration;
                feEventTime = fClockTime + feEventDuration;
                fePreviousEventTime = fClockTime;
                InsertEvent(eventlist,ieEventNumber,feEventTime,fePreviousEventTime,
                    ieAircraftNumber,ieNumberOfAircraft,ieLRIPosition,ieLRIType,
                    ieNumberOfLRIs,feEventDuration);
                break;
            }
        }
    }
}
void CALSSimDoc::CreateAircraft(CObList& aircraftlist, CObList& randomnumberlist,
    CObList& stocklist)
{
    // Set up the pointer to access the aircraft list
    CAircraft* pAircraft;
    // Create the aircraft and the fitted LRIs
    for (iAircraftNumber = 1;iAircraftNumber <= iNumberOfAircraft;
        iAircraftNumber++)
    {
        aircraftlist.AddTail(new CAircraft(iAircraftNumber));
        pAircraft = (CAircraft*)aircraftlist.GetTail();
        // Open the LRI data and LRI stock files
        ifstream LRIDataIF(sLRIDataFile);
        ifstream LRIStockIF(sLRIStockFile);
        //Read in the LRI data and build LRIs into the aircraft and LRI Stock files
```

```
LRIDataIF >> iLRIType >> iLRINumberFitted >> fLRIMeanFailureTime >>
    iLRIEssential >> fLRIRemovalTime >> fLRIRefitTime >>
    fLRIProportionRepairedAtUnit >> fLRIUnitRepairTime >>
    fLRIDepotRepairTime;
iLRIPosition = 1;
while (LRIDataIF)
{
    // Calculate LRI FailureTime
    iRandomNumberStream = 1;
    for (iLRINumberFitted;iLRINumberFitted > 0;iLRINumberFitted--)
    {
        // action on failure distribution used
        switch(iFailureDistributionUsed)
        {
        case 0:  // none
            fLRIFailureTime = fReliabilityFactor * fLRIMeanFailureTime;
            break;
        case 1:  // exponential
            fLRIFailureTime = fReliabilityFactor *
                GetExponentialResult(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime);
            break;
        case 2:  // Lognormal
            fLRIFailureTime = fReliabilityFactor *
                GetLognormalResult(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime,fFailureVariance);
            break;
        case 3:  // Normal
            // Return is bounded to ensure no negative values
            // Max, Min and Variance read in from Simulation parameters File
            fLRIFailureTime = fReliabilityFactor *
                GetBoundedNormal(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime,fFailureMin,fFailureMax,fFailureVariance);
            break;
        case 4:  //  Triangular
            fLRIFailureTime = fReliabilityFactor *
                GetTriangularResult(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime,fFailureMin,fFailureMax,iFailureDivisor);
            break;
        case 5:  // Weibull
            fLRIFailureTime = fReliabilityFactor *
                GetWeibullResult(randomnumberlist,iRandomNumberStream,
                fLRIMeanFailureTime,fFailureShape,fFailureGamma);
            break;
        }
        // Generate new LRIs within the Aircraft class and return LRI position
        iLRIPosition = pAircraft->
            AddNewLRI(iLRIPosition,iLRIType,fLRIFailureTime);
    }
    if (iAircraftNumber == 1)
    {
```

```
                LRIStockIF >> iLRIType  >> iLRIUnitStock >> iLRIDepotStock;
                stocklist.AddTail(new CStock(iLRIType,fLRIMeanFailureTime,
                    iLRIEssential,fLRIRemovalTime,fLRIRefitTime,
                    fLRIProportionRepairedAtUnit,fLRIUnitRepairTime,
                    fLRIDepotRepairTime,iLRIUnitStock,iLRIDepotStock));
            }
            LRIDataIF >> iLRIType >> iLRINumberFitted >> fLRIMeanFailureTime >>
                iLRIEssential >> fLRIRemovalTime >> fLRIRefitTime >>
                fLRIProportionRepairedAtUnit >> fLRIUnitRepairTime >>
                fLRIDepotRepairTime;
        }
    }
}
void CALSSimDoc::CreateDailyResults(CObList& dailyresultslist,int iNumberOfDays)
{
    for (iDayNumber = 1;iDayNumber <= iNumberOfDays;iDayNumber++)
    {
        dailyresultslist.AddTail(new CDailyResults(iDayNumber));
    }
}
void CALSSimDoc::CreateRandomNumberStreams(CObList& randomnumberlist)
{
        randomnumberlist.AddTail(new CRandomNumber());
}
void CALSSimDoc::EndSimulation(CObList& aircraftlist,CObList& dailyresultslist,
    CObList& randomnumberlist, CObList& stocklist)
{
    // list control variables
    CAircraft* pAircraft;
    POSITION AircraftPos;
    CDailyResults* pDailyResults;
    POSITION DailyResultsPos;
    // Open the Output file
    ofstream ResultsOF(sResultsFile);
    // Change integer counters to floating point counters
    fNumberOfFlightsCancelled = iTotalFlightsCancelled;
    fNumberOfFlightsFail = iTotalFlightsFail;
    fNumberOfFlightsFirstHalfDelay = iTotalFlightsFirstHalfDelay;
    fNumberOfFlightsInFlightAbort = iTotalFlightsInFlightAbort;
    fNumberOfFlightsOnTime = iTotalFlightsOnTime;
    fNumberOfFlightsSecondHalfDelay = iTotalFlightsSecondHalfDelay;
    fNumberOfFlightsSucceed = iTotalFlightsSucceed;
    fNumberOfFlightsTakeOff = iTotalFlightsTakeOff;
    fNumberOfFlightsTasked = iTotalFlightsTasked;
    // Calculate Means
    fMeanFlightsTasked = fNumberOfFlightsTasked/iNumberOfRuns;
    fMeanFlightsOnTime = fNumberOfFlightsOnTime/iNumberOfRuns;
    fMeanFlightsFirstHalfDelay = fNumberOfFlightsFirstHalfDelay/iNumberOfRuns;
    fMeanFlightsSecondHalfDelay =
        fNumberOfFlightsSecondHalfDelay/iNumberOfRuns;
    fMeanFlightsCancelled = fNumberOfFlightsCancelled/iNumberOfRuns;
```

```
fMeanFlightsTakeOff = fMeanFlightsOnTime + fMeanFlightsFirstHalfDelay +
    fMeanFlightsSecondHalfDelay;
fMeanFlightsInFlightAbort = fNumberOfFlightsInFlightAbort/iNumberOfRuns;
fMeanFlightsFail = fNumberOfFlightsFail/iNumberOfRuns;
fMeanFlightsSucceed = fNumberOfFlightsSucceed/iNumberOfRuns;
// Calculate Variances
fVarianceFlightsOnTime = (iSquareNumberOfFlightsOnTime -
    (2 * fNumberOfFlightsOnTime * fMeanFlightsOnTime) +
    (iNumberOfRuns * pow(fMeanFlightsOnTime,2)))/(iNumberOfRuns - 1);
fVarianceFlightsFirstHalfDelay = (iSquareNumberOfFlightsFirstHalfDelay -
    (2 * fNumberOfFlightsFirstHalfDelay * fMeanFlightsFirstHalfDelay) +
    (iNumberOfRuns * pow(fMeanFlightsFirstHalfDelay,2)))/(iNumberOfRuns - 1);
fVarianceFlightsSecondHalfDelay = (iSquareNumberOfFlightsSecondHalfDelay -
    (2 * fNumberOfFlightsSecondHalfDelay * fMeanFlightsSecondHalfDelay) +
    (iNumberOfRuns * pow(fMeanFlightsSecondHalfDelay,2)))/
    (iNumberOfRuns - 1);
fVarianceFlightsCancelled = (iSquareNumberOfFlightsCancelled -
    (2 * fNumberOfFlightsCancelled * fMeanFlightsCancelled) +
    (iNumberOfRuns * pow(fMeanFlightsCancelled,2)))/(iNumberOfRuns - 1);
fVarianceFlightsInFlightAbort = (iSquareNumberOfFlightsInFlightAbort -
    (2 * fNumberOfFlightsInFlightAbort * fMeanFlightsInFlightAbort) +
    (iNumberOfRuns * pow(fMeanFlightsInFlightAbort,2)))/(iNumberOfRuns - 1);
fVarianceFlightsTakeOff = (iSquareNumberOfFlightsTakeOff -
    (2 * fNumberOfFlightsTakeOff * fMeanFlightsTakeOff) +
    (iNumberOfRuns * pow(fMeanFlightsTakeOff,2)))/(iNumberOfRuns - 1);
fVarianceFlightsFail = (iSquareNumberOfFlightsFail -
    (2 * fNumberOfFlightsFail * fMeanFlightsFail) +
    (iNumberOfRuns * pow(fMeanFlightsFail,2)))/(iNumberOfRuns - 1);
fVarianceFlightsSucceed = (iSquareNumberOfFlightsSucceed -
    (2 * fNumberOfFlightsSucceed * fMeanFlightsSucceed) +
    (iNumberOfRuns * pow(fMeanFlightsSucceed,2)))/(iNumberOfRuns - 1);
// Calculate Percentages
fPercentageFlightsOnTime = fMeanFlightsOnTime/fMeanFlightsTasked *100;
fPercentageFlightsFirstHalfDelay = fMeanFlightsFirstHalfDelay/
    fMeanFlightsTasked *100;
fPercentageFlightsSecondHalfDelay = fMeanFlightsSecondHalfDelay/
    fMeanFlightsTasked *100;
fPercentageFlightsCancelled = fMeanFlightsCancelled/fMeanFlightsTasked *100;
fPercentageFlightsTakeOff = fPercentageFlightsOnTime +
    fPercentageFlightsFirstHalfDelay + fPercentageFlightsSecondHalfDelay;
fPercentageFlightsInFlightAbort = fMeanFlightsInFlightAbort/
    fMeanFlightsTasked*100;
fPercentageFlightsFail = fMeanFlightsFail/fMeanFlightsTakeOff*100;
fPercentageFlightsSucceed = fMeanFlightsSucceed/fMeanFlightsTakeOff*100;
// Save The Simulation Results to the results file
// Simulation Filenames
ResultsOF << "Simulation Run Results File\n\n";
ResultsOF << "Simulation Files\n";
ResultsOF << "Simulation Input File\t\t" << sSimParametersFile << "\n";
ResultsOF << "Flying Programme \t\t" << sFlyingProgrammeFile << "\n";
ResultsOF << "LRI Data File \t\t" << sLRIDataFile << "\n";
```

```
ResultsOF << "LRI Stock File \t\t" << sLRIStockFile << "\n";
ResultsOF << "Results File \t\t" << sResultsFile << "\n\n";
// Simulation Input Parameters
ResultsOF << "Simulation Input parameters \n";
ResultsOF << "Number of Runs\t\t\t\t"<< iNumberOfRuns << "\n";
ResultsOF << "Number of Days\t\t\t\t"<< iNumberOfDays << "\n";
ResultsOF << "Number of Aircraft\t\t\t\t" << iStartNumberOfAircraft << "\n";
ResultsOF << "Number of Pre-Flight Servicing teams\t\t\t\t" <<
    iNumberOfPreFlightServicingTeams << "\n";
ResultsOF << "Pre-Flight Servicing Duration (hours)\t\t\t\t" <<
    fPreFlightServicingDuration << "\n";
ResultsOF << "Maximum Delay for Flights (hours)\t\t\t\t" <<
    fMaximumFlightDelay << "\n\n";
// Distribution Types and Parameters
ResultsOF << "Distributions Used\n";
// Failure Distributions used
switch (iFailureDistributionUsed)
{
case 0: // None
    ResultsOF << "Failure - None";
    break;
case 1: // Exponential
    ResultsOF << "Failure - Exponential";
    break;
case 2: // Lognormal
    ResultsOF << "Failure - Lognormal\nVariance" << fFailureVariance;
    break;
case 3: // Normal
    ResultsOF << "Failure - Normal\nMinimum\t" << fFailureMin <<
        "\nMaximum\t" << fFailureMax;
    ResultsOF << "\nVariance\t" << fFailureVariance;
    break;
case 4: // Triangular
    ResultsOF << "Failure - Triangular\nMinimum\t" << fFailureMin<<
        "\nMaximum\t" << fFailureMax;
    ResultsOF << "\nDivisor\t" << iFailureDivisor;
    break;
case 5: // Weibull
    ResultsOF << "Failure - Weibull\nShape\t" << fFailureShape;
    ResultsOF << "\nGamma\t" << fFailureGamma;
    break;
}
switch (iRepairDistributionUsed)
{
case 0: // None
    ResultsOF << "\nRepair - None";
    break;
case 1: // Exponential
    ResultsOF << "\nRepair - Exponential";
    break;
case 2: // Lognormal
```

```
        ResultsOF << "\nRepair - Lognormal\nVariance\t" << fRepairVariance;
        break;
    case 3: // Normal
        ResultsOF << "\nRepair - Normal\nMinimum\t" << fRepairMin <<
            "\nMaximum\t" << fRepairMax;
        ResultsOF << "\nVariance\t" << fRepairVariance;
        break;
    case 4: // Triangular
        ResultsOF << "\nRepair - Triangular\nMinimum\t" << fRepairMin<<
            "\nMaximum\t" << fRepairMax;
        ResultsOF << "\nDivisor\t" << iRepairDivisor;
        break;
    case 5: // Weibull
        ResultsOF << "\nRepair - Weibull\nShape\t" << fRepairShape;
        ResultsOF << "\nGamma\t" << fRepairGamma;
        break;
    }
    // Simulation Flight Statistics
    ResultsOF << "\n\nSimulation Means\n";
    ResultsOF << "\t\t\t\tTotal\tVariance\tPercentage\n";
    ResultsOF << "Tasked Flights\t\t\t\t" << fMeanFlightsTasked <<"\n";
    ResultsOF << "Flights Achieved\t\t\t\t" <<fMeanFlightsTakeOff << "\t" <<
            fVarianceFlightsTakeOff << "\t" << fPercentageFlightsTakeOff <<"\n";
    ResultsOF << "On Time\t\t\t\t" << fMeanFlightsOnTime << "\t"<<
        fVarianceFlightsOnTime << "\t" << fPercentageFlightsOnTime << "\n";
    ResultsOF << "First Half Flight Delay Maximum\t\t\t\t" <<
        fMeanFlightsFirstHalfDelay << "\t" << fVarianceFlightsFirstHalfDelay << "\t" <<
        fPercentageFlightsFirstHalfDelay << "\n";
    ResultsOF << "Second Half Flight Delay Maximum\t\t\t\t" <<
        fMeanFlightsSecondHalfDelay << "\t" << fVarianceFlightsSecondHalfDelay <<
        "\t" << fPercentageFlightsSecondHalfDelay << "\n";
    ResultsOF << "Cancelled Flights\t\t\t\t" << fMeanFlightsCancelled << "\t" <<
        fVarianceFlightsCancelled << "\t" << fPercentageFlightsCancelled << "\n";
    ResultsOF << "In Flight Aborts\t\t\t\t" << fMeanFlightsInFlightAbort << "\t" <<
        fVarianceFlightsInFlightAbort << "\t" << fPercentageFlightsInFlightAbort <<
        "\n";
    ResultsOF << "Launched Flights Succeeded\t\t\t\t" << fMeanFlightsSucceed <<
        "\t" << fVarianceFlightsSucceed << "\t" << fPercentageFlightsSucceed << "\n";
    ResultsOF << "Launched Flights Failed\t\t\t\t" << fMeanFlightsFail << "\t" <<
        fVarianceFlightsFail << "\t" << fPercentageFlightsFail << "\n\n";
    // Aircraft mean time per run spent in States
    ResultsOF << "Mean Hours For Aircraft States \n\n";
    ResultsOF << "Aircraft\tUnserviceable\tAwaiting Flight\tIn Flight\tServicable\t
        Flying\n";
    ResultsOF << "Number\t\tServicing\tServicing\n";
    // Zero Counters for calculation of means
    fTimeUnserviceable = 0;
    fTimeAwaitingPreFlightServicing = 0;
    fTimeInPreFlightServicing = 0;
    fTimeServiceable = 0;
    fTimeFlying = 0;
```

```
fMeanTimeUnserviceable = 0;
fMeanTimeAwaitingPreFlightServicing = 0;
fMeanTimeInPreFlightServicing = 0;
fMeanTimeServiceable = 0;
fMeanTimeFlying = 0;
for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos!=NULL;)
{
    pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
    iAircraftNumber = pAircraft->GetAircraftNumber();
    ResultsOF << iAircraftNumber <<"\t";
    fTimeUnserviceable = pAircraft->GetTimeUnserviceable(iNumberOfRuns);
    fMeanTimeUnserviceable += fTimeUnserviceable;
    ResultsOF <<fTimeUnserviceable << "\t";
    fTimeAwaitingPreFlightServicing = pAircraft->
        GetTimeAwaitingPreFlightServicing(iNumberOfRuns);
    fMeanTimeAwaitingPreFlightServicing += fTimeAwaitingPreFlightServicing;
    ResultsOF << fTimeAwaitingPreFlightServicing << "\t";
    fTimeInPreFlightServicing = pAircraft->
        GetTimeInPreFlightServicing(iNumberOfRuns);
    fMeanTimeInPreFlightServicing += fTimeInPreFlightServicing;
    ResultsOF << fTimeInPreFlightServicing  << "\t";
    fTimeServiceable = pAircraft->GetTimeServiceable(iNumberOfRuns);
    fMeanTimeServiceable += fTimeServiceable;
    ResultsOF << fTimeServiceable << "\t";
    fTimeFlying = pAircraft->GetTimeFlying(iNumberOfRuns);
    fMeanTimeFlying += fTimeFlying;
    ResultsOF << fTimeFlying << "\n";
}
// save Mean values to file
fMeanTimeUnserviceable = fMeanTimeUnserviceable/iAircraftNumber;
fMeanTimeAwaitingPreFlightServicing =
    fMeanTimeAwaitingPreFlightServicing/iAircraftNumber;
fMeanTimeInPreFlightServicing = fTimeInPreFlightServicing/iAircraftNumber;
fMeanTimeServiceable = fMeanTimeServiceable/iAircraftNumber;
fMeanTimeFlying = fMeanTimeFlying/iAircraftNumber;
ResultsOF << "Mean\t" << fMeanTimeUnserviceable << "\t" <<
    fMeanTimeAwaitingPreFlightServicing << "\t" << fTimeInPreFlightServicing <<
    "\t" << fMeanTimeServiceable << "\t" << fMeanTimeFlying << "\n";
// Aircraft percentage of time per run in States
ResultsOF << "\nPercentage of Simulation For Aircraft States \n\n";
ResultsOF << "Aircraft\tUnserviceable\tAwaiting Flight\tIn Flight\tServicable\t
    Flying\n";
ResultsOF << "Number\t\tServicing\tServicing\n";
// Zero Counters for calculation of means
fTimeUnserviceable = 0;
fTimeAwaitingPreFlightServicing = 0;
fTimeInPreFlightServicing = 0;
fTimeServiceable = 0;
fTimeFlying = 0;
fMeanTimeUnserviceable = 0;
fMeanTimeAwaitingPreFlightServicing = 0;
```

```
fMeanTimeInPreFlightServicing = 0;
fMeanTimeServiceable = 0;
fMeanTimeFlying = 0;
for (AircraftPos = aircraftlist.GetHeadPosition();AircraftPos!=NULL;)
{
    pAircraft = (CAircraft*)aircraftlist.GetNext(AircraftPos);
    iAircraftNumber = pAircraft->GetAircraftNumber();
    ResultsOF << iAircraftNumber <<"\t";
    fTimeUnserviceable = pAircraft->
        GetTimeUnserviceable(iNumberOfRuns)/fClockTime*100;
    fMeanTimeUnserviceable += fTimeUnserviceable;
    ResultsOF <<fTimeUnserviceable << "\t";
    fTimeAwaitingPreFlightServicing = pAircraft->
        GetTimeAwaitingPreFlightServicing(iNumberOfRuns)/fClockTime*100;
    fMeanTimeAwaitingPreFlightServicing += fTimeAwaitingPreFlightServicing;
    ResultsOF <<fTimeAwaitingPreFlightServicing << "\t";
    fTimeInPreFlightServicing = pAircraft->
        GetTimeInPreFlightServicing(iNumberOfRuns)/fClockTime*100;
    fMeanTimeInPreFlightServicing += fTimeInPreFlightServicing;
    ResultsOF <<fTimeInPreFlightServicing << "\t";
    fTimeServiceable = pAircraft->
        GetTimeServiceable(iNumberOfRuns)/fClockTime*100;
    fMeanTimeServiceable += fTimeServiceable;
    ResultsOF <<fTimeServiceable << "\t";
    fTimeFlying = pAirc aft->GetTimeFlying(iNumberOfRuns)/fClockTime*100;
    fMeanTimeFlying += fTimeFlying;
    ResultsOF <<fTimeFlying << "\n";
}
// save Mean values to file
fMeanTimeUnserviceable = fMeanTimeUnserviceable/iAircraftNumber;
fMeanTimeAwaitingPreFlightServicing =
    fMeanTimeAwaitingPreFlightServicing/iAircraftNumber;
fTimeInPreFlightServicing = fTimeInPreFlightServicing/iAircraftNumber;
fMeanTimeServiceable = fMeanTimeServiceable/iAircraftNumber;
fMeanTimeFlying = fMeanTimeFlying/iAircraftNumber;
ResultsOF << "Mean\t" << fMeanTimeUnserviceable << "\t" <<
    fMeanTimeAwaitingPreFlightServicing << "\t" << fTimeInPreFlightServicing <<
    "\t" << fMeanTimeServiceable << "\t" << fMeanTimeFlying << "\n";
// Daily Aircraft State figures
ResultsOF << "\nDaily AircraftStates\n\n";
ResultsOF << "Day\tServiceable\tFlying\tUnserviceable\tIn Pre\tAwaiting\n";
ResultsOF << "Number\t\t\t\tFlight Servicing\tPre Flight Servicing\n";
for (iDayNumber = 1;iDayNumber <= iNumberOfDays;iDayNumber++)
{
    for (DailyResultsPos = dailyresultslist.GetHeadPosition();
        DailyResultsPos!=NULL;)
    {
        pDailyResults = (CDailyResults*)dailyresultslist.GetNext(DailyResultsPos);
        if (pDailyResults->GetDayNumber() == iDayNumber)
        {
            // Read in Aircraft state details
```

```
                fNumberOfAircraftServiceable = pDailyResults->
                    GetNumberOfAircraftServiceable();
                fNumberOfAircraftFlying = pDailyResults->
                    GetNumberOfAircraftFlying();
                fNumberOfAircraftUnserviceable = pDailyResults->
                    GetNumberOfAircraftUnserviceable();
                fNumberOfAircraftInPreFlightServicing = pDailyResults->
                    GetNumberOfAircraftInPreFlightServicing();
                fNumberOfAircraftAwaitingPreFlightServicing = pDailyResults->
                    GetNumberOfAircraftAwaitingPreFlightServicing();
                // Calculate the means
                fMeanNumberOfAircraftServiceable =
                    fNumberOfAircraftServiceable/iNumberOfRuns;
                fMeanNumberOfAircraftFlying =
                    fNumberOfAircraftFlying/iNumberOfRuns;
                fMeanNumberOfAircraftUnserviceable =
                    fNumberOfAircraftUnserviceable/iNumberOfRuns;
                fMeanNumberOfAircraftInPreFlightServicing =
                    fNumberOfAircraftInPreFlightServicing/iNumberOfRuns;
                fMeanNumberOfAircraftAwaitingPreFlightServicing =
                    fNumberOfAircraftAwaitingPreFlightServicing/iNumberOfRuns;
                ResultsOF << iDayNumber << "\t" <<
                    fMeanNumberOfAircraftServiceable << "\t" <<
                    fMeanNumberOfAircraftFlying;
                ResultsOF << "\t" << fMeanNumberOfAircraftUnserviceable << "\t" <<
                    fMeanNumberOfAircraftInPreFlightServicing;
                ResultsOF << "\t" << fMeanNumberOfAircraftAwaitingPreFlightServicing
                    << "\n";
                break;
            }
        }
    }
    // Daily Flight Results
    ResultsOF << "\nDaily Flight Results\n\n";
    ResultsOF << "Day\tTasked\tOn Time\tLess Than\t
        More Than\tCancelled\tSuccessful\tIn Flight\tSuccessful\tFailed\n";
    ResultsOF << "Number\tFlights\t\tHalf Maximum Delay\t
        Half Maximum Delay\t\tTake Offs\tAbort\tMissions\tMissions\n";
    for (iDayNumber = 1;iDayNumber <= iNumberOfDays;iDayNumber++)
    {
        for (DailyResultsPos =
            dailyresultslist.GetHeadPosition();DailyResultsPos!=NULL;)
        {
            pDailyResults = (CDailyResults*)dailyresultslist.GetNext(DailyResultsPos);
            if (pDailyResults->GetDayNumber() == iDayNumber)
            {
                // Read in Flight Details
                fNumberOfDailyFlightsCancelled = pDailyResults->
                    GetNumberOfDailyFlightsCancelled();
                fNumberOfDailyFlightsFail = pDailyResults->
                    GetNumberOfDailyFlightsFail();
```

```
                fNumberOfDailyFlightsFirstHalfDelay = pDailyResults->
                    GetNumberOfDailyFlightsFirstHalfDelay();
                fNumberOfDailyFlightsInFlightAbort = pDailyResults->
                    GetNumberOfDailyFlightsInFlightAbort();
                fNumberOfDailyFlightsOnTime = pDailyResults->
                    GetNumberOfDailyFlightsOnTime();
                fNumberOfDailyFlightsSecondHalfDelay = pDailyResults->
                    GetNumberOfDailyFlightsSecondHalfDelay();
                fNumberOfDailyFlightsSucceed = pDailyResults->
                    GetNumberOfDailyFlightsSucceed();
                fNumberOfDailyFlightsTasked = pDailyResults->
                    GetNumberOfDailyFlightsTasked();
                // Calculte the means
                fNumberOfDailyFlightsTakeOff = fNumberOfDailyFlightsOnTime +
                    fNumberOfDailyFlightsFirstHalfDelay +
                    fNumberOfDailyFlightsSecondHalfDelay;
                fMeanNumberOfDailyFlightsCancelled =
                    fNumberOfDailyFlightsCancelled/iNumberOfRuns;
                fMeanNumberOfDailyFlightsFail =
                    fNumberOfDailyFlightsFail/iNumberOfRuns;
                fMeanNumberOfDailyFlightsFirstHalfDelay =
                    fNumberOfDailyFlightsFirstHalfDelay/iNumberOfRuns;
                fMeanNumberOfDailyFlightsInFlightAbort =
                    fNumberOfDailyFlightsInFlightAbort/iNumberOfRuns;
                fMeanNumberOfDailyFlightsOnTime =
                    fNumberOfDailyFlightsOnTime/iNumberOfRuns;
                fMeanNumberOfDailyFlightsSecondHalfDelay =
                    fNumberOfDailyFlightsSecondHalfDelay/iNumberOfRuns;
                fMeanNumberOfDailyFlightsSucceed =
                    fNumberOfDailyFlightsSucceed/iNumberOfRuns;
                fMeanNumberOfDailyFlightsTakeOff =
                    fNumberOfDailyFlightsTakeOff/iNumberOfRuns;
                fMeanNumberOfDailyFlightsTasked =
                    fNumberOfDailyFlightsTasked/iNumberOfRuns;
                // Save the means to the results file
                ResultsOF << iDayNumber << "\t" <<
                    fMeanNumberOfDailyFlightsTasked << "\t" <<
                    fMeanNumberOfDailyFlightsOnTime ;
                ResultsOF << "\t" << fMeanNumberOfDailyFlightsFirstHalfDelay <<
                    "\t" << fMeanNumberOfDailyFlightsSecondHalfDelay;
                ResultsOF << "\t" << fMeanNumberOfDailyFlightsCancelled <<
                    "\t" << fMeanNumberOfDailyFlightsTakeOff;
                ResultsOF << "\t" << fMeanNumberOfDailyFlightsInFlightAbort <<
                    "\t" << fMeanNumberOfDailyFlightsSucceed;
                ResultsOF << "\t" << fMeanNumberOfDailyFlightsFail << "\n";
            }
        }
    }
    // Release the memory used before displaying results
    ReleaseMemory(aircraftlist,randomnumberlist,dailyresultslist,stocklist);
```

```
    // Show basic simulation results on screen
    CDlgSimulationFinished dlg;
    dlg.m_MeanFlightsTasked = fMeanFlightsTasked;
    dlg.m_MeanFlightsOnTime = fMeanFlightsOnTime;
    dlg.m_MeanFlightsFirstHalfDelay = fMeanFlightsFirstHalfDelay;
    dlg.m_MeanFlightsSecondHalfDelay = fMeanFlightsSecondHalfDelay;
    dlg.m_MeanFlightsCancelled = fMeanFlightsCancelled;
    dlg.m_MeanFlightsTakeOff = fMeanFlightsTakeOff;
    dlg.m_MeanFlightsInFlightAbort = fMeanFlightsInFlightAbort;
    dlg.m_MeanFlightsFail = fMeanFlightsFail;
    dlg.m_MeanFlightsSucceed = fMeanFlightsSucceed;
    dlg.m_PercentageFlightsOnTime = fPercentageFlightsOnTime;
    dlg.m_PercentageFlightsFirstHalfDelay = fPercentageFlightsFirstHalfDelay;
    dlg.m_PercentageFlightsSecondHalfDelay = fPercentageFlightsSecondHalfDelay;
    dlg.m_PercentageFlightsCancelled = fPercentageFlightsCancelled;
    dlg.m_PercentageFlightsTakeOff = fPercentageFlightsTakeOff;
    dlg.m_PercentageFlightsInFlightAbort = fPercentageFlightsInFlightAbort;
    dlg.m_PercentageFlightsFail = fPercentageFlightsFail;
    dlg.m_PercentageFlightsSucceed = fPercentageFlightsSucceed;
    // Dialog to allow choice between exiting the programme or running another problem
    if (dlg.DoModal() == IDOK)
    {
        CDlgClearInputParameters dlg;
        if (dlg.DoModal() == IDOK)
        {
            bSimulationParametersFileLoaded = false;
            bFlyingProgrammeFileLoaded = false;
            bLRIDataFileLoaded = false;
            bLRIStockFileLoaded = false;
        }
        bResultsFileLoaded = false;
    }
}
//Insert an event into the queue
void CALSSimDoc::InsertEvent(CObList& eventlist, int iEventNumber,
        float fEventTime,float fPreviousEventTime,int iAircraftNumber,
        int iNumberOfAircraft,int iLRIPosition,int iLRIType,int iNumberOfLRIs,
        float fEventDuration)
{
    // Set up pointers to Event Class
    CEvent* pEvent;
    POSITION EventPos1, EventPos2;
    for(EventPos1 = eventlist.GetHeadPosition();EventPos1 != NULL;)
    {
        // set up a second position reference
        // this is required as GetNext moves the pointer on one place
        EventPos2 = EventPos1;
        pEvent=(CEvent*)eventlist.GetNext(EventPos1);
        if (pEvent->GetEventTime() > fEventTime)
        {
            eventlist.InsertBefore(EventPos2,new CEvent(iEventNumber,fEventTime,
```

```
                fPreviousEventTime,iAircraftNumber,iNumberOfAircraft,iLRIPosition,
                iLRIType,iNumberOfLRIs,fEventDuration));
            break;
        }
    }
}
void CALSSimDoc::LoadSimulationScenarioDetails()
{
    // Open the Simulation Parameter files
    ifstream SimParaIF(sSimParametersFile);
    SimParaIF>> iNumberOfRuns >> iNumberOfAircraft >>
        iMaximumFailuresNonEssentialLRIs  >> fMaximumFlightDelay >>
        fMissionSuccessPoint >> iNumberOfPreFlightServicingTeams >>
        fPreFlightServicingDuration >> fDepotUnitTransferTime >>
        iFailureDistributionUsed >> iRepairDistributionUsed >> fReliabilityFactor >>
        fRepairFactor >> fNoFaultFoundAtUnitFactor >> fNoFaultFoundAtDepotFactor;
    // Read in extra data for those distributions that need it
    switch (iFailureDistributionUsed)
    {
        case 2: // Lognormal - Variance for underlying Normal
            SimParaIF >> fFailureVariance;
            break;
        case 3:  // Normal - Min, Max and Variance needed
            SimParaIF >> fFailureMin >>fFailureMax >> fFailureVariance;
            break;
        case 4:  // Triangular - Max, Min and Mode Divisor needed
            SimParaIF >> fFailureMin >>fFailureMax >> iFailureDivisor;
            break;
        case 5:  // Weibull - Shape and Gamma needed
            SimParaIF >> fFailureShape >> fFailureGamma;
            break;
    }
    switch (iRepairDistributionUsed)
    {
        case 2: // Lognormal - Variance for underlying Normal
            SimParaIF >> fRepairVariance;
            break;
        case 3:  // Normal - Min, Max and Variance needed
            SimParaIF >> fRepairMin >>fRepairMax >> fRepairVariance;
            break;
        case 4:  // Triangular - Max, Min and Mode Divisor needed
            SimParaIF >> fRepairMin >>fRepairMax >> iRepairDivisor;
            break;
        case 5:  // Weibull - Shape and Gamma needed
            SimParaIF >> fRepairShape >>fFailureGamma;
            break;
    }
    // get value for Aircraft and Pre Flight Servicing Teams for resetting for each run
    iStartNumberOfAircraft = iNumberOfAircraft;
    iStartRunNumberOfPreFlightServicingTeams = iNumberOfPreFlightServicingTeams;
}
```

```
void CALSSimDoc::OnFileOpenFlyingProgramme()
{
    static char BASED_CODE szFilter[] =
        "Flying Programme Files (*.fly)|*.fly|All Files (*.*)"\
        "|*.*||";
    CFileDialog dlg(true,NULL,sFlyingProgrammeFile,NULL,szFilter);
    if (dlg.DoModal() == IDOK)
    {
        sFlyingProgrammeFile = dlg.GetPathName();
        // Check for bad Flying Programme file
        if(!IsValidFileSpec(sFlyingProgrammeFile))
        {
            bFlyingProgrammeFileLoaded = false;
            CString ErrMsg;
            AfxFormatString1(ErrMsg,IDS_ERRFMT_INVALIDFILE2,
                sFlyingProgrammeFile);
            MessageBox (NULL,ErrMsg,"File Selection Error",MB_ICONERROR);
        }
        else
        {
            bFlyingProgrammeFileLoaded = true;
        }
    }
    else
    {
        sFlyingProgrammeFile = "";
        bFlyingProgrammeFileLoaded = false;
    }
}
void CALSSimDoc::OnFileOpenLRIDataFile()
{
    static char BASED_CODE szFilter[] =
        "LRI Data Files (*.dat)|*.dat|All Files (*.*)"\
        "|*.*||";
    CFileDialog dlg(true,NULL,sLRIDataFile,NULL,szFilter);
    if (dlg.DoModal() == IDOK)
    {
        sLRIDataFile = dlg.GetPathName();
        // Check for bad LRI Data file
        if(!IsValidFileSpec(sLRIDataFile))
        {
            bLRIDataFileLoaded = false;
            CString ErrMsg;
            AfxFormatString1(ErrMsg,IDS_ERRFMT_INVALIDFILE3,sLRIDataFile);
            MessageBox (NULL,ErrMsg,"File Selection Error",MB_ICONERROR);
        }
        else
        {
            bLRIDataFileLoaded = true;
        }
    }
```

```
    else
    {
        sLRIDataFile = "";
        bLRIDataFileLoaded = false;
    }
}
void CALSSimDoc::OnFileOpenLRIStockFile()
{
    static char BASED_CODE szFilter[] =
        "LRI Stock Files (*.stk)|*.stk|All Files (*.*)"\
        "|*.*||";
    CFileDialog dlg(true,NULL,sLRIStockFile,NULL,szFilter);
    if (dlg.DoModal() == IDOK)
    {
        sLRIStockFile = dlg.GetPathName();
        //  Check for bad LRI Stock file
        if(!IsValidFileSpec(sLRIStockFile))
        {
            bLRIStockFileLoaded = false;
            CString ErrMsg;
            AfxFormatString1(ErrMsg,IDS_ERRFMT_INVALIDFILE4,sLRIStockFile);
            MessageBox (NULL,ErrMsg,"File Selection Error",MB_ICONERROR);
        }
        else
        {
            bLRIStockFileLoaded = true;
        }
    }
    else
    {
        sLRIStockFile = "";
        bLRIStockFileLoaded = false;
    }
}
void CALSSimDoc::OnFileOpenResultsFile()
{
    static char BASED_CODE szFilter[] =
        "Results Files (*.res)|*.res|All Files (*.*)"\
        "|*.*||";
    CFileDialog dlg(false,NULL,sResultsFile,NULL,szFilter);
    if (dlg.DoModal() == IDOK)
    {
        sResultsFile = dlg.GetPathName();
        if (dlg.GetFileExt() == "")
        {
            sResultsFile += ".res";
        }
        // Check if Results file selected exists
        if(!IsValidFileSpec(sResultsFile))
        {
            hFile = (HANDLE)CreateFile(sResultsFile,GENERIC_WRITE,0,
```

```
                NULL ,CREATE_NEW, FILE_ATTRIBUTE_NORMAL,NULL);
            CloseHandle( hFile);
            bResultsFileLoaded = true;
        }
        // Results file already exists
        else
        {
            CString FileMsg;
            AfxFormatString1(FileMsg, IDS_ERRFMT_FILEEXISTS,sResultsFile);
            if(MessageBox (NULL,FileMsg,"Results Filename",
                MB_SYSTEMMODAL|MB_ICONQUESTION|MB_YESNO) ==
            IDYES)
            {
                hFile = (HANDLE)CreateFile(sResultsFile,GENERIC_WRITE,0,
                    NULL ,CREATE_ALWAYS,
                FILE_ATTRIBUTE_NORMAL,NULL);
                CloseHandle( hFile);
                bResultsFileLoaded = true;
            }
            else
            {
                bResultsFileLoaded = false;
            }
        }
    }
    else
    {
        sResultsFile = "";
        bResultsFileLoaded = false;

    }
}
void CALSSimDoc::OnFileOpenSimulationParameters()
{
    static char BASED_CODE szFilter[] =
        "Simulation Parameter Files (*.sim)|*.sim|All Files (*.*)"\
        "|*.*||";
    CFileDialog dlg(true,NULL,sSimParametersFile,NULL,szFilter);
    if (dlg.DoModal() == IDOK)
    {
        sSimParametersFile = dlg.GetPathName();
        //  Check for bad Simulation Parameter file
        if(!IsValidFileSpec(sSimParametersFile))
        {
            bSimulationParametersFileLoaded = false;
            CString ErrMsg;
            AfxFormatString1(ErrMsg,IDS_ERRFMT_INVALIDFILE1,
                sSimParametersFile);
            MessageBox (NULL,ErrMsg,"File Selection Error",MB_ICONERROR);
        }
        else
```

```
        {
            bSimulationParametersFileLoaded = true;
        }
    }
    else
    {
        sSimParametersFile = "";
        bSimulationParametersFileLoaded = false;
    }
}
void CALSSimDoc::OnUpdateFileOpenFlyingProgramme(CCmdUI* pCmdUI)
{
    if(bFlyingProgrammeFileLoaded == true)
    {
        pCmdUI->SetCheck(1);
    }
    else
        pCmdUI->SetCheck(0);
}
void CALSSimDoc::OnUpdateFileOpenLRIDataFile(CCmdUI* pCmdUI)
{
    if(bLRIDataFileLoaded == true)
    {
        pCmdUI->SetCheck(1);
    }
    else
        pCmdUI->SetCheck(0);
}
void CALSSimDoc::OnUpdateFileOpenLRIStockFile(CCmdUI* pCmdUI)
{
    if(bLRIStockFileLoaded == true)
    {
        pCmdUI->SetCheck(1);
    }
    else
        pCmdUI->SetCheck(0);
}
void CALSSimDoc::OnUpdateFileOpenResultsFile(CCmdUI* pCmdUI)
{
    if(bResultsFileLoaded == true)
    {
        pCmdUI->SetCheck(1);
    }
    else
        pCmdUI->SetCheck(0);
}
void CALSSimDoc::OnUpdateFileOpenSimulationParameters(CCmdUI* pCmdUI)
{
    if(bSimulationParametersFileLoaded == true)
    {
        pCmdUI->SetCheck(1);
```

```
    }
    else
        pCmdUI->SetCheck(0);
}
void CALSSimDoc::OnUpdateSimulationRun(CCmdUI* pCmdUI)
{
    if((bSimulationParametersFileLoaded == true) &&
        (bFlyingProgrammeFileLoaded == true) && (bLRIDataFileLoaded == true) &&
        (bLRIStockFileLoaded == true)&& (bResultsFileLoaded == true))
    {
        pCmdUI->Enable(true);
    }
    else
        pCmdUI->Enable(false);
}
void CALSSimDoc::ReleaseMemory(CObList& aircraftlist,
    CObList& randomnumberlist,CObList& dailyresultslist,CObList& stocklist)
// Deletes the lists and releases the memory used by the simulation
{
    // Set up pointers to the simulation objects
    CAircraft* pAircraft;
    POSITION AircraftPos = aircraftlist.GetHeadPosition();
    // Delete the Aircraft and LRI objects
    while (AircraftPos != NULL)
    {
        // Delete the LRI objects for each aircraft
        pAircraft =(CAircraft*)aircraftlist.GetAt(AircraftPos);
        pAircraft->DeleteLRIs();
        delete aircraftlist.GetNext(AircraftPos);
    }
    aircraftlist.RemoveAll();
    // Delete the Random Number objects
    POSITION RandomNumberPos = randomnumberlist.GetHeadPosition();
    while (RandomNumberPos != NULL)
    {
        delete randomnumberlist.GetNext(RandomNumberPos);
    }
    randomnumberlist.RemoveAll();
    // Delete the Daily Results objects
    POSITION DailyResultsPos = dailyresultslist.GetHeadPosition();
    while (DailyResultsPos != NULL)
    {
        delete dailyresultslist.GetNext(DailyResultsPos);
    }
    dailyresultslist.RemoveAll();
    //Delete the Stock Objects
    POSITION StockPos = stocklist.GetHeadPosition();
    while (StockPos != NULL)
    {
        delete stocklist.GetNext(StockPos);
    }
```

```
    stocklist.RemoveAll();
}
void CALSSimDoc::SaveDailyFlyingStats(CObList& dailyresultslist,int iDayNumber,
int iNumberOfDailyFlightsTasked,int iNumberOfDailyFlightsOnTime,
int iNumberOfDailyFlightsFirstHalfDelay,int iNumberOfDailyFlightsSecondHalfDelay,
int iNumberOfDailyFlightsCancelled,int iNumberOfDailyFlightsInFlightAbort,
int iNumberOfDailyFlightsFail,int iNumberOfDailyFlightsSucceed)
{
    // Set up pointers to the simulation objects
    CDailyResults* pDailyResults;
    POSITION DailyResultsPos;
    // Get correct daily results memory block then
    // store daily flight achievements for the day
    for (DailyResultsPos = dailyresultslist.GetHeadPosition();DailyResultsPos != NULL;)
    {
        pDailyResults =(CDailyResults*)dailyresultslist.GetNext(DailyResultsPos);
        if (pDailyResults->GetDayNumber() == (iDayNumber - 1))
        {
            pDailyResults->SaveDailyFlyingStats(iNumberOfDailyFlightsTasked,
                iNumberOfDailyFlightsOnTime,iNumberOfDailyFlightsFirstHalfDelay,
                iNumberOfDailyFlightsSecondHalfDelay,
                iNumberOfDailyFlightsCancelled,iNumberOfDailyFlightsInFlightAbort,
                iNumberOfDailyFlightsFail,iNumberOfDailyFlightsSucceed);
            break;
        }
    }
}
void CALSSimDoc::SaveEvent(int iEventNumber,float fClockTime,int iAircraftNumber,
    int iNumberOfAircraft,int iLRIPosition,int iLRIType,int iNumberOfLRIs,
    float fEventDuration)
{
    // Open the Events storage file
    ofstream EventOF("Event.txt",ios::app );
    // Save event details
    EventOF << iEventNumber <<"\t"<< fClockTime <<"\t" <<iAircraftNumber<<
        "\t" << iNumberOfAircraft << "\t" << iLRIPosition <<"\t"<< iLRIType<< "\t" <<
        iNumberOfLRIs << "\t" << fEventDuration <<"\n";
}
// Create the Simulation Event List
void CALSSimDoc::SetUpEventList(CObList& eventlist)
{
    // Open the flying programme file
    ifstream FlightDataIF(sFlyingProgrammeFile);
    // Open the Events storage file
    ofstream EventOF("Event.txt",ios::app );
    if (bStoringEvents)
    {
        // save run number
        EventOF << "Run Number ="<< iRunNumber << "\n";
    }
    // set up First New Day event for time = 0.0
```

```
    ieEventNumber = 1;
    feEventTime = 0.0;
    eventlist.AddTail(new CEvent(ieEventNumber,feEventTime));
    // Set up the target flying programme
    ieEventNumber = 2;
    ieAircraftNumber = 0;
    ieLRIPosition = 0;
    ieLRIType = 0;
    ieNumberOfLRIs = 0;
    fePreviousEventTime = 0;
    FlightDataIF >> feEventTime >> feEventDuration >> ieNumberOfAircraft;
    while (FlightDataIF)
    {
        eventlist.AddTail(new CEvent(ieEventNumber,feEventTime,
            fePreviousEventTime,ieAircraftNumber,ieNumberOfAircraft,ieLRIPosition,
            ieLRIType,ieNumberOfLRIs,feEventDuration));
        iNumberOfFlightsTasked+= ieNumberOfAircraft;
        FlightDataIF >> feEventTime >> feEventDuration >> ieNumberOfAircraft;
    }
    iNumberOfDays = feEventTime/24.0 + 1;
    // Set up new day events
    ieEventNumber = 1;
    ieAircraftNumber = 0;
    ieNumberOfAircraft = 0;
    ieLRIPosition = 0;
    ieLRIType = 0;
    ieNumberOfLRIs = 0;
    feEventDuration = 0;
    fePreviousEventTime = 0;
    for (iDayNumber =1;iDayNumber < iNumberOfDays;iDayNumber++)
    {
        feEventTime = iDayNumber * 24.0;
        InsertEvent(eventlist,ieEventNumber,feEventTime,fePreviousEventTime,
            ieAircraftNumber,ieNumberOfAircraft,ieLRIPosition,ieLRIType,
            ieNumberOfLRIs,feEventDuration);
    }
    // Set up end run event
    ieEventNumber = 0;
    feEventTime = iNumberOfDays * 24.0;
    eventlist.AddTail(new CEvent(ieEventNumber,feEventTime));
    // allocate an event number to ensure that get event runs
    iEventNumber =1;
    // If storing events, save event headings to the Event file
    if (bStoringEvents)
    {
        EventOF << "Event\tTime\tAircraft\tNumber of\tLRI\tLRI\tNumber\tEvent\n";
        EventOF << "Number\t\tNumber\tAircraft\tPosition\tType\tof LRIs\tDuration\n";
    }
}
void CALSSimDoc::ZeroDailyFlightsCounters()
{
```

```
    iNumberOfDailyFlightsCancelled = 0;
    iNumberOfDailyFlightsFail = 0;
    iNumberOfDailyFlightsFirstHalfDelay = 0;
    iNumberOfDailyFlightsInFlightAbort = 0;
    iNumberOfDailyFlightsOnTime = 0;
    iNumberOfDailyFlightsSecondHalfDelay = 0;
    iNumberOfDailyFlightsSucceed = 0;
    iNumberOfDailyFlightsTasked = 0;
}
void CALSSimDoc::ZeroDailyStatesCounters()
{
    // Zero the aircraft state counters
    iNumberOfAircraftAwaitingPreFlightServicing = 0;
    iNumberOfAircraftFlying = 0;
    iNumberOfAircraftInPreFlightServicing = 0;
    iNumberOfAircraftServiceable = 0;
    iNumberOfAircraftUnserviceable = 0;
}
void CALSSimDoc::ZeroRunCounters()
{
    // Zero Flight Counters
    iNumberOfFlightsDelayed = 0;
    iNumberOfFlightsTasked = 0;
    iNumberOfFlightsOnTime = 0;
    iNumberOfFlightsFirstHalfDelay = 0;
    iNumberOfFlightsSecondHalfDelay = 0;
    iNumberOfFlightsCancelled = 0;
    iNumberOfFlightsInFlightAbort = 0;
    iNumberOfFlightsFail = 0;
    iNumberOfFlightsSucceed = 0;
}
void CALSSimDoc::ZeroSimulationCounters()
{
    iSquareNumberOfFlightsCancelled = 0;
    iSquareNumberOfFlightsFail = 0;
    iSquareNumberOfFlightsSucceed = 0;
    iSquareNumberOfFlightsFirstHalfDelay = 0;
    iSquareNumberOfFlightsInFlightAbort = 0;
    iSquareNumberOfFlightsOnTime = 0;
    iSquareNumberOfFlightsSecondHalfDelay = 0;
    iSquareNumberOfFlightsTakeOff = 0;
    iTotalFlightsCancelled = 0;
    iTotalFlightsFail = 0;
    iTotalFlightsFirstHalfDelay = 0;
    iTotalFlightsInFlightAbort = 0;
    iTotalFlightsOnTime = 0;
    iTotalFlightsSecondHalfDelay = 0;
    iTotalFlightsSucceed = 0;
    iTotalFlightsTakeOff = 0;
    iTotalFlightsTasked = 0;
}
```

Daily Results

```
//DailyResults.h interface of the CDailyResults class
class CDailyResults : public CObject
{
//Attributes
private:
    int iDayNumber;
    int iNumberOfAircraftAwaitingPreFlightServicing;
    int iNumberOfAircraftFlying;
    int iNumberOfAircraftInPreFlightServicing;
    int iNumberOfAircraftServiceable;
    int iNumberOfAircraftUnserviceable;
    int iNumberOfDailyFlightsCancelled;
    int iNumberOfDailyFlightsFail;
    int iNumberOfDailyFlightsFirstHalfDelay;
    int iNumberOfDailyFlightsInFlightAbort;
    int iNumberOfDailyFlightsOnTime;
    int iNumberOfDailyFlightsSecondHalfDelay;
    int iNumberOfDailyFlightsSucceed;
    int iNumberOfDailyFlightsTasked;
//Operations
public:
    CDailyResults(){}
    CDailyResults(int iDayNo)
    {
        iDayNumber =iDayNo;
        iNumberOfAircraftServiceable= 0;
        iNumberOfAircraftFlying= 0;
        iNumberOfAircraftUnserviceable= 0;
        iNumberOfAircraftInPreFlightServicing= 0;
        iNumberOfAircraftAwaitingPreFlightServicing= 0;
        iNumberOfDailyFlightsCancelled = 0;
        iNumberOfDailyFlightsFail = 0;
        iNumberOfDailyFlightsFirstHalfDelay = 0;
        iNumberOfDailyFlightsInFlightAbort = 0;
        iNumberOfDailyFlightsOnTime = 0;
        iNumberOfDailyFlightsSecondHalfDelay = 0;
        iNumberOfDailyFlightsSucceed = 0;
        iNumberOfDailyFlightsTasked = 0;
    }
    int GetDayNumber();
    int GetNumberOfAircraftAwaitingPreFlightServicing();
    int GetNumberOfAircraftFlying();
    int GetNumberOfAircraftInPreFlightServicing();
    int GetNumberOfAircraftServiceable();
    int GetNumberOfAircraftUnserviceable();
    int GetNumberOfDailyFlightsCancelled();
    int GetNumberOfDailyFlightsFail();
    int GetNumberOfDailyFlightsFirstHalfDelay();
    int GetNumberOfDailyFlightsInFlightAbort();
```

```
    int GetNumberOfDailyFlightsOnTime();
    int GetNumberOfDailyFlightsSecondHalfDelay();
    int GetNumberOfDailyFlightsSucceed();
    int GetNumberOfDailyFlightsTasked();
    void SaveDailyAircraftStates(int iNumberOfAircraftServiceable,
        int iNumberOfAircraftFlying,int iNumberOfAircraftUnserviceable,
        int iNumberOfAircraftInPreFlightServicing,
        int iNumberOfAircraftAwaitingPreFlightServicing);
    void SaveDailyFlyingStats(int iNumberOfDailyFlightsTasked,
        int iNumberOfDailyFlightsOnTime,int iNumberOfDailyFlightsFirstHalfDelay,
        int iNumberOfDailyFlightsSecondHalfDelay,
        int iNumberOfDailyFlightsCancelled,int iNumberOfDailyFlightsInFlightAbort,
        int iNumberOfDailyFlightsFail,int iNumberOfDailyFlightsSucceed);
    ~CDailyResults(){}
};
//DailyResults.cpp :implementation of the CDailyResults class
#include "stdafx.h"
#include "DailyResults.h"
// CDailyResults commands
int CDailyResults::GetDayNumber()
{
    return iDayNumber;
}
int CDailyResults::GetNumberOfAircraftAwaitingPreFlightServicing()
{
    return iNumberOfAircraftAwaitingPreFlightServicing;
}
int CDailyResults::GetNumberOfAircraftFlying()
{
    return iNumberOfAircraftFlying;
}
int CDailyResults::GetNumberOfAircraftInPreFlightServicing()
{
    return iNumberOfAircraftInPreFlightServicing;
}
int CDailyResults::GetNumberOfAircraftServiceable()
{
    return iNumberOfAircraftServiceable;
}
int CDailyResults::GetNumberOfAircraftUnserviceable()
{
    return iNumberOfAircraftUnserviceable;
}
int CDailyResults::GetNumberOfDailyFlightsCancelled()
{
    return iNumberOfDailyFlightsCancelled;
}
int CDailyResults::GetNumberOfDailyFlightsFail()
{
    return iNumberOfDailyFlightsFail;
}
```

```
int CDailyResults::GetNumberOfDailyFlightsFirstHalfDelay()
{
    return iNumberOfDailyFlightsFirstHalfDelay;
}
int CDailyResults::GetNumberOfDailyFlightsInFlightAbort()
{
    return iNumberOfDailyFlightsInFlightAbort;
}
int CDailyResults::GetNumberOfDailyFlightsOnTime()
{
    return  iNumberOfDailyFlightsOnTime;
}
int CDailyResults::GetNumberOfDailyFlightsSecondHalfDelay()
{
    return iNumberOfDailyFlightsSecondHalfDelay;
}
int CDailyResults::GetNumberOfDailyFlightsSucceed()
{
    return iNumberOfDailyFlightsSucceed;
}
int CDailyResults::GetNumberOfDailyFlightsTasked()
{
    return iNumberOfDailyFlightsTasked;
}
void CDailyResults::SaveDailyAircraftStates(int iNumAircraftServ,int iNumAircraftFly,
    int iNumAircraftUnserv,int iNumAircraftPreFlight,
    int NumAircraftAwaitingPreFlight)
{
    iNumberOfAircraftServiceable += iNumAircraftServ;
    iNumberOfAircraftFlying += iNumAircraftFly;
    iNumberOfAircraftUnserviceable += iNumAircraftUnserv;
    iNumberOfAircraftInPreFlightServicing += iNumAircraftPreFlight;
    iNumberOfAircraftAwaitingPreFlightServicing += iNumAircraftAwaitingPreFlight;
}
void CDailyResults::SaveDailyFlyingStats(int iDailyFlightsTasked,
    int iDailyFlightsOnTime,int iDailyFlightsFirstHalfDelay,
    int iDailyFlightsSecondHalfDelay,int iDailyFlightsCancelled,
    int iDailyFlightsInFlightAbort,int iDailyFlightsFail,int iDailyFlightsSucceed)
{
    iNumberOfDailyFlightsTasked += iDailyFlightsTasked;
    iNumberOfDailyFlightsOnTime+= iDailyFlightsOnTime;
    iNumberOfDailyFlightsFirstHalfDelay += iDailyFlightsFirstHalfDelay;
    iNumberOfDailyFlightsSecondHalfDelay += iDailyFlightsSecondHalfDelay;
    iNumberOfDailyFlightsCancelled += iDailyFlightsCancelled;
    iNumberOfDailyFlightsInFlightAbort += iDailyFlightsInFlightAbort;
    iNumberOfDailyFlightsFail += iDailyFlightsFail;
    iNumberOfDailyFlightsSucceed += iDailyFlightsSucceed;
}
```

Delayed Flight

```
// DelayedFlights.h : interface of the CDelayedFlights class
```

```
class CDelayedFlight : public CObject
{
private:
//Attributes
    float fPlannedTakeOffTime;
    float fLastTakeOffTime;
    float fFlightDuration;
    int iNumberOfAircraftRequired;
public:
//Operations
    CDelayedFlight(){}
    CDelayedFlight(float fClockTime,int iNumberOfAircraft,
        float fFlightDelayMaximum,float fDuration)
    {
        iNumberOfAircraftRequired = iNumberOfAircraft;
        fPlannedTakeOffTime = fClockTime;
        fLastTakeOffTime = fClockTime + fFlightDelayMaximum;
        fFlightDuration = fDuration;
    }
    float GetLastTakeOffTime();
    float GetFlightDuration();
    int GetNumberOfAircraftRequired();
    ~CDelayedFlight(){}
};
// DelayedFlights.cpp : implementation of the CDelayedFlights class
#include "stdafx.h"
#include "DelayedFlight.h"
// CDelayedFlight commands
float CDelayedFlight::GetLastTakeOffTime()
{
    return fLastTakeOffTime;
}
float CDelayedFlight::GetFlightDuration()
{
    return fFlightDuration;
}
int CDelayedFlight::GetNumberOfAircraftRequired()
{
    return iNumberOfAircraftRequired;
}
```

DlgSimulationFinished

```
#if
!defined(AFX_DLGSIMULATIONFINISHED_H__19FF31CA_2D64_11D2_9F27_4445
5354616F__INCLUDED_)
#define
AFX_DLGSIMULATIONFINISHED_H__19FF31CA_2D64_11D2_9F27_44455354616
F__INCLUDED_
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
```

```
// DlgSimulationFinished.h : header file
// CDlgSimulationFinished dialog
class CDlgSimulationFinished : public CDialog
{
// Construction
public:
    CDlgSimulationFinished(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgSimulationFinished)
    enum { IDD = IDD_SIMULATIONFINISHEDDIALOG };
    float    m_MeanFlightsCancelled;
    float    m_MeanFlightsFirstHalfDelay;
    float    m_MeanFlightsOnTime;
    float    m_MeanFlightsSecondHalfDelay;
    float    m_MeanFlightsTasked;
    float    m_PercentageFlightsCancelled;
    float    m_PercentageFlightsFirstHalfDelay;
    float    m_PercentageFlightsOnTime;
    float    m_PercentageFlightsSecondHalfDelay;
    float    m_MeanFlightsFail;
    float    m_MeanFlightsInFlightAbort;
    float    m_MeanFlightsSucceed;
    float    m_MeanFlightsTakeOff;
    float    m_PercentageFlightsFail;
    float    m_PercentageFlightsInFlightAbort;
    float    m_PercentageFlightsSucceed;
    float    m_PercentageFlightsTakeOff;
    //}}AFX_DATA
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDlgSimulationFinished)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL
// Implementation
protected:
    // Generated message map functions
    //{{AFX_MSG(CDlgSimulationFinished)
    afx_msg void OnButtonExitSimulation();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
//{{AFX_INSERT_LOCATION}}
// Microsoft Developer Studio will insert additional declarations immediately before the
previous line.
#endif //
!defined(AFX_DLGSIMULATIONFINISHED_H__19FF31CA_2D64_11D2_9F27_4445
5354616F__INCLUDED_)
// DlgSimulationFinished.cpp : implementation file
#include "stdafx.h"
```

```
#include "ALSSim.h"
#include "DlgSimulationFinished.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
// CDlgSimulationFinished dialog
CDlgSimulationFinished::CDlgSimulationFinished(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgSimulationFinished::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgSimulationFinished)
    m_MeanFlightsCancelled = 0.0f;
    m_MeanFlightsFirstHalfDelay = 0.0f;
    m_MeanFlightsOnTime = 0.0f;
    m_MeanFlightsSecondHalfDelay = 0.0f;
    m_MeanFlightsTasked = 0.0f;
    m_MeanFlightsFail = 0.0f;
    m_MeanFlightsInFlightAbort = 0.0f;
    m_MeanFlightsSucceed = 0.0f;
    m_MeanFlightsTakeOff = m_MeanFlightsOnTime + m_MeanFlightsFirstHalfDelay +
        m_MeanFlightsSecondHalfDelay;
    m_PercentageFlightsCancelled = 0.0f;
    m_PercentageFlightsFirstHalfDelay = 0.0f;
    m_PercentageFlightsOnTime = 0.0f;
    m_PercentageFlightsSecondHalfDelay = 0.0f;
    m_PercentageFlightsFail = 0.0f;
    m_PercentageFlightsInFlightAbort = 0.0f;
    m_PercentageFlightsSucceed = 0.0f;
    m_PercentageFlightsTakeOff = 0.0f;
    //}}AFX_DATA_INIT
}
void CDlgSimulationFinished::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgSimulationFinished)
    DDX_Text(pDX, IDC_MEANFLIGHTSCANCELLED, m_MeanFlightsCancelled);
    DDX_Text(pDX, IDC_MEANFLIGHTSFIRSTHALFDELAY,
        m_MeanFlightsFirstHalfDelay);
    DDX_Text(pDX, IDC_MEANFLIGHTSONTIME, m_MeanFlightsOnTime);
    DDX_Text(pDX, IDC_MEANFLIGHTSSECONDHALFDELAY,
        m_MeanFlightsSecondHalfDelay);
    DDX_Text(pDX, IDC_MEANFLIGHTSTASKED, m_MeanFlightsTasked);
    DDX_Text(pDX, IDC_PERCENTAGEFLIGHTSCANCELLED,
        m_PercentageFlightsCancelled);
    DDX_Text(pDX, IDC_PERCENTAGEFLIGHTSFIRSTHALFDELAY,
        m_PercentageFlightsFirstHalfDelay);
    DDX_Text(pDX, IDC_PERCENTAGEFLIGHTSONTIME,
        m_PercentageFlightsOnTime);
    DDX_Text(pDX, IDC_PERCENTAGEFLIGHTSSECONDHALFDELAY,
        m_PercentageFlightsSecondHalfDelay);
```

```
    DDX_Text(pDX, IDC_MEANFLIGHTSFAILED, m_MeanFlightsFail);
    DDX_Text(pDX, IDC_MEANFLIGHTSINFLIGHTABORT,
        m_MeanFlightsInFlightAbort);
    DDX_Text(pDX, IDC_MEANFLIGHTSSUCCEED, m_MeanFlightsSucceed);
    DDX_Text(pDX, IDC_MEANFLIGHTSTAKEOFF, m_MeanFlightsTakeOff);
    DDX_Text(pDX, IDC_PERCENTAGEFLIGHTSFAILED, m_PercentageFlightsFail);
    DDX_Text(pDX, IDC_PERCENTAGEFLIGHTSINFLIGHTABORT,
        m_PercentageFlightsInFlightAbort);
    DDX_Text(pDX, IDC_PERCENTAGEFLIGHTSSUCCEED,
        m_PercentageFlightsSucceed);
    DDX_Text(pDX, IDC_PERCENTAGEFLIGHTSTAKEOFF,
        m_PercentageFlightsTakeOff);
    //}}AFX_DATA_MAP
}
BEGIN_MESSAGE_MAP(CDlgSimulationFinished, CDialog)
    //{{AFX_MSG_MAP(CDlgSimulationFinished)
    ON_BN_CLICKED(IDC_BUTTON_EXITSIMULATION, OnButtonExitSimulation)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
// CDlgSimulationFinished message handlers
void CDlgSimulationFinished::OnButtonExitSimulation()
{
    // Same as selecting the x on the top right
    ASSERT(AfxGetMainWnd() != NULL);
    AfxGetMainWnd()->SendMessage(WM_CLOSE);
}
```

Event

```
// Event.h : interface of the CEvent class
class CEvent : public CObject
{
private:
//Attributes
    float fEventDuration;
    float fEventTime;
    float fPreviousEventTime;
    int iAircraftNumber;
    int iEventNumber;
    int iLRIPosition;
    int iLRIType;
    int iNumberOfAircraft;
    int iNumberOfLRIs;
public:
//Operations
    CEvent(){}
    CEvent(int iEvent,float fTime,float fPreviousTime = 0,int iTailNumber = 0,
        int iNumber = 0,int iPosition = 0,int iType = 0,int iNumberLRI = 0,
        float fDuration = 0.0)
    {
        iEventNumber = iEvent;
        fEventTime = fTime;
```

```
        fPreviousEventTime = fPreviousTime;
        iAircraftNumber = iTailNumber;
        iNumberOfAircraft = iNumber;
        iLRIPosition = iPosition;
        iLRIType = iType;
        iNumberOfLRIs = iNumberLRI;
        fEventDuration = fDuration;
    }
    float GetEventDuration();
    float GetEventTime();
    float GetPreviousEventTime();
    int GetAircraftNumber();
    int GetEventNumber();
    int GetLRIPosition();
    int GetLRIType();
    int GetNumberOfAircraft();
    int GetNumberOfLRIs();
    ~CEvent(){}
};
//Event.cpp :implementation of the CEvent class
include "stdafx.h"
#include "Event.h"
/ CEvent commands
float CEvent::GetEventDuration()
{
    return fEventDuration;
}
float CEvent::GetEventTime()
{
    return fEventTime;
}
float CEvent::GetPreviousEventTime()
{
    return fPreviousEventTime;
}
int CEvent::GetAircraftNumber()
{
    return iAircraftNumber;
}
int CEvent::GetEventNumber()
{
    return iEventNumber;
}
int CEvent::GetLRIPosition()
{
    return iLRIPosition;
}
int CEvent::GetLRIType()
{
    return iLRIType;
}
```

```
int CEvent::GetNumberOfAircraft()
{
    return iNumberOfAircraft;
}
int CEvent::GetNumberOfLRIs()
{
    return iNumberOfLRIs;
}
```

Random Number

```
//RandomNumber.h interface of the CRandomNumber class
#ifndef H_CLCG4_H
#define H_CLCG4_H
#define Maxgen  100
typedef  unsigned short int Gen;
typedef  enum {InitialSeed, LastSeed, NewSeed}  SeedType;
class CRandomNumber : public CObject
{
//Attributes
//Operations
public:
    CRandomNumber()
    {
        Init(31,41);
    };
    void Init (long v, long w);
    void InitDefault ();
    void SetInitialSeed (long s[4]);
    void InitGenerator (Gen g, SeedType Where);
    void SetSeed (Gen g, long s[4]);
    void GetState (Gen g, long s[4]);
    void WriteState (Gen g);
    void SetNewRandomNumberSeed();
    double GenVal (Gen g);
    ~CRandomNumber(){}
};
#endif
//RandomNumber.cpp :implementation of the CRandomNumber class
#include "stdafx.h"
#include "RandomNumber.h"
// CRandomNumber commands
/*****************************************************************/
// Private part.                                                 */
/*****************************************************************/
#define H   32768           /* = 2^15 : use in MultModM.          */
static long aw[4], avw[4],     /*  a[j]^{2^w} et a[j]^{2^{v+w}}.    */
    a[4] = { 45991, 207707, 138556, 49689 },
    m[4] = { 2147483647, 2147483543, 2147483423, 2147483323 };
static long Ig[4][Maxgen+1], Lg[4][Maxgen+1], Cg[4][Maxgen+1];
    /* Initial seed, previous seed, and current seed. */
```

```
static short i, j;
static long MultModM (long s, long t, long M)
// Returns (s*t) MOD M.  Assumes that -M < s < M and -M < t < M.
// See L'Ecuyer and Cote (1991).
{
    long R, S0, S1, q, qh, rh, k;
    if (s < 0)
        s += M;
    if (t < 0)
        t += M;
    if (s < H)
    {
        S0 = s;
        R = 0;
    }
    else
    {
        S1 = s/H;
        S0 = s - H*S1;
        qh = M/H;
        rh = M - H*qh;
        if (S1 >= H)
        {
            S1 -= H;   k = t/qh;
            R = H * (t - k*qh) - k*rh;
        while (R < 0)
            R += M;
        }
        else R = 0;
        if (S1 != 0)
        {
            q = M/S1;   k = t/q;
            R -= k * (M - S1*q);
            if (R > 0)
                R -= M;
            R += S1*(t - k*q);
            while (R < 0)
                R += M;
        }
        k = R/qh;
        R = H * (R - k*qh) - k*rh;
        while (R < 0)
            R += M;
    }
    if (S0 != 0)
    {
        q = M/S0;
        k = t/q;
        R -= k* (M - S0*q);
        if (R > 0)
            R -= M;
```

```
            R += S0 * (t - k*q);
            while (R < 0)
                R += M;
        }
    return R;
    }
    /*-----------------------------------------------------------------*/
    /* Public part.                                                    */
    /*-----------------------------------------------------------------*/
    void CRandomNumber::InitGenerator (Gen g, SeedType Where)
    {
        if (g > Maxgen) printf ("ERROR: InitGenerator with g > Maxgen \n");
        for (j = 0; j < 4; j++)
        {
            switch (Where)
            {
                case InitialSeed :
                    Lg [j][g] = Ig [j][g];
                    break;
                case NewSeed :
                    Lg [j][g] = MultModM (aw [j], Lg [j][g], m [j]);
                    break;
                case LastSeed :
                    break;
            }
            Cg [j][g] = Lg [j][g];
        }
    }
    void CRandomNumber::SetNewRandomNumberSeed()
    {
        Gen g;
        for (g = 1; g<= Maxgen;g++)
        {
            InitGenerator(g,NewSeed);
        }
    }
    void CRandomNumber::SetSeed (Gen g, long s[4])
    {
        if (g > Maxgen) printf ("ERROR: SetSeed with g > Maxgen \n");
        for (j = 0; j < 4; j++) Ig [j][g] = s [j];
        InitGenerator (g, InitialSeed);
    }
    void CRandomNumber::WriteState (Gen g)
    {
        printf ("\n State of generator g = %u :", g);
        for (j = 0; j < 4; j++)
        {
            printf ("\n   Cg[%u] = %lu", j, Cg[j][g]);
        }
        printf ("\n");
    }
```

```
void CRandomNumber::GetState (Gen g, long s[4])
{
    for (j = 0; j < 4; j++)  s [j] = Cg [j][g];
}
void CRandomNumber::SetInitialSeed (long s[4])
{
    Gen g;
    for (j = 0; j < 4; j++)
        Ig [j][0] = s [j];
    InitGenerator (0, InitialSeed);
    for (g = 1; g <= Maxgen; g++)
    {
        for (j = 0; j < 4; j++)
            Ig [j][g] = MultModM (avw [j], Ig [j][g-1], m [j]);
        InitGenerator (g, InitialSeed);
    }
}
void CRandomNumber::Init (long v, long w)
{
    long sd[4] = {11111111, 22222222, 33333333, 44444444};
    for (j = 0; j < 4; j++)
    {
        aw [j] = a [j];
        for (i = 1; i <= w; i++)
            aw [j]  = MultModM (aw [j], aw [j], m[j]);
        avw [j] = aw [j];
        for (i = 1; i <= v; i++)
            avw [j] = MultModM (avw [j], avw [j], m[j]);
    }
    SetInitialSeed (sd);
}
double CRandomNumber::GenVal (Gen g)
{
    long k,s;
    double u;
    u = 0.0;
    if (g > Maxgen)
        printf ("ERROR: Genval with g > Maxgen \n");
    s = Cg [0][g];
    k = s / 46693;
    s = 45991 * (s - k * 46693) - k * 25884;
    if (s < 0) s = s + 2147483647;
    Cg [0][g] = s;
    u = u + 4.65661287524579692e-10 * s;
    s = Cg [1][g];
    k = s / 10339;
    s = 207707 * (s - k * 10339) - k * 870;
    if (s < 0)
        s = s + 2147483543;
    Cg [1][g] = s;
```

```
    u = u - 4.65661310075985993e-10 * s;
    if (u < 0)
        u = u + 1.0;
    s = Cg [2][g];
    k = s / 15499;
    s = 138556 * (s - k * 15499) - k * 3979;
    if (s < 0)
        s = s + 2147483423;
    Cg [2][g] = s;
    u = u + 4.65661336096842131e-10 * s;
    if (u >= 1.0)
        u = u - 1.0;
    s = Cg [3][g];
    k = s / 43218;
    s = 49689 * (s - k * 43218) - k * 24121;
    if (s < 0)
        s = s + 2147483323;
    Cg [3][g] = s;
    u = u - 4.65661357780891134e-10 * s;
    if (u < 0)
        u = u + 1.0;
    return (u);
}
void CRandomNumber::InitDefault ()
{
    Init (31, 41);
}
```

Simulation Progress Bar

```
#if !defined(AFX_SBARSIMPROGRESS_H__C665B001_E38A_11D1_A97A_
    44455354616F__INCLUDED_)
#define
AFX_SBARSIMPROGRESS_H__C665B001_E38A_11D1_A97A_44455354616F__
    INCLUDED_
#if _MSC_VER >= 1000
#pragma once
#endif // _MSC_VER >= 1000
// SBarSimProgress.h : header file
    const int PROGRESS_CTRL_CX = 300;//160;
    const int X_MARGIN = 5; // X value used for margins and control spacing
    const int Y_MARGIN = 2; // Y value used for margins and control spacing
// CSimProgressStatusBar window
class CSimProgressStatusBar : public CStatusBar
{
// Construction
public:
    CSimProgressStatusBar();
// Attributes
public:
protected:
    bool m_bSimProgressMode;
```

```
    CProgressCtrl m_SimProgressCtrl;
    Cstatic m_SimProgressLabel;
    int m_iSimProgressCtrlWidth;
// Operations
public:
    CProgressCtrl* GetProgressCtrl() {return &m_SimProgressCtrl;}
    void RecalcSimProgressDisplay();
    void SetSimProgressCtrlWidth(UINT nWidth = PROGRESS_CTRL_CX);
    void SetSimProgressLabel (LPCSTR lpszSimProgressLabel);
    void ShowSimProgressDisplay(bool bShow = true);
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSimProgressStatusBar)
    //}}AFX_VIRTUAL
// Implementation
public:
    virtual ~CSimProgressStatusBar();
    // Generated message map functions
protected:
    //{{AFX_MSG(CSimProgressStatusBar)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnPaint();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
//{{AFX_INSERT_LOCATION}}
#endif // !defined(AFX_SBARSIMPROGRESS_H__C665B001_E38A_11D1_A97A_
    44455354616F__INCLUDED_)
// SBarSimProgress.cpp : implementation file
#include "stdafx.h"
#include "ALSSim.h"
#include "SBarSimProgress.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
// CSimProgressStatusBar
CSimProgressStatusBar::CSimProgressStatusBar()
{
    m_bSimProgressMode = false;
    m_iSimProgressCtrlWidth = PROGRESS_CTRL_CX;
}
CSimProgressStatusBar::~CSimProgressStatusBar()
{
}
BEGIN_MESSAGE_MAP(CSimProgressStatusBar, CStatusBar)
    //{{AFX_MSG_MAP(CSimProgressStatusBar)
    ON_WM_CREATE()
    ON_WM_PAINT()
    //}}AFX_MSG_MAP
```

```
END_MESSAGE_MAP()
// CSimProgressStatusBar Commands
// Set width of the status bar
void CSimProgressStatusBar::RecalcSimProgressDisplay()
{
    // Adjust the positions of the Label and Progress Controls
    // Place the Label Control to the left of the
    // Progress Control
    //  Label Text [Progress control]
    CRect ControlRect;
    CRect ClientRect;
    GetClientRect(&ClientRect);
    ControlRect = ClientRect;
    //Set up Text Label using the rest of the status bar area
    ControlRect.left += X_MARGIN ;
    ControlRect.right = ControlRect.left + 110;
    ControlRect.top += Y_MARGIN;
    ControlRect.bottom -= Y_MARGIN;
    m_SimProgressLabel.MoveWindow(ControlRect, false);
    // Set up the Progress Bar
    ControlRect.left = ControlRect.right + X_MARGIN;
    ControlRect.right = ControlRect.left + m_iSimProgressCtrlWidth;
    m_SimProgressCtrl.MoveWindow(ControlRect, false);
}
// Set width of the status bar
void CSimProgressStatusBar::SetSimProgressCtrlWidth(UINT nWidth)
{
    m_iSimProgressCtrlWidth = nWidth;
}
void CSimProgressStatusBar::SetSimProgressLabel (LPCSTR lpszSimProgressLabel)
{
    m_SimProgressLabel.SetWindowText(lpszSimProgressLabel);

    // If displaying progress, update
    // placement of label and progress control
    if(m_bSimProgressMode)
    {
        RecalcSimProgressDisplay();
        Invalidate();
        UpdateWindow();
    }
}
void CSimProgressStatusBar::ShowSimProgressDisplay(bool bShow)
{
    m_bSimProgressMode = bShow;
    if(m_bSimProgressMode)
    {
        RecalcSimProgressDisplay();
    }
    m_SimProgressLabel.ShowWindow(m_bSimProgressMode ?
        SW_SHOW:SW_HIDE);
```

```
    m_SimProgressCtrl.ShowWindow(m_bSimProgressMode ? SW_SHOW :SW_HIDE);
    Invalidate();
    UpdateWindow();
}
// CProgressStatusBar message handlers
int CSimProgressStatusBar::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CStatusBar::OnCreate(lpCreateStruct) == -1)
        return -1;
    //  Create the Progress Control, size and position will be calculated
    // later from ShowSimProgressDisplay() call
    if(!m_SimProgressCtrl.Create(0,  // Style - Don't Show Position or Percent
        CRect(0,0,0,0),     // Initial position
        this,    // Parent
        0))      // Child ID
    {
        return -1;
    }
    //  Create the Progress Label - we'll calculate its size and
    //  position later - in response to a ShowProgressDisplay() call.
    if (!m_SimProgressLabel.Create( NULL,      // Text
        WS_CHILD|SS_LEFT,    // Style
        CRect(0,0,0,0),     //Initial Position
        this))   // Parent
    {
        return -1;
    }
    // Use the same font as the Status Bar
    m_SimProgressLabel.SetFont(GetFont());
    return 0;
}
void CSimProgressStatusBar::OnPaint()
{
    CPaintDC dc(this); // device context for painting
    // If displaying the Progress Control we need to handle the
    // painting of the Status Bar,otherwise use the base class
    if(!m_bSimProgressMode)
    {
        CStatusBar::OnPaint();
    }
}
```

Stock

```
// Stock.h : interface of the CStock class
class CStock : public CObject
{
private:
//Attributes
    bool bLRIRepairedOnUnit;
    float fLRIDepotRepairTime;
    float fLRIMeanFailureTime;
```

```
    float fLRIProportionRepairedAtUnit;
    float fLRIRefitTime;
    float fLRIRemovalTime;
    float fLRIUnitRepairTime;
    int iLRIDepotAllocatedStock;
    int iLRIDepotServiceableStock;
    int iLRIDepotStartingStock;
    int iLRIEssential;
    int iLRINumberRequiredForAircraft;
    int iLRIType;
    int iLRIUnitServiceableStock;
    int iLRIUnitUnserviceableStock;
    int iLRIUnitStartingStock;
public:
//Operations
    CStock(){}
    CStock(int iType,float fFailure,int iEssential,float fRemoval,float fRefit,
        float fProportionRepairedAtUnit,float fUnitRepairTime,
        float fDepotRepairTime,int iUnitStock,int iDepotStock)
    {
        iLRIDepotStartingStock = iDepotStock;
        iLRIDepotServiceableStock = iDepotStock;
        iLRIType = iType;
        iLRIEssential = iEssential;
        iLRIUnitStartingStock = iUnitStock;
        iLRIUnitServiceableStock = iUnitStock;
        fLRIDepotRepairTime = fDepotRepairTime;
        fLRIMeanFailureTime = fFailure;
        fLRIProportionRepairedAtUnit = fProportionRepairedAtUnit;
        fLRIRefitTime = fRefit;
        fLRIRemovalTime = fRemoval;
        fLRIUnitRepairTime = fUnitRepairTime;
        // Zero the stock counters
        iLRIDepotAllocatedStock = 0;
        iLRINumberRequiredForAircraft = 0;
        iLRIUnitUnserviceableStock = 0;
    }
    bool LRIRepairedOnUnit(float fLRIProportion);
    bool MoreLRIsRequiredAtUnit();
    float GetDepotRepairTime();
    float GetMeanFailureTime();
    float GetRemovalTime();
    float GetReplacementTime();
    float GetUnitRepairTime();
    int GetDepotAllocatedStock();
    int GetDepotServiceableStock();
    int GetLRIEssentiality();
    int GetNumberRequiredForAircraft();
    int GetStockType();
    int GetUnitServiceableStock();
    int GetUnitUnserviceableStock();
```

```
    void IncreaseDepotAllocatedStock();
    void IncreaseDepotServiceableStock();
    void IncreaseNumberRequiredForAircraft();
    void IncreaseUnitServiceableStock(int iNumber);
    void IncreaseUnitUnserviceableStock();
    void ReduceDepotServiceableStock();
    void ReduceNumberRequiredForAircraft();
    void ReduceUnitServiceableStock();
    void ResetStock();
    void ZeroDepotAllocatedStock();
    void ZeroUnitUnserviceableStock();
    ~CStock(){}
};
// Stock.cpp : implementation of the CStock class
#include "stdafx.h"
#include "Stock.h"
// CStock commands
bool CStock::LRIRepairedOnUnit(float fLRIProportion)
{
    // Identify whether LRI repaired on the unit or at the depot
    if (fLRIProportion <= fLRIProportionRepairedAtUnit)
    {
        bLRIRepairedOnUnit = true;
    }
    else
    {
        bLRIRepairedOnUnit = false;
    }
    return bLRIRepairedOnUnit;
}
bool CStock::MoreLRIsRequiredAtUnit()
{
    if (iLRIUnitServiceableStock < iLRIUnitStartingStock)
        return true;
    else
        return false;
}
float CStock::GetDepotRepairTime()
{
    return fLRIDepotRepairTime;
}
float CStock::GetMeanFailureTime()
{
    return fLRIMeanFailureTime;
}
float CStock::GetRemovalTime()
{
    return fLRIRemovalTime;
}
float CStock::GetReplacementTime()
{
```

```
    return fLRIRefitTime;
}
float CStock::GetUnitRepairTime()
{
    return fLRIUnitRepairTime;
}
int CStock::GetDepotAllocatedStock()
{
    return  iLRIDepotAllocatedStock;
}
int CStock::GetDepotServiceableStock()
{
    return iLRIDepotServiceableStock;
}
int CStock::GetLRIEssentiality()
{
    return iLRIEssential;
}
int CStock::GetNumberRequiredForAircraft()
{
    return iLRINumberRequiredForAircraft;
}
int CStock::GetStockType()
{
    return iLRIType;
}
int CStock::GetUnitServiceableStock()
{
    return iLRIUnitServiceableStock;
}
int CStock::GetUnitUnserviceableStock()
{
    return iLRIUnitUnserviceableStock;
}
void CStock::IncreaseDepotAllocatedStock()
{
    iLRIDepotAllocatedStock++;
}
void CStock::IncreaseDepotServiceableStock()
{
    iLRIDepotServiceableStock++;
}
void CStock::IncreaseNumberRequiredForAircraft()
{
    iLRINumberRequiredForAircraft++;
}
void CStock::IncreaseUnitServiceableStock(int iNumber)
{
    iLRIUnitServiceableStock+= iNumber;
}
void CStock::IncreaseUnitUnserviceableStock()
```

```
{
    iLRIUnitUnserviceableStock++;
}
void CStock::ReduceDepotServiceableStock()
{
    iLRIDepotServiceableStock--;
}
void CStock::ReduceNumberRequiredForAircraft()
{
    iLRINumberRequiredForAircraft--;
}
void CStock::ReduceUnitServiceableStock()
{
    iLRIUnitServiceableStock--;
}
void CStock::ResetStock()
{
    iLRIDepotServiceableStock = iLRIDepotStartingStock;
    iLRIUnitServiceableStock = iLRIUnitStartingStock;
    iLRIDepotAllocatedStock = 0;
    iLRIUnitUnserviceableStock = 0;
    iLRINumberRequiredForAircraft = 0;
}
void CStock::ZeroDepotAllocatedStock()
{
    iLRIDepotAllocatedStock = 0;
}
void CStock::ZeroUnitUnserviceableStock()
{
    iLRIUnitUnserviceableStock = 0;
}
```

APPENDIX TWO RESULTS OF THE TESTING OF THE ALSSIM RANDOM NUMBER GENERATOR

This Appendix deals with the tests applied to the ALSSim pseudo Random number generator. For these tests a total of 4,000 numbers were generated lying in the range $0 \leq x \leq 1$ and the following series of tests applied to them. The results of the tests and the analysis follows this descriptive section.

Scatter Chart

The purpose of a scatter chart is to attempt to identify if there are any clearly discernible patterns which would point to a lack of randomness within the generator. This is accomplished by taking a line from the current number on the x axis and one from the previous number on the y axis and plotting a point where the 2 intersect.

Summary Table

Although not directly a test in its own right a summary sheet detailing the results of the various tests is included at this stage for ease of examination. A summary of the X^2 statistic calculated from the test results detailed below and the X^2 0.95 quantile for the appropriate degrees of freedom. This is followed by a brief statement identifying any tests which have failed. The data underpinning this summary is included afterwards for each group of numbers.

Frequency Test

In the frequency test we are attempting to ascertain whether the numbers are evenly distributed over the range $0 \leq x \leq 1$ and thus confirming that the generator produces an acceptable approximation to a normal distribution.

Poker Test

In a 5 card poker test the number produced are converted into integers in the range $0 \leq x \leq 9$ and divided into 800 groups of 5 numbers each. The frequency of the occurrence of particular groups of numbers is then examined against the known probabilities of these sequences occurring in 5 card poker.

Gap Test

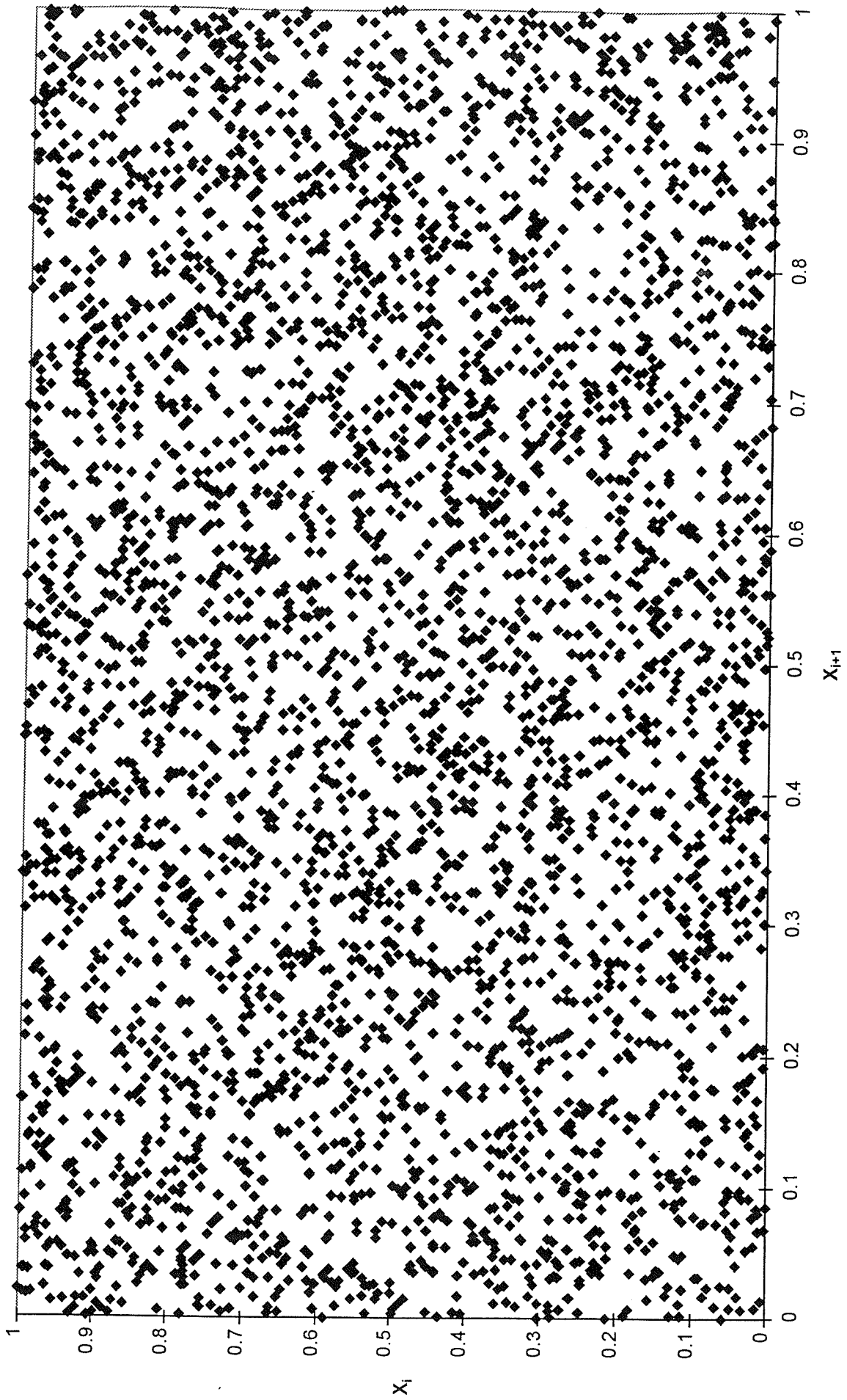
As with the poker test the gap test uses the numbers converted into integers in the range $0 \leq x \leq 9$ and then examines the length of the gap between occurrences of a particular number that occurs frequently, in this case 9, and compares this with the expected frequency.

Distribution of Pairs Test

This test examines the number of pairs found within the group of numbers being examined. The numbers are converted into integers in the range $0 \leq x \leq 9$ and the group is then divided into 400 groups of 10 numbers each. The number of pairs within each sub-group is counted and the findings compared with the theoretical expectation.

Frequency of Pairs Test

The frequency of pairs test uses the numbers converted into integers in the range $0 \leq x \leq 9$ and tests for pairs of numbers over the entire rage of 4000 numbers. The frequency of the occurrence of each pair is compared with the theoretical expectation.


1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
x_i
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
x_{i+1}

Summary

Test	Degrees of Freedom	C^2 Test Result	C^2 95% from Table
Frequency	9	11.406	16.92
Poker	4	2.347	9.488
Gap	10	4.140	18.307
Distribution of Pairs	9	12.183	16.92
Frequency of Pairs	3	6.231	7.815

Examination of the test results for this group of 4000 pseudorandom numbers against a null hypothesis shows that in all cases the C^2statistic lies inside the 95% region of acceptability. Thus this generator produces numbers that can be considered to be sufficiently random to be used within the simulation.

Frequency Test

Range	Actual	Expected	C^2
0.1	410	400	0.25
0.2	365	400	3.063
0.3	364	400	3.24
0.4	424	400	1.44
0.5	400	400	0
0.6	424	400	1.44
0.7	399	400	0.003
0.8	404	400	0.04
0.9	386	400	0.49
1	424	400	1.44
Total	4000	4000	11.406

Poker Test

Hand	Observed Actual	Observed Expected	Grouped Actual	Grouped Expected	C^2
All different	238	241.92	238	241.92	0.064
1 Pair	419	403.2	419	403.2	0.7
2 Pairs	76	86.4	76	86.4	1.252
3 of a kind	57	57.6	57	57.6	0.006
Full House	5	7.2	9	10.88	0.325
4 of a kind	3	3.6			
5 of a kind	1	0.08			
Total	799	800	799	800	2.347

Gap Test

Gap	Actual	Expected	C^2
1	45	42.4	0.159
2	33	38.16	0.698
3	32	34.34	0.159
4	39	30.91	2.117
5	29	27.82	0.05
6	29	25.04	0.626
7	22	22.53	0.012
8	19	20.28	0.081
9	18	18.25	0.003
≥10	158	164.27	0.239
Total	424	424	4.144

Distribution of Pairs Test

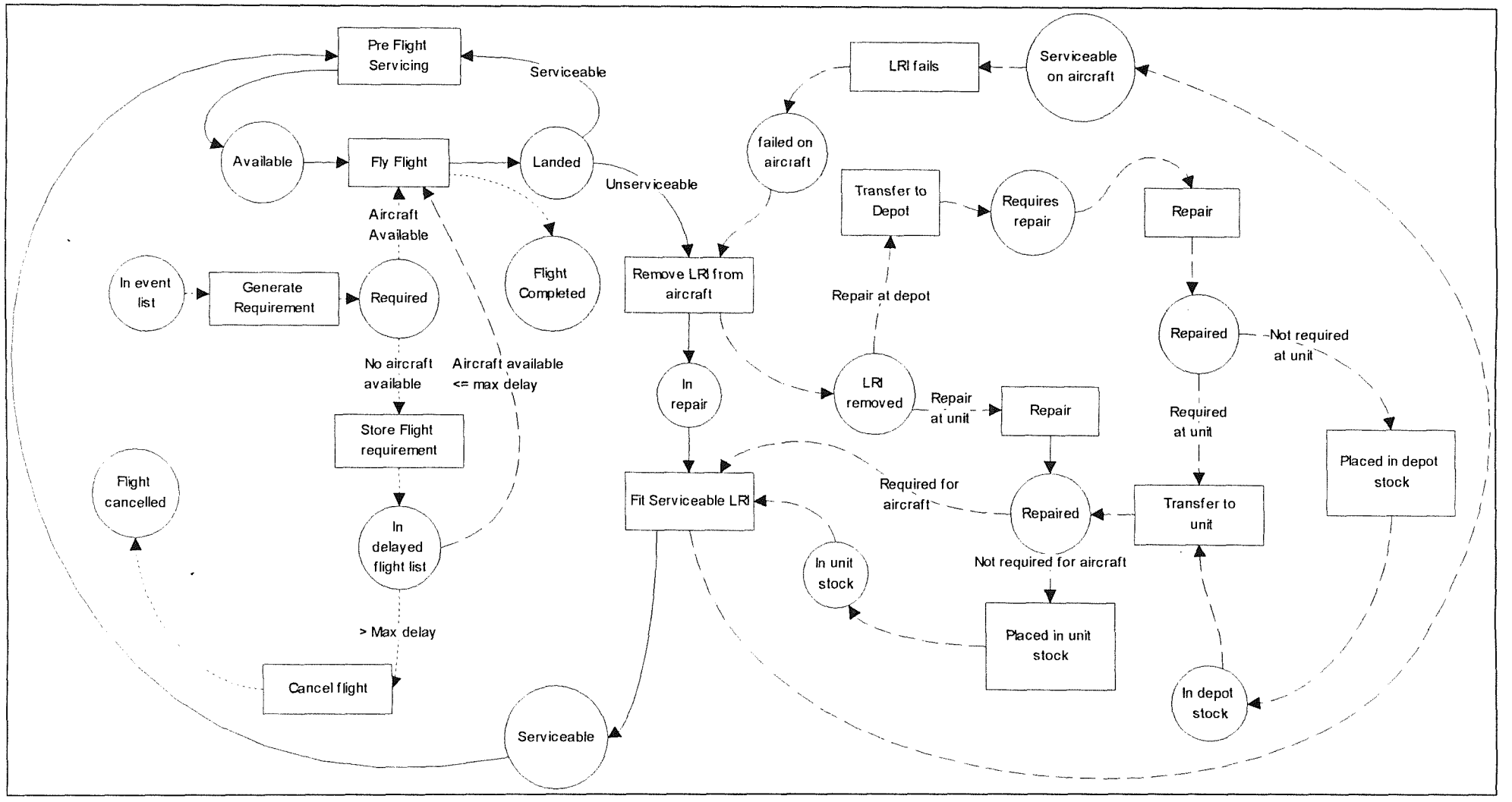
Pair	Actual	Expected	C^2
0	33	33	0
11	26	33	1.485
22	28	33	0.758
33	34	33	0.03
44	45	33	4.364
55	26	33	1.485
66	28	33	0.758
77	39	33	1.091
88	30	33	0.273
99	41	33	1.939
Total	330	330	12.183

Frequency of Pairs Test

	Observed		Grouped		
Number	Actual	Expected	Actual	Expected	C^2
0	169	154.97	169	154.96	1.270
1	149	170.29	149	170.28	2.662
2	67	64.49	67	64.48	0.098
3	13	9.77	15	10.25	2.201
4	2	0.48			
5	0	0			
Total	400	400	400	399.97	6.231

APPENDIX THREE ALSSIM ACTIVITY CYCLE DIAGRAM

This appendix shows the total activity diagram for the three entities contained within ALSSim. Each of the entities; mission, aircraft and LRI have been discussed independently within the main text and were drawn in figures 4 to 6. The purpose of this Appendix is to draw the 3 figures together to allow the interaction of the three entity types to be clearly seen.


Pre Flight Servicing
Serviceable
Available
Fly Flight
Landed
Unserviceable
Aircraft Available
Flight Completed
In event list
Generate Requirement
Required
No aircraft available
Aircraft available <= max delay
Store Flight requirement
Flight cancelled
In delayed flight list
> Max delay
Cancel flight
Serviceable
Remove LRI from aircraft
In repair
Fit Serviceable LRI
LRI fails
Serviceable on aircraft
failed on aircraft
Transfer to Depot
Requires repair
Repair
Repair at depot
LRI removed
Repair at unit
Repair
Repaired
Not required at unit
Required at unit
Placed in depot stock
Transfer to unit
Required for aircraft
Repaired
Not required for aircraft
In unit stock
Placed in unit stock
In depot stock

GLOSSARY OF TERMS

Backorder	A term used to describe a hole awaiting a spare. An aircraft subject to a backorder is unserviceable until such time as the LRI is received and fitted.
Depot	A combined repair and storage location for LRIs
Failure Time	The time taken for a component fitted to an aircraft to fail in use.
First Line	The operational Squadron for the aircraft.
Fourth Line	The location within industry that removed LRIs are dispatched to for deep repairs.
Mission	A number of flights that are required to be flown at the same time..
Mission Duration	The length of a mission. Adding this value to the achieved take off time gives the landing time.
Mission Success Rate	The proportion of missions that are launched within the acceptable window.
Pipeline	The route between the operational and repair locations through which defective components and spares pass.
Scale	The number of each type of spare required to support system availability targets. The scale is derived by using a deterministic scaling application.
Second Line	The component servicing bays located on the operational unit.
Serviceable	The term used to describe a LRI that is working.

Shelf Satisfaction Rate	The percentage of items satisfied from shelf stock without the need for a backorder.
Squadron	The operational organisation which operate the aircraft.
System Availability	The proportion of the total modelled time that a given system can be expected to be available for use.
Technician	RAF Maintenance Personnel.
Third Line	The location within the RAF that removed components are dispatched to for deep repairs.
Unserviceable	Term used to describe a LRI that has failed.

BIBLIOGRAPHY

Banks J Ed., *Handbook of Simulation*, John Wiley and Sons, 1998

Conolly B, *Techniques in Operational Research Volume 2: Models Search and Randomization*, Ellis Horwood,1981.

Dye P, 'The Royal Flying Corps Logistic Organisation', *Air Power Review Autumn 1998.*

Kleijnen J and van Groenendaal W, *Simulation a Statistical Perspective*, John Wiley and Sons, 1992.

Knuth D E, *The Art of Computer Programming Vol 2: Seminumerical Algorithms*, Adison-Wesley,1981.

Kobayashi H, *Modeling and Analysis: An introduction to System Performance Evaluation Methodology*, Adison-Wesley, 1978.

L'Ecuyer P and Andres T H, *A Random Number Generator Based on the Combination of Four LCGs*, http://www.iro.umontreal.ca/~lecuyer/papers.htm.

Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill,1991.

Marsaglia G, *Marsaglia's Random Number CDROM*, http://stat.fsu.edu/~geo/

Morgan B J T, *Elements of Simulation*, Chapman and Hall, 1984.

Pidd M, *Computer Simulation in Management Science*, John Wiley and Sons, 1992.

Rand Corporation, *A Million Random Digits with 100,000 Normal Deviates*, Free Press, 1955

Rumbaugh J, Blaha M, Premerlani W, Eddy F and Lorenson W, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Sherbrooke C C, *Optimal Inventory Modeling of Systems*, John Wiley and Sons, 1992.

LIST OF REFERENCES

1 Dye P, 'The Royal Flying Corps Logistic Organisation', *Air Power Review Autumn 1998* pp 42-58.

2 Sherbrooke C C, *Optimal Inventory Modeling of Systems*, John Wiley and Sons, 1992 pp 9-11.

3 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 pp 106-107.

4 Sherbrooke C C, *Optimal Inventory Modeling of Systems*, John Wiley and Sons, 1992 pp 9-12.

5 Rand Corporation, *A Million Random Digits with 100,000 Normal Deviates*, Free Press, 1955.

6 Marsaglia G, *Marsaglia's Random Number CDROM*, http://stat.fsu.edu/~geo/

7 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 pp 423-424.

8 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 pp 424-431.

9 Pidd M, *Computer Simulation in Management Science*, John Wiley and Sons, 1992 pp 196-200.

10 Banks J Ed., *Handbook of Simulation*, John Wiley and Sons, 1998 pp 94 - 95, p104.

11 Knuth D E, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Second Edition*, Addison-Wesley, 1981 pp 16-18.

12 Banks J Ed., *Handbook of Simulation*, John Wiley and Sons, 1998 pp 102-106.

13 L'Ecuyer P and Andres T H, *A Random Number Generator Based on the Combination of Four LCGs*, http://www.iro.umontreal.ca/~lecuyer/papers.htm, paper 39.

14 Pidd M, *Computer Simulation in Management Science*, John Wiley and Sons, 1992 pp 227-228.

15 Morgan B J T, *Elements of Simulation*, Chapman and Hall, 1984 pp 139-148.

16 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 pp 436-447.

17 Conolly B, *Techniques in Operational Research Volume 2: Models Search and Randomization*, Ellis Horwood,1981 pp 212-220.

18 Kleijnen J and van Groenendaal W, *Simulation a Statistical Perspective*, John Wiley and Sons, 1992 pp 22-28.

19 L'Ecuyer P and Andres T H, *A Random Number Generator Based on the Combination of Four LCGs*, http://www.iro.umontreal.ca/~lecuyer/papers.htm, paper 39.

20 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 pp 332-350.

21 Banks J Ed., *Handbook of Simulation*, John Wiley and Sons, 1998 pp 150-159.

22 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 p333.

23 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 p337.

24 Pidd M, *Computer Simulation in Management Science*, John Wiley and Sons, 1992 pp 31-46.

25 Rumbaugh J, Blaha M, Premerlani W, Eddy F and Lorenson W, *Object-Oriented Modeling and Design*, Prentice Hall, 1991 pp 173-179.

26 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 pp 302-306.

27 Kleijnen J and van Groenendaal W, *Simulation a Statistical Perspective*, John Wiley and Sons, 1992 pp 205-207.

28 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 pg 288.

29 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 pg 738 Table T.1

30 Law A M and Kelton W D, *Simulation Modeling and Analysis*, McGraw-Hill, 1991 pp 297-288.