A SIMULATION METHODOLOGY FOR CONTINUOUS SYSTEMS

by

Niels Stchedroff

MATHEMATICS DEPARTMENT

OF THE

UNIVERSITY OF SOUTHAMPTON

MPHIL THESIS

SEPTEMBER 2009

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

SCHOOL OF MATHEMATICS

Master of Philosophy

By Niels Stchedroff

This paper discusses the problem of modelling a continuous supply chain efficiently. Some existing modelling systems have poor performance, severely limiting their utility. The core of this work is the design, implementation and testing of a more efficient computational pattern that is claimed to improve performance. While the problem is apparently continuous, analysis suggests that this problem can be modelled using an adaption of discrete techniques. A pattern involving a modification of the Three Phase Approach discrete-event simulation technique was developed. Analysis of the way in which the effects of an event spread within the system modelled leads to a method by which excessive re-calculation can be avoided, yielding a model that is computationally more efficient. The pattern is then used in the investigation of automated design of the structure of the supply chain. The production, processing, transportation and consumption of Liquid National Gas (LNG) and the associated products form a complex supply chain and were selected as the example problem to be the subject of this work. The results demonstrate a high level of performance – sufficient speed to make experimentation with supply chain structure problems, with a real world level of complexity, practical.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Acknowledgements

This thesis is dedicated to the memory of my mother, Vivien Stchedroff, who persuaded me to turn my ideas into formal research.

I would like to especially thank my supervisors, Professor Chris Potts and Professor Russell Cheng, for their help in getting me to the conclusion of this work.

I would also like to thank Professor Jörg Fliege for his suggestions regarding optimization techniques which formed the basis for latter portion of this thesis.

Thanks are also due to the administrative staff of the School of Mathematics (most recently Garry Hancock) for helping me navigate some of the complexities of being a part-time student.

Academic Thesis: Declaration Of Authorship

I, Niels Stchedroff declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.

Title: A SIMULATION METHODOLOGY FOR CONTINUOUS SYSTEMS

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as:

   Stchedroff N. & Cheng, R.C.H. "Modelling A Continuous Process With Discrete Simulation Techniques And Its Application To LNG Supply Chains", Proceedings of The Winter Simulation Conference, 2003

Signed:

Date:

Chapter 1. Introduction

1.1 Background

Creating an efficient simulation model of a continuous problem is problematic, since digital computers are fundamentally discrete state devices. The Liquid National Gas (LNG) supply chain is a good example of such a problem – and of particular interest to the author.

The LNG supply chain is complex, technically challenging, and extremely costly to operate. Operations require a very high level of safety, and at the same time, a very high level of utilization of equipment. To ensure that planned operations are efficient and yet safe, high quality modelling is important – speed is essential to running sufficient tests to determine if the projected plan is stable in the face of uncertain events.

Models in the field to date have issues in terms of their performance. In the experience of the author, simulating a single year of operation might take seconds of computer time. While they are generally good software in the sense of the implementations, there seems to be an under use of the established concepts of simulation. Performance is generally poor as a result – time slicing systems such as Witness (Lanner) are popular. The slow models lead to analysis by intuition, backed by a limited number of experiments.

The concept behind the current work was to do was to review the existing concepts in simulation modelling, select an efficient methodology, implement it and then use it to demonstrate the possibilities for automated optimisation inherent in a fast system.

The intent is to create a modified approach to the problem of modelling the LNG supply chain.

1.2 Aims for this thesis

Given the above the following objectives were formulated for this work –

To select a suitable simulation methodology

As a first step analyse the system to be modelled against the existing simulation methodologies. Then use the results to select an approach and devise a pattern to apply it to the problem

To investigate the flexibility and performance of the modelling method

A key requirement is that the model has sufficient performance to support a very large number of runs in a useful period of time – a few hours. This enables the model to be used for experimentation and effective testing of the configuration for robustness.

Select experimentation methods

Find and implement suitable algorithms to enable effective experimentation of some aspects of the supply chain structure and operations.

Investigate the performance of the methods and the possibilities for automating design of the supply chain structure.

Investigate and select the specific parts of the supply chain structure to apply optimization techniques to. This includes which objects and their properties to uses as parameters and the way in which the supply chain structure are valued. The last part of this section of the work involves investigating the performance of the model and the results of the optimisation methods employed, given the choices made.

1.3 Original work

The core of this work is to design, implement and test an efficient environment for modelling continuous simulation problems. This technique will be validated and then used to investigate automated design. The optimization techniques to be used are not novel, but their application to this class of problem is.

1.4 Structure

In Chapter 2, the Liquid Natural Gas (LNG) supply chain is explained, along with the assumptions made in modelling it and the equations required. While detailed knowledge of the LNG supply chain is not vital in understanding the methodology that forms the backbone of this work, it provides a good illustration of the level of complexity involved in a typical real world problem. Chapter 3 deals with choosing the methodology – an examination of existing concepts and the selection of the chosen idea. In Chapter 4 the detailed design of the new methodology is examined. Chapter 5 presents the details of the implementation of the methodology. Chapter 6 deals with the design of experiments in automated design using the model. Chapters 7 & 8 sets out the experiments performed used the model and their results. Chapter 9 contains the conclusions and the suggestions for future work arising from them.

Chapter 2. The LNG Supply Chain

The liquefaction of natural gas makes it possible to ship it in liquid form to faraway markets, which are out of reach for conventional transport by pipeline due to the volume reduction achieved in the liquefaction process.

An LNG supply chain consists at the highest level of loading ports shipping LNG to one or more receiving ports. A typical supply chain is depicted in Figure 1.

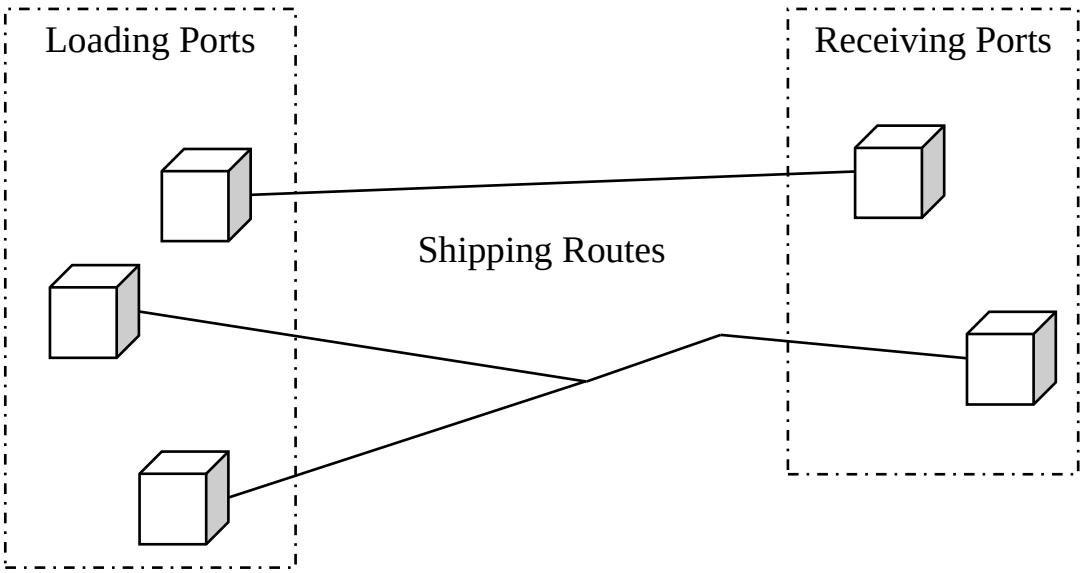

Figure 1. Overall Structure

A primary driver in this process is time – LNG decays over time as the lighter chemicals (technically referred to as *fractions*), which tend to have the highest energy coefficients, boil off. Equally, the equipment used is very expensive and minimizing the size of installations can save millions or even billions of dollars. The overall aim is to keep material flowing through the system, even if it means shutting down individual items – though this is to be avoided if possible.

The loading ports and receiving port structures are depicted in more detail in Figure 2.

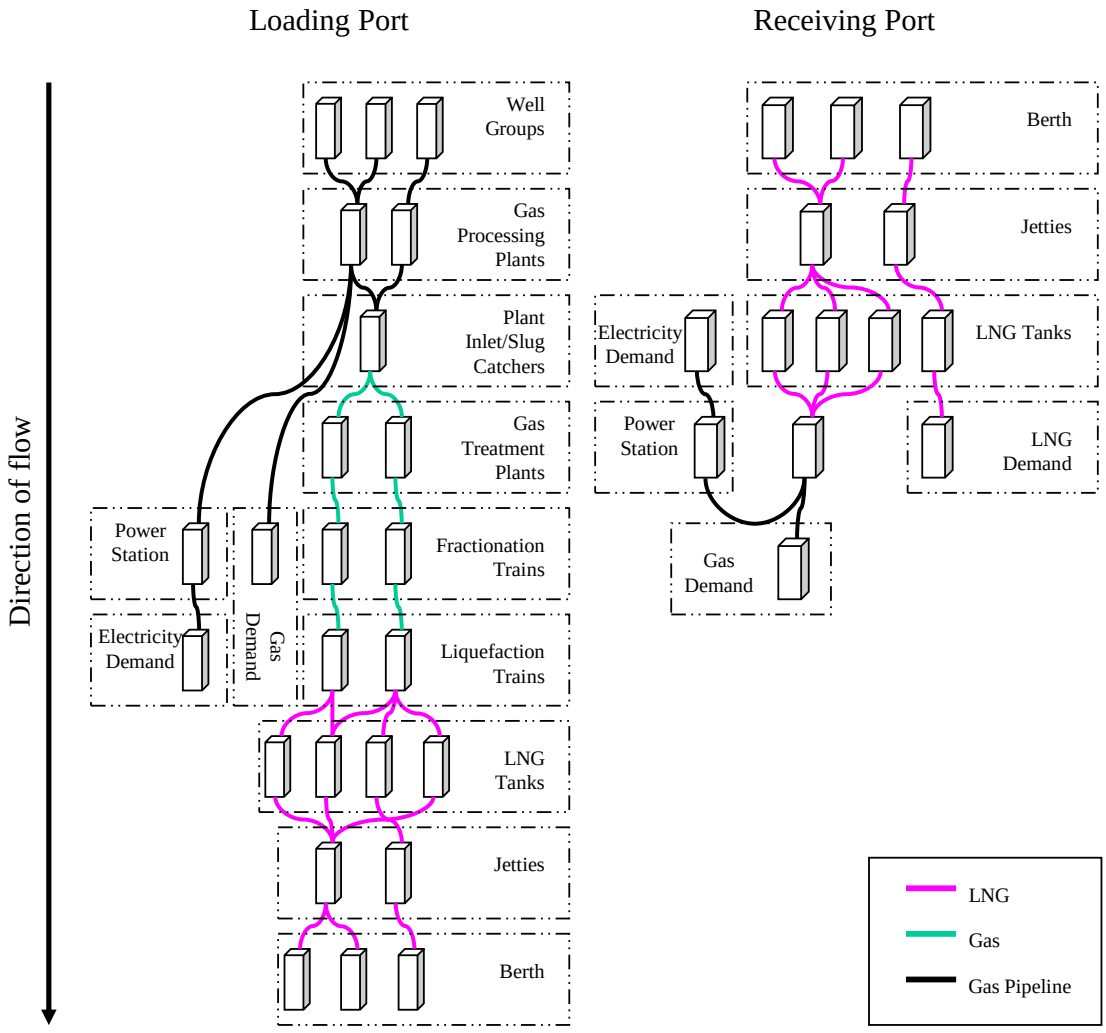

Figure 2. A simplified view of typical port structures

In the loading ports, gas is produces by wells, and processed before being sent to the main plant. The processing included removing the majority of water, present in the gas as vapour.

At the plant, the *slug catchers* manage the arrival of the gas from the pipelines – controlling flow, pressure and temperature. The gas is cleaned of impurities such as water, carbon dioxide and sulphur compounds. As processed gas enters the plant through the Gas Transmission System (GTS), slugs of liquid are trapped by the Slug Catcher whilst the processed gas continues to flow into the plant. The liquid recovered from the slug catcher is too heavy in composition to be acceptable for inclusion in the LNG production. Therefore this liquid or condensate is stored in a dedicated condensate tank for subsequent export.

The heavier hydrocarbon components such as Liquefied Petroleum Gas (LPG) and condensate are also removed to meet the very exact quality specifications imposed on LNG by its customers

The gas is then converted into LNG by the liquefaction trains – gigantic refrigeration facilities. There are two refrigeration cycles in use at an LNG Plant. The first cycle, the Pre Cooling Cycle uses pure propane as a refrigerant and chills the natural gas to minus 35 degrees centigrade. The second cycle, the Liquefaction Cycle, uses a mixture of components (Nitrogen, Methane, Ethane and Propane in varying proportions). This refrigerant called Mixed Refrigerant chills and liquefies the Natural Gas from minus 35 degrees centigrade to minus 161 degrees centigrade. Alternatively, the second cycle can be of the cascade type, where each of the components are used separately in a sequence (Finn, A.J, Johnson G.L, Tomlinson, T.R. 2000). The reason for cooling to liquid in stages is to separate the fractions being removed as they liquefy and to make the refrigeration process more efficient.

The liquefied gas is stored at -160ºC in above ground tanks. When ships are arrive, the LNG is pumped over long jetties to the berths where the ships are waiting – the ships are too big to tie up at a quay directly. Sometimes LNG is directly piped to customers at the production facility – but this is relatively rare.

The receiving ports are much simpler – the LNG is taken ashore and stored. From there it is sent to customers, either in liquid form or converted back into gas.

2.1 Object structure and States

In the supply chain structure all the key objects to be modelled possess a very similar basic structure. Figures 1& 2 show that within the ports are a number of substructures, such as tanks, pipelines and processing equipment. Each item can be considered to have the following basic properties illustrated in Figure 3 & Table 1. This is the level that the operations of the supply chain will be modelled at – flow in and out of objects, with their internal working represented only by a contents level.

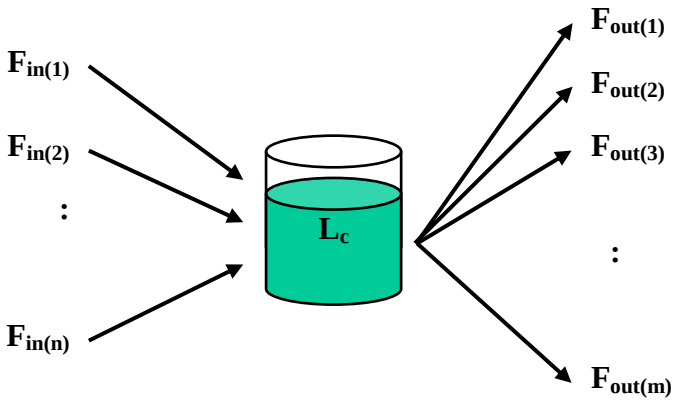

Figure 3. Basic Item

Table 1. Basic Item Properties

$F_{in(n)}$	Flow into the object from another object per unit of time
$F_{out(m)}$	Flow out of the object to another object per unit of time
L_c	Current level in the object.

Each object is assumed to have one input and one output point. However, multiple objects can be attached to each such point. Note that pipelines and connectors are also treated as similar objects under this definition. This approach has the advantage of greatly simplifying the executive – the structure to handle this aspect of object behaviour can be implemented once and inherited by each object requiring it.

An important point is that the rates of flow are considered to be constant, rather than ramping up and down over a period of time. While this might appear approximate, the rate of change in flow in real life is very rapid compared to the time scale at which the modelling takes place. If it takes a minute or so to go from full flow to shutdown and visa versa, this is close to instantaneous when considering a model on the hourly or daily scale.

Given the linear nature of operations, the level at any future point can be calculated as:

$$l_f = \left\{ \sum_{i=1}^{n} F_{in(i)} - \sum_{j=1}^{m} F_{out(j)} \right\} \times \left(t_f - t_c \right) + l_c$$

where t_f = the future time point, t_c = the current time point, l_c = the material level at the current time, l_f = the material level in the object at time t_f.

From the above, we can see that the following basic states that need to be considered.

2.2 Object Maximum

In this state, the object has been filled to its maximum internal capacity. As a result, the inputs must be scaled back so that:

$$\sum_{i=1}^{n} F_{in(i)} \leq \sum_{j=1}^{m} F_{out(j)}$$

Dividing the normal total output by the total input that the connecting objects can provide does this. This gives the ratio of input that is possible to the ratio of input that is being supplied. For example, if the total input can only be 60% of the possible amount, then each input is scaled back to 60% of the possible amount:

$$F_{new\_in(x)} = \left\{ \sum_{i=1}^{m} F_{out(i)} \Big/ \sum_{j=1}^{n} F_{in(j)} \right\} \times F_{in(x)}$$

2.3 Object Minimum

In this state the object has reached the minimum internal level. As a result, output must be scaled back to match input, as in the Object Maximum. This is done (as before) by calculating the ratio between the total possible and the total actual, and using it to scale the outputs accordingly:

$$F_{new\_out(x)} = \left\{ \sum_{i=1}^{n} F_{in(i)} \Big/ \sum_{j=1}^{m} F_{out(j)} \right\} \times F_{out(x)}$$

2.4 Object in restricted operation

In this state the object can only pass through a proportion of its capacity. Outputs are scaled back accordingly:

$$F_{new\_out(x)} = \left\{ \sum_{i=1}^{m} F_{out(i)} \right\} \times e$$

where $\boldsymbol{e}$ is the effect of the restriction.

2.5 Object in normal operation

The object has space from all the input from the connecting objects, and capacity to satisfy all the outputs.

2.6 Ships

Ships can be considered as above when they are in port, loading or unloading. In the former case, there are no outputs, in the latter, no inputs. Leaving and entering ports creates an event, which disconnects the ship, causing the jetties to recalculate their in/out flow rates.

Chapter 3. Choosing a Methodology

3.1 Preamble

This chapter examines the historic usage of simulation in capacity and operations planning in the LNG industry, examines the general structural issues involved with simulating continuous systems, discusses the various types of methodologies and finally, picks an approach.

3.2 Historic usage of simulation in the LNG field

Operations in the LNG industry are carefully planned in advance, typically by creating an ADP (Annual Delivery Program). This is a schedule for production and consumption of LNG, with detailed information on the timing and sizing of the ship cargos required.

Simulation models are used to construct the delivery program in two phases. In the first, an ADP is generated with all probabilistic events are temporarily removed from consideration and replaced with a fixed contingency (extra time) attached to each ship voyage. This is generally referred to as the **"Generate"** phase. In the second, the ADP in question is then tested for robustness by running the model with the random events, and detecting whether the model stayed within the ADP. This is generally known as the **"Test"** phase. The results from the second stage can be used to modify the contingencies, or the basic structure of the model and the process is repeated.

3.3 Structural Issues

The equations discussed in the previous section are extremely simple. The events that cause state changes occur at discrete points in time. This means that calculating the future states of the system does not require solution of differential or integral equations. In this case continuous methods are not required, leading towards selection of a method based on discrete modelling techniques. The differentiation between the discrete and continuous simulation is somewhat artificial, in any case. In particular, since digital computers cannot operate truly continuously, we can only produce the illusion of continuity, as in AweSim, Extend and Witness (Pidd 1998). For example, WITNESS can give the impression that it is operating in a truly continuous fashion. It is, in fact, calculating values on a next-event basis.

The overhead of providing threaded program execution for continuous modelling is quite high, particularly in the case of complex systems, where large amounts of data will be required to be moved in and out of memory.

In the case of FLEET, the ADGENT models for Shell and some systems created by Lanner in WITNESS, shipping systems are modelled in this discrete manner, using changes in state at specified times to represent the various operations.

Performance is an important concern. When validating a configuration or using experimentation techniques to create a configuration hundreds or thousands of runs may be required. Users of this kind of model have also expressed an interest in being able to manipulate the results generated by the system and using the model to validate the changes. To be usable this would require nearly instantaneous recalculation.

The complexity and the stochastic nature of system being modelled require that simulation should be used – mathematical programming approaches would not be suitable (Pidd 1998).

3.4 Approaches for Discrete Event Simulation

The transaction-flow worldview often provides the basis for discrete-event simulation. A system is visualised as unit of “traffic” that move from point to point, competing

for scarce resources. Discrete event simulation may be defined as one in which the state of the model changes only at discrete, but possibly random, set of simulated time points. Two or more "traffic units" often have to be dealt with at the same time. This is done serially at that time point (Scriber & Brunner, 2000).

There are four basic methods to be considered– Event, Process and Activity based approaches (Mitriani 1982 & Pidd 1998) and Three Phase Approach (Pidd 1998). It should be noted that the approaches are equivalent in the functional sense – a given model can be implemented in any one (Mitriani 1982 & Pidd 1998).

The following as some example of the use of Discrete Event Simulation in the literature –

- Beck & Nowak (Beck & Nowak 2000) used discrete event modelling combined with activity based costing to create costing models for manufacturing environments.

- Burgsteden, Joustra, Bouwman & Hullegie (Burgsteden, Joustra, Bouwman & Hullegie, 2000) used discrete event simulation to model road traffic at Schipol airport.

- Schunk & Plott (Schunk & Plott, 2000) used Micro Saint to create a discrete event model for the manufacturing process for vehicles.

- Andersson & Olsson (Andersson & Olsson, 1998) used Taylor II to construct a discrete event model of an assembly line, for capacity & operations planning purposes.

- Kiran & Cetinkaya & Og (Kiran & Cetinkaya & Og, 2000) used ProModel to construct a discrete event simulation model of a new international terminal for Istanbul Airport.

- Swedish (Swedish, 1998) used ProModel 4.0, a discrete system to model a barge transportation problem.

- Kyle & Ludka (Kyle & Ludka, 2000) used ProModel to create a discrete model for the furniture manufacturing industry.

- Trone, Guerin & Clay (Trone, Guerin & Clay, 2000) used Extend to create a discrete model for the transportation, processing and disposal of radioactive waste.

- Bruzzone, Giribone & Revetria (Bruzzone, Giribone & Revetria, 1999) used C++ to build an object oriented, discrete simulation model for shipping container terminals.

- Daum & Sargent (Daum & Sargent, 1999) examined discrete event simulation paradigms in their investigation of scaling and reuse in object oriented systems.

- Joines & Roberts (Joines & Roberts 1999) based their examination of object oriented techniques around event and process based systems, and used C++ to illustrate methods.

3.4.1 Event based

This was once a very common approach, primarily due to its use by SIMSCRIPT (Russell 1987). Since latter versions of SIMSCRIPT emphasised the Process based approach (see below), this method has fallen out of favour.

The event based model works by constructing event routines based on the activity-cycle diagrams for the various parts of the system being modelled. The executive creates a list of the events in these routines that are due to occur and executes them. When the next event in the list is reached, it is executed, the system clock is advanced and a check is made to see if new future events need to be added to the list (Law & Kelton 2000).

The advantage of this approach compared to others is speed (Pidd 1998) – however, the weakness is in considering interaction between parts of the model. The advantage of this approach is in not checking all the conditional events every time an event executes. Instead all the results of an event occurring are dealt with in the event routine. This means that all such consequences must be foreseen and built into the model.

3.4.2 Process based

In this approach, for each class of entity, the life-cycle (process) of the entity is considered (Law & Kelton 2000). For example, an LNG tanker sails between ports, docks, loads, unloads etc. Each of these operations of states forms a part of life-cycle. The job of the executive, in this case, is to move each entity forward through its process, if possible. Underlying the system are lists for future events and for entities that have been suspended due to unconditional delays (conditional only on time), and conditional (waiting for other specific conditions to occur) (Schriber & Brunner, 2000).

Process based simulation is used by SIMCRIPT II.5 (Russell, 1987) and is quite common (Pidd 1998). The following are some example of the use of Process based systems in the literature –

- Takakuwa (Takakuwa 1998) used ARENA/SIMAN, a discrete process based system to examine methods for transportation/inventory systems.

- Takakuwa, Takizawa, Ito & Hiraoka (Takakuwa, Takizawa, Ito & Hiraoka, 2000) used ARENA/SIMAN, a discrete process based system to model the operation of warehouses.

- Golkar, Shekar & Buddhavarapu (Golkar, Shekar &Buddhavarapu, 1998) used C++ and SIMAN to construct a discrete, process based system.

- Kilgore & Burke (Kilgore & Burke 2000) used Java to create a process based, object oriented modelling system.

Huang & Iyer (Huang & Iyer 1998) proposed that process oriented discrete models should be converted to event based, in order to improve performance. The analysis of their results showed that the equivalent event based model was not only considerably quicker running - more importantly, for the process based approach the time taken increased non-linearly with complexity, while the event based method did not.

Perumalla & Fujimoto (Perumalla & Fujimoto 1998) looked at increasing performance in process-oriented views. They further argued in favour of process-

based models, but suggest that the features of the process-oriented model should be limited to the point where the model is in fact nearly an event based one. This is done to improve performance.

Pidd (Pidd 1998) pointed out that the process-based approach is broadly equivalent to the Three Phase Approach – the conditional and unconditional events have their equivalent in the Bs and Cs, and the executive will uses a phased approach. However, process based approaches are vulnerable to deadlock. Schriber & Brunner (Schriber & Brunner 2000) comment on the key nature of the resources management system in discrete simulation.

Pidd (Pidd 1998) takes the view that for complex systems, the Three-Phase approach is preferable, largely for this reason.

3.4.3 Activity based

In this approach, the focus is on the activities that are performed in the system, each activity has a test head that determines whether the activity can execute. The executive scans the activities to find the simulation time at which the next activity (or activities) can start, moves the clock to that time, enables the activities and then moves to the next suitable time point.

The activity-based approach is not used very much. A major drawback is that simulation programs written using this technique are slower (Pidd 1998). The cause is that there is no differentiation between the types of activities – all must be scanned at each step. In the Three Phase Approach, which has largely replaced this methodology, the activities are separated into Bs (fixed time events) and Cs (events conditional on resources etc.). Only the Cs (which are generally fewer) are scanned (Law & Kelton 2000).

When Perumalla & Fujimoto (Perumalla & Fujimoto 1998) reviewed the main types of discrete simulation – event, activity and processed oriented, they almost ignored activity-based methods.

Activity based models are still used, however – Shi (Shi 2000) discuses an activity based, object oriented modelling approach to problems in the construction industry.

3.4.4 Three Phase Approach

The core of the three-phase approach is dividing the way an activity starts into two categories – Conditional and Bound. Bound events occur at particular, pre-computed times. Conditional events are affected by other factors, such as the availability of resources.

The executive operates in three phases – the first (A phase) finds simulation time point when the next event will occur. The second (B phase) executes all the Bound tasks that due to occur. The third phase (C Phase) tries all the outstanding Conditional events, to see if the required conditions have been met (Law & Kelton 2000; Pidd 1998).

There are several advantages to this approach -

- The dead locking problems of the process-based approach are avoided.
- The inefficiency of the activity based approach trying the test head for every activity is avoided.
- The complexity of modelling interaction in the event based method (where each event routine must contain the actions required to deal with interaction), is dealt with in the handling of the conditional events.

Examples of this approach being used include:

- Ioannou (Ioannou 1999) using Three Phase Approach simulation to model construction work.
- Marzouk & Mosehli (Marzouk & Moselhi 2000) used C++ to construct a Three-Phase system to model earth moving in construction.
- Pidd & Castro (Pidd & Castro , 1998) used C++ and Three Phase Approach modelling techniques in their examination of problems relating to large-scale modelling.

3.4.5 Performance

Several simulation systems have been created in the LNG supply chain modelling area – most notably, FLEET for Lloyds of London and ADGENT for Shell. In the former case, the performance of the model is such that that for reasonably complex models, runtime is in the order of hours. ADGENT is somewhat better. While in both cases the performance is adequate, it is my belief that neither has optimised the fundamental method of modelling, relying instead on the power of the hardware they are run on. Lanner, who are best known for their work on the Witness modelling environment, appear to have used time-slicing (the system steps through simulation time in regular, fixed increments). This results in extremely slow operation for any reasonable level of complexity, particularly when modelling systems that are continuous in nature.

A major aim of this work is to construct a computationally efficient modelling system. As we have seen, both Huang & Iyer (Huang & Iyer, 1998) and Perumalla & Fujimoto (Perumalla & Fujimoto, 1998) commented extensively on the efficiency of the process based approach – in the former, arguing that processes based systems be translated into event based ones, and in the second, arguing that by limiting the process based approach (to the point of becoming an event based model) higher performance can be achieved. The activity-based method is slower inherently – since it indiscriminately scans for the start of the next activity, rather than jumping to the time of the next event in a list (Pidd, 1998; Law & Kelton 2000).

The above tends to argue for the adoption of either the event based or three-phase methods, which are more efficient (Huang & Iyer, 1998; Pidd, 1998).

3.4.6 Conclusions on the selection of methodology

To create a reasonably efficient system, two principles need to apply

- The level of detail must be variable – if the user wants to build an elaborate system of breakdowns, representing the complex systems aboard each tanker, he/she is free to do so. By the same token, if the user wants to build a simpler model, the computational load should reflect that

- The method selected should work on the principle of only doing something if there is an event – periodic recalculations (such as in time-slicing) would lead to an excessively slow system.

As previously discussed, the boundary between discrete and continuous modelling is to a certain extent, an illusion. Modelling of continuous systems on a digital computer is accomplished by computing the values of simulation-time dependent equations at a given point. In this case, the nature of the problem indicates a discrete approach.

Of the discrete system the most flexible and efficient for a reasonable order of complexity is the three-phase approach. The time handling is flexible, utilising a Future Event List (FEL), and it avoids the complexity of entity interaction associated with the event scanning approaches (Banks, Carson, Nelson & Nicol, 2000). As previously mentioned several authorities have given the opinion that process based systems are less computationally efficient than event based.

The recommendation therefore is a discrete simulation system, based on the three-phase approach.

The key to the three-phase approach is to divide the way an activity starts into two event types – Conditional and Bound. Bound events occur at particular, pre-computed times. Conditional events are affected by other factors, such as the availability of resources. The performance of a simulation model depends very much on an effective choice of conditional events, and we discuss this in the next section.

Chapter 4. Modifications to methodology

The methodology selected, Three-Phase requires some modification to make it suitable for use with respect to this problem. This section describes these modifications and the reasons that they were made.

4.1 Basic Concepts

The executive operates in the traditional Three Phase Approach manner – the first (A phase) finds simulation time point when the next event will occur. The second (B phase) executes all the Bound tasks that due. The third phase (C Phase) tries all the outstanding Conditional events, to see if the required conditions have been met (Law & Kelton, 2000; Pidd, 1998).

The flow through an object is subject to a single limit, for both inputs and outputs. This consists of a maximum and minimum flow rate. If the minimum flow rate is not met the object must be completely shut down.

Breakdowns, maintenance etc. are modelled by reducing the maximum flow value – to zero, if required.

4.2 B Events

B events in the classic Three-Phase model are those that have their start or finish time determined in advance (Pidd 1998). This can apply not only to inherently deterministic events such as darkness and tides but also probabilistic events such as

weather and breakdowns. Pre-computing the time of the next occurrence, using a given distribution, can accomplish this.

In this work the following B events have been implemented:

Table 2. B Event Types

$B_{Breakdown}$	A breakdown has occurred on the object affecting flow through it. Used in the Test mode of operation of the model only.
B_{Demand}	Seasonal demand variation – demand for LNG varies between Summer and Winter (for example). Used in both the Generate and Test modes for the system

At the start of a run, the B events will be computed and a list of them constructed, ordered on start time. In this model B events are actually pairs - a start and an end event. Thus in our case a B event is in effect two events. This innovation was driven in part by performance – at a given point in time, several B events may be acting on a single object.

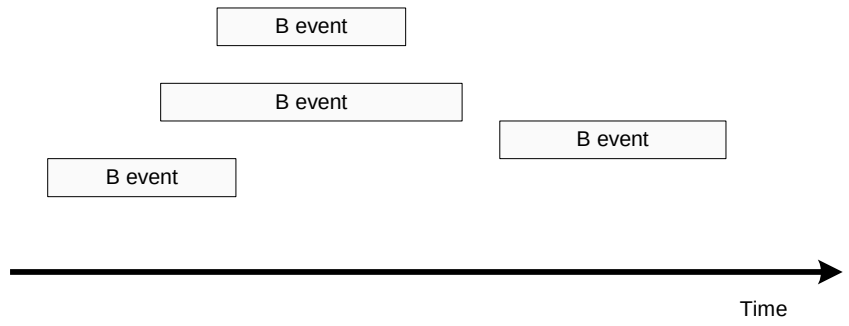

Figure 4. Original B events affecting the object

Each object has one or more performance parameters that can be affected by events. In this thesis we will be considering events effecting flow of material. This effect will continue for an amount of time that is generally probabilistic and hence can be calculated before the start of the run. From this it can be seen that combining the overlapping B events is possible, and will simplify event handling. Figure 4 shows an

example of the original events and Figure 5 shows the corresponding runtime versions.

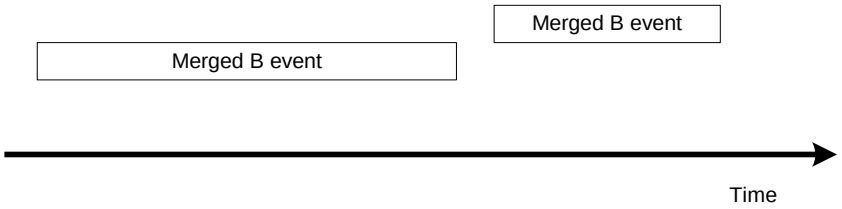

Figure 5. Runtime B events for object

Simultaneous events are additive – if two events are occurring at the same time on the same object, then the total effect is the sum of the individual effects (Figure 6).

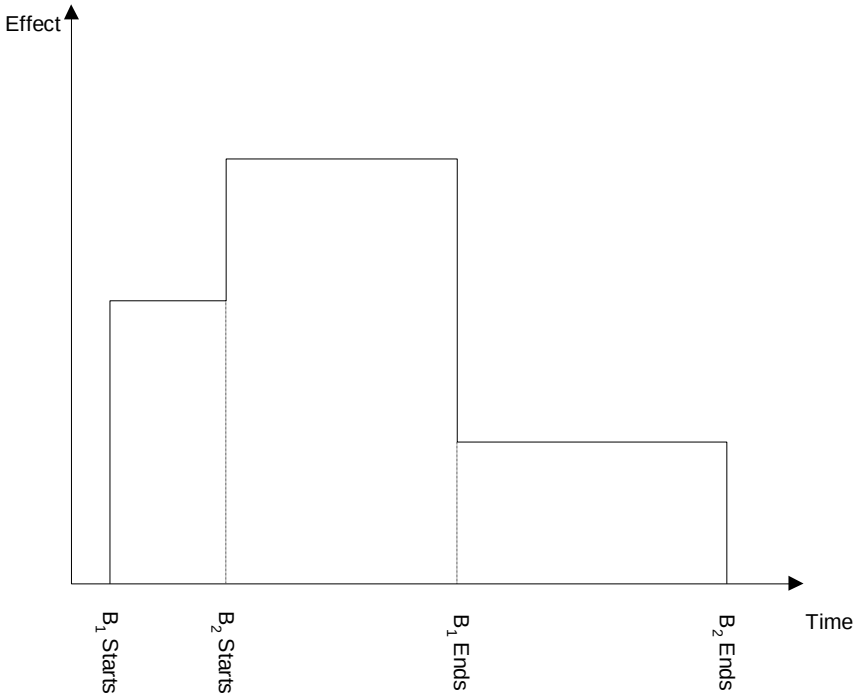

Figure 6. B Events for a single object

A major concern with this modelling method is speed. For this reason, the obvious method of scanning through the events, determining the next one to occur was rejected.

The way that the B events are handled is determined by the way in which the model is intended to be used. There are two modes to be considered:

Table 3. Modes of operation for the model

Generate	A plan of operations is constructed. Probabilistic events are represented by adding a fixed amount of time to each ship voyage. Only ***demand change B events*** are used.
Test	The probabilistic events are turned on, and the model is run a number of times, to discover whether the plan is feasible, and if not, where it may fail. So far, ***the breakdown events*** have been modelled.

The B events themselves are not typically a subject of experimentation when designing LNG systems – they are a pre-condition of the equipment used, the tides and the weather.

4.3 C Events

The C events, in the Three Phase Approach, are those are conditional in nature – that is they occur as a result of internal operation of the model. For the LNG supply chain model they are the key to the construction of an efficient model. The list of C events is given in Table 4.

Table 4. C Event Types

C_{Full}	Object reaches maximum internal level
C_{Empty}	Object reaches minimum internal level
C_{Normal}	Object returns to a situation where it is neither at the maximum or minimum internal level
$C_{Filling\_To\_Normal}$	Object has completed refilling back to the normal minimum level
$C_{Emptying\_To\_Normal}$	Object has completed emptying back to the normal maximum level after having exceeded it.
$C_{End\_Laden\_Voyage}$	Specific to ships – a laden voyage (to a receiving port) has been completed.
$C_{End\_Empty\_Voyage}$	Specific to ships – an empty ship arrives at the loading port.
C_{Attach}	Specific to ships – a ship has actually attached to the flow system at a port.

In the classic Three Phase Approach C events generally have a matching B event. Due to the nature of this model, it is however, possible to compute when these events will occur (and when the event effect will end) at a given time point unless another event occurs before this.

If such an event occurs the matching B event would have to be recomputed dynamically during the course of the run – which would make them difficult to handle as standard B events. For this reason they are handled as another C event.

It is possible to consider the C events in this way because the operation of the model is deterministic – unless an event occurs, the future state of the model can be calculated by applying the time elapsed to the equations governing flow. The time to the next C event is also recalculated if the object has been affected by change to a

neighbour, or a probabilistic event (a B event), such as weather or a breakdown has affected the operation of the object.

4.4 How the effects of an event propagate

Let us consider the case of a C_{Full} event affecting a static (non-ship) object connected to a supply chain, such as in Figure 7.

The model reduces inflow into the object affected. This in turn affects the objects that feed the object in question. In particular it will affect the rate at which these objects are filling or emptying themselves. In other words, the time to the next C_{Full}, C_{Empty}, $C_{Filling\_To\_Normal}$ or $C_{Emptying\_To\_Normal}$ is modified.

This does not directly affect the other items in the supply chain yet. Since we are not trying to predict the effect of one C event on the timing of another, we can consider them in isolation – what we are interested in is the time of the next event that will occur.

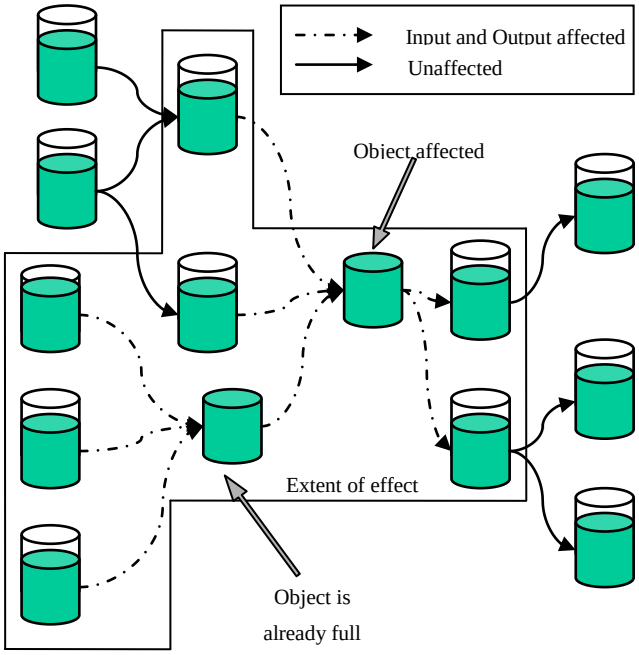

Figure 7. Objects Affected by the Event

This also holds good for all other C events – in fact for both B and C events. Consider that in each case the events modify the operating capability of an object. When estimating the time of the next C event for each object, only the current input and

output values need to be considered. This is because the estimates for the times of the C events are updated when they are affected by a change in object performance.

Note the case of a neighbouring object inputting into the object affected by the C event that is full itself. Here, the input rate into the neighbouring object is affected – in turn affecting its inputs. This chain of effects will only continue so far as the effects pass through the objects, effecting further objects.

This is fundamental to the performance of this approach – we only need consider the effect of an event on the neighbours of the object affected, as well as the object itself. This means that a blanket recalculation of C events is not necessary every time an event (both B and C type) executes.

After an event (of either type) occurs the C events for that object and the objects immediately affected by it are recalculated.

4.5 The Executive

The approach is the standard Three-Phase model, with some modifications to the events types and their handling. This section discusses the modifications.

4.5.1 Pre Processing

The ordered list of B events is created, and all the objects are scanned to find the first C event(s) – there may be more than one C event occurring at this time point, of course.

4.5.2 B Phase

Currently only one type of B event is implemented – B_{Flow} which affects flow through the object (see figure below). The change is executed, and then the object status is checked. If there is too much input, reduce it. If there is too much output, reduce it. If the event has less effect than the previous B_{Flow} on the object, try to increase the flow into the object. This comes into play when a B event ends – when the end event is reached, the model attempts to restore flow.

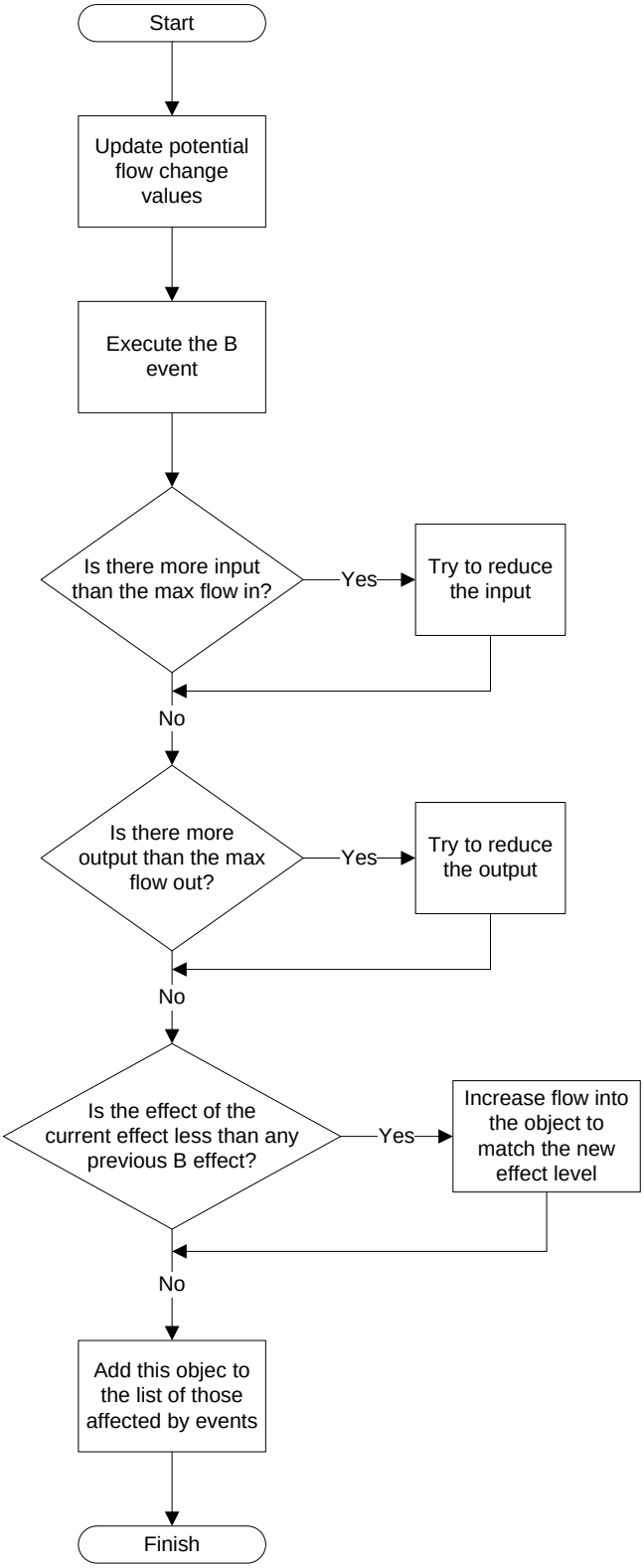

Figure 8. Handling B_{Flow} Event

Table 5. B event code

```
/**
 * Deal with a B event
 *
 * @param clock
 *            The time
 */
public void bFlow(final double clock)
{
	updatePotentialChanges();
	final double effect = getLargestEffect();
	final double oldEffect = getBEventModification();
	setBEventModification(effect);
	final double current_inputs = getCurrentFlowFromInputObjects();
	final double current_outputs = getCurrentFlowIntoOutputObjects();
	double required_change_in = 0.0;
	double required_change_out = 0.0;

	// If we have too much flow in, reduce if possible
	if (current_inputs > getMaxFlowIn())
	{
		required_change_in = current_inputs - getMaxFlowIn();

		decreaseFlowIn(required_change_in);
	}

	// If we have too much flow out, reduce if possible
	if (current_outputs > getMaxFlowOut())
	{
		required_change_out = current_outputs - getMaxFlowOut();

		decreaseFlowOut(required_change_out * getConversionFactorUp());
	}

	// Is the effect of the new B event less (i.e. closer to 1.0)
	if (effect - oldEffect > 0)
	{
		double change_in = getPotentialIncreaseInFlowInFromObjects();
		final double max_change_in = getMaxFlowIn() - current_inputs;
		if (change_in > max_change_in)
		{
			change_in = max_change_in;
		}

		increaseFlowIn(change_in);

		if(hasMaxContents())
		{
			double change_out = getPotentialIncreaseInFlowOutToObjects();
			final double max_change_out = getMaxFlowOut() - current_outputs;
			if (change_out > max_change_out)
			{
				change_out = max_change_out;
			}

			increaseFlowOut(change_out);
		}
	}

	updatePotentialChanges();
	getAffectedObjectsInSim().add(this);
}
```

4.5.3 C Phase

The C events generally relate to flow – the apparent exception is the C_{Attach} event when a ship arrives at a port. In fact this is handled as a flow change event as well – the effects are all to do with flow into the neighbouring objects.

4.5.3.1 **C Phase Object States**

An object can have following states relating to the C events:

Table 6. C Event Related Object States

C_{Full}	The object has been filled to its maximum capacity. This means that output from the object must be greater or equal input.
C_{Empty}	The object has reached its minimum internal level. This means that input must now equal of exceed output.
C_{Normal}	The object has an internal level that is between the maximum and minimum levels for *normal operations* (see below)
$C_{Emptying\_To\_Normal}$	The object hasn't yet emptied to the maximum level for normal operation (N_{max})
$C_{Filling\_To\_Normal}$	The object has an internal level that has risen above, but has not reached a given level for the minimum level for normal operation yet (N_{min})
C_{Attach}	A new object is attached to the model

Reaching each object state is a C event (Pidd 1998); when they occur, a recalculation is undertaken for that part of the model. After each change in state, the time to the next state change (C event) is calculated. The following figure shows the state transition diagram for the state changes.

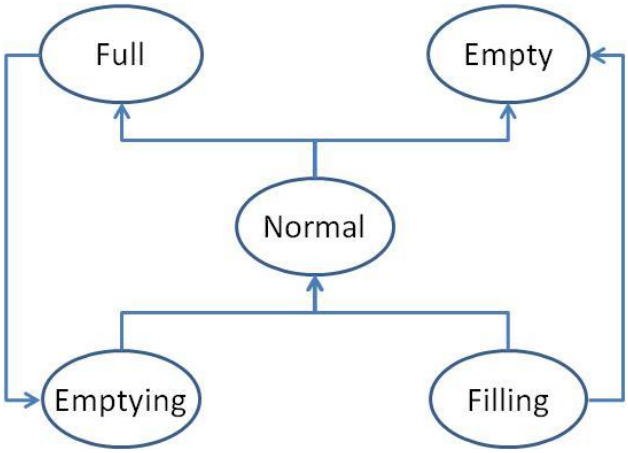

Figure 9. Object State Transition Diagram

This is a valid approach because the operation of the model is deterministic. The time to the next C event is also recalculated if the object has been affected by change to a neighbour, or a probabilistic event (B event), such as weather or a breakdown has affected the operation of the object.

The reason for the emptying and filling states is to prevent the system oscillating between Full /Empty and the Normal state, while never giving the structures time to return to relatively normal levels.

The object states relate to the level in the object. These are important because reaching specified levels triggers some state changes. These are given in Figure 10.

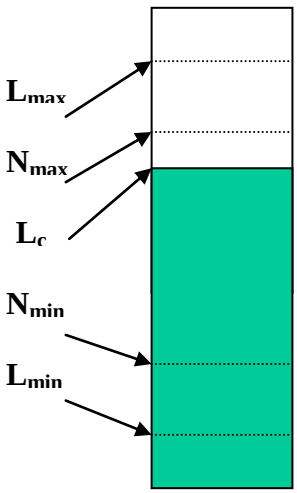

Figure 10. Object Levels

The following table shows how these object levels relate to the states outlined above.

Table 7. Object Level & Corresponding States

Level	**Description**	**New State**
L_{Max}	The maximum level for the object. If it is exceeded, the object is shut down	C_{Full}
N_{Max}	The level at which normal operations resume when the level drops to this point, after the object has reached L_{Max}	C_{Normal}
L_c	Current Level	N/A
N_{Min}	The level at which normal operations resume, after the level reaches this point after the object level has hit L_{Min}	C_{Normal}
L_{min}	The minimum operating level for the object – when it is reached the object is shut down.	C_{Empty}

4.5.3.2 **Handling States when they are reached**

4.5.3.2.1 C_{Full}

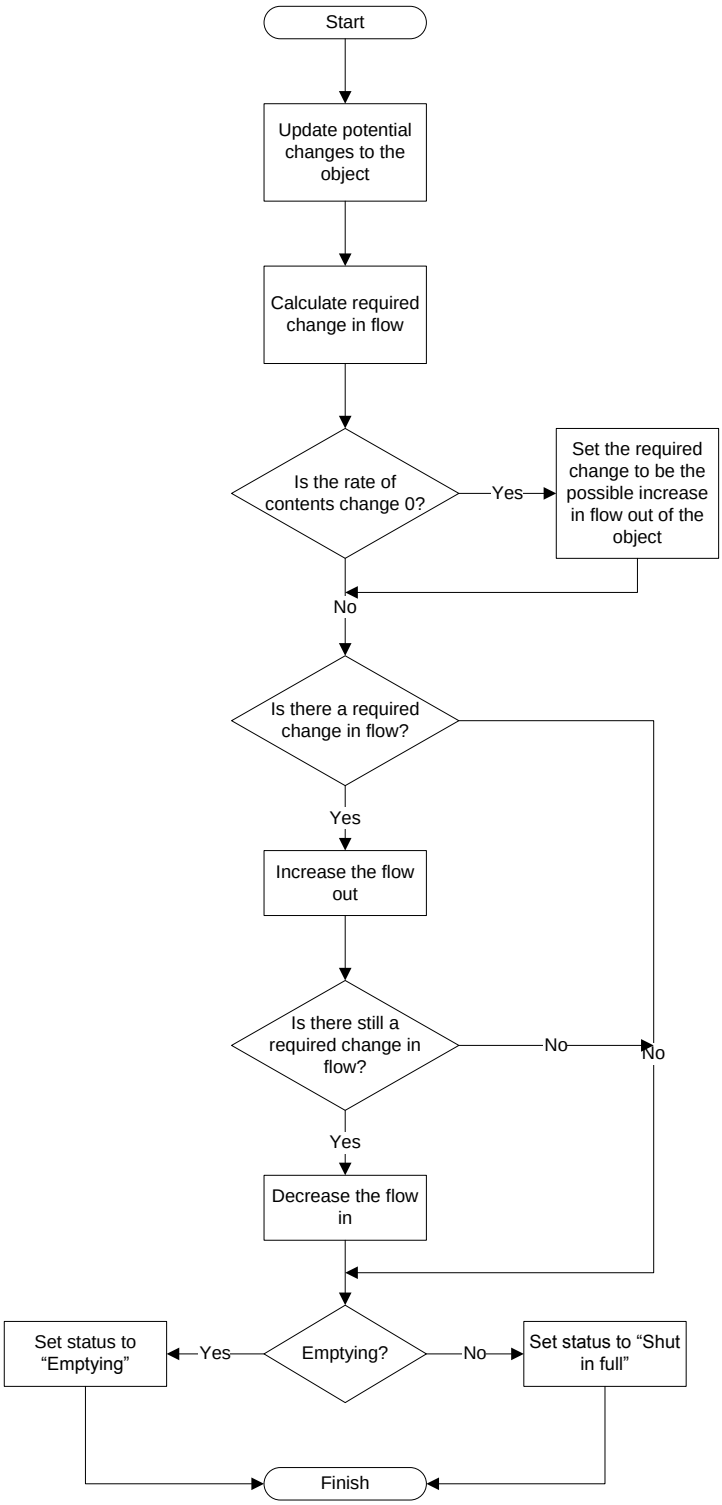

Figure 11. Handling C_{Full}

Initially, the objects potential changes are updated. Next the required change in flow is calculated – this is calculated as the amount required to stop the object contents increase plus an extra factor (25% as a default), so that the object will start to empty.

If the required change in flow comes out as zero – the object is full but stable, then the required factor is set as the possible decrease in flow out.

The change in flow required is made by –

1. First the flow out is increased, if possible,
2. If more is required, the flow in is decreased – again, if possible.
3. If the object is now emptying the status is updated. Otherwise the item is shut down.

Table 8. Code for handling C_{Full}

```
/**
 * @throws GenesisRunException
 * @throws AbortException
 */
public void cFull() throws GenesisRunException, AbortException
{
        // Set object status
        setStatus(FlowStatus.SHUT_IN_FULL);
        updatePotentialChanges();
        final double rateOfChange = getRateOfContentsChange();
        double required_change = rateOfChange;

        final double rs = getResettingFactor();
        final double max = getMaxFlow();

        final int fc = (int) (rs * max);
        required_change += fc;

        // The object is full but stable - try to increase flow out
        if (rateOfChange == 0.0)
        {
                required_change = getPotentialIncreaseInFlowOutToObjects();
        }

        cFull(required_change);
}

/**
 * @param requiredChange
 * @throws GenesisRunException
 * @throws AbortException
 */
public void cFull(double requiredChange) throws GenesisRunException, AbortException
{
        // If the object is actually filling...
        if (requiredChange > 0)
        {
                requiredChange = increaseFlowOut(requiredChange);

                requiredChange = getRateOfContentsChange();

                // if the required change was not entirely done by
```

```
                        // increasing the output, try decreasing input.
                        if (requiredChange > 0)
                        {
                                requiredChange = decreaseFlowIn(requiredChange);
                        }
                }

                // Have we succeeded?
                if (isEmptying())
                {
                        setStatus(FlowStatus.EMPTYING);
                }

                if (isStable())
                {
                        setStatus(FlowStatus.NORMAL);
                }
        }
```

4.5.3.2.2 C_{Empty}

First the objects potential changes are updated. Next the required change in flow is calculated – this is calculated as the amount required to stop the object contents decrease plus an extra factor (25% as a default), so that the object will start filling. If the required change in flow comes out as zero – the object is full but stable, then the required factor is set as the possible decrease in flow out.

First the flow in is increased, if possible, then the flow out is decreased if possible. If the object is now filling the status is updated. Otherwise the item is shut down.

Table 9. Code for handling C_{Empty}

```
/**
 * @throws GenesisRunException
 * @throws AbortException
 */
public void cEmpty() throws GenesisRunException, AbortException
{
        // Set object status
        setStatus(FlowStatus.SHUT_IN_EMPTY);
        updatePotentialChanges();
        final double rateOfChange = getRateOfContentsChange();
        final double rs = getResettingFactor();
        final double max = getMaxFlow();

        double required_change = -rateOfChange;
        final int fc = (int) (rs * max);
        required_change += fc;

        if (rateOfChange == 0.0)
        {
                required_change = getPotentialIncreaseInFlowInFromObjects();
        }

        cEmpty(required_change);
}
```

```
        /**
         * @param requiredChange
         * @throws GenesisRunException
         * @throws AbortException
         */
        public void cEmpty(double requiredChange) throws GenesisRunException, AbortException
        {
                // If the object is actually emptying...
                if (requiredChange > 0)
                {
                        requiredChange = increaseFlowIn(requiredChange);

                        requiredChange = -getRateOfContentsChange();

                        // if the required change was not entirely done by
                        // increasing the input, try decreasing output.
                        if (requiredChange > 0)
                        {
                                requiredChange = decreaseFlowOut(requiredChange);
                        }
                }

                // Have we succeeded?
                if (isFilling())
                {
                        setStatus(FlowStatus.FILLING);
                }

                if (isStable())
                {
                        setStatus(FlowStatus.NORMAL);
                }
        }
```

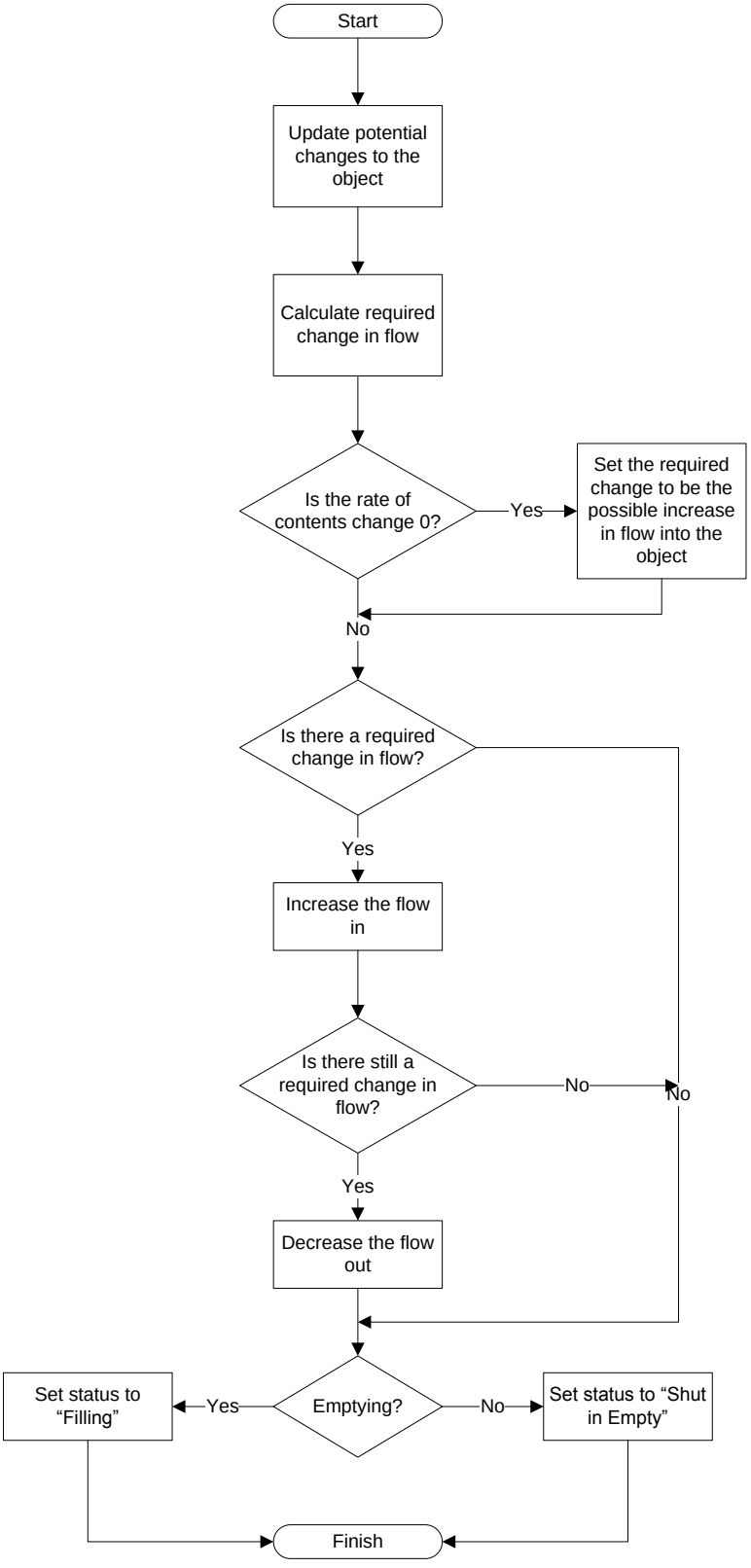

Figure 12. Handle C_{Empty}

4.5.3.2.3 $C_{EmptiedToNormal}$

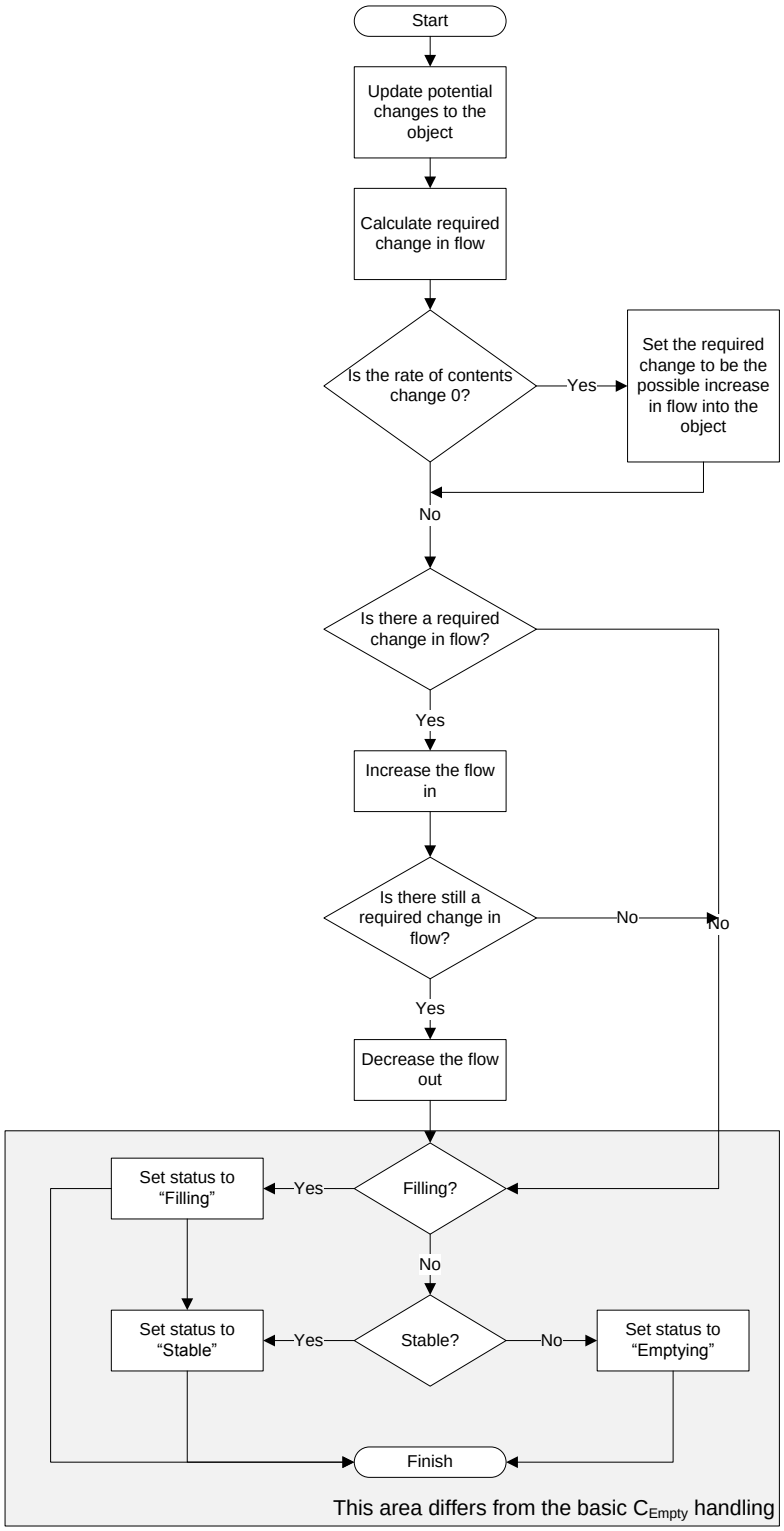

Figure 13. Handle $C_{EmptiedToNormal}$

This event is handled in a very similar manner to C_{Empty} – the difference is at the end the object may be emptying, stable or filling.

Table 10. Code for handling $C_{EmptiedToNormal}$

```
/**
 * Emptied until the object is at the normal level
 */
public void cEmptiedToNormal()
{
        setStatus(FlowStatus.NORMAL);
        updatePotentialChanges();
        double required_change = -getRateOfContentsChange();

        if (required_change == 0.0)
        {
                required_change = getPotentialIncreaseInFlowInFromObjects();
        }

        // If the object is actually emptying...
        if (required_change > 0)
        {
                required_change = increaseFlowIn(required_change);

                // if the required change was not entirely done by
                // increasing the input, try decreasing output.
                if (required_change > 0)
                {
                        required_change = decreaseFlowOut(required_change);
                }
        }

        // Have we succeeded?
        if (isFilling())
        {
                setStatus(FlowStatus.FILLING);
        }

        if (isStable())
        {
                setStatus(FlowStatus.NORMAL);
        }
}
```

4.5.3.2.4 $C_{Refilled}$

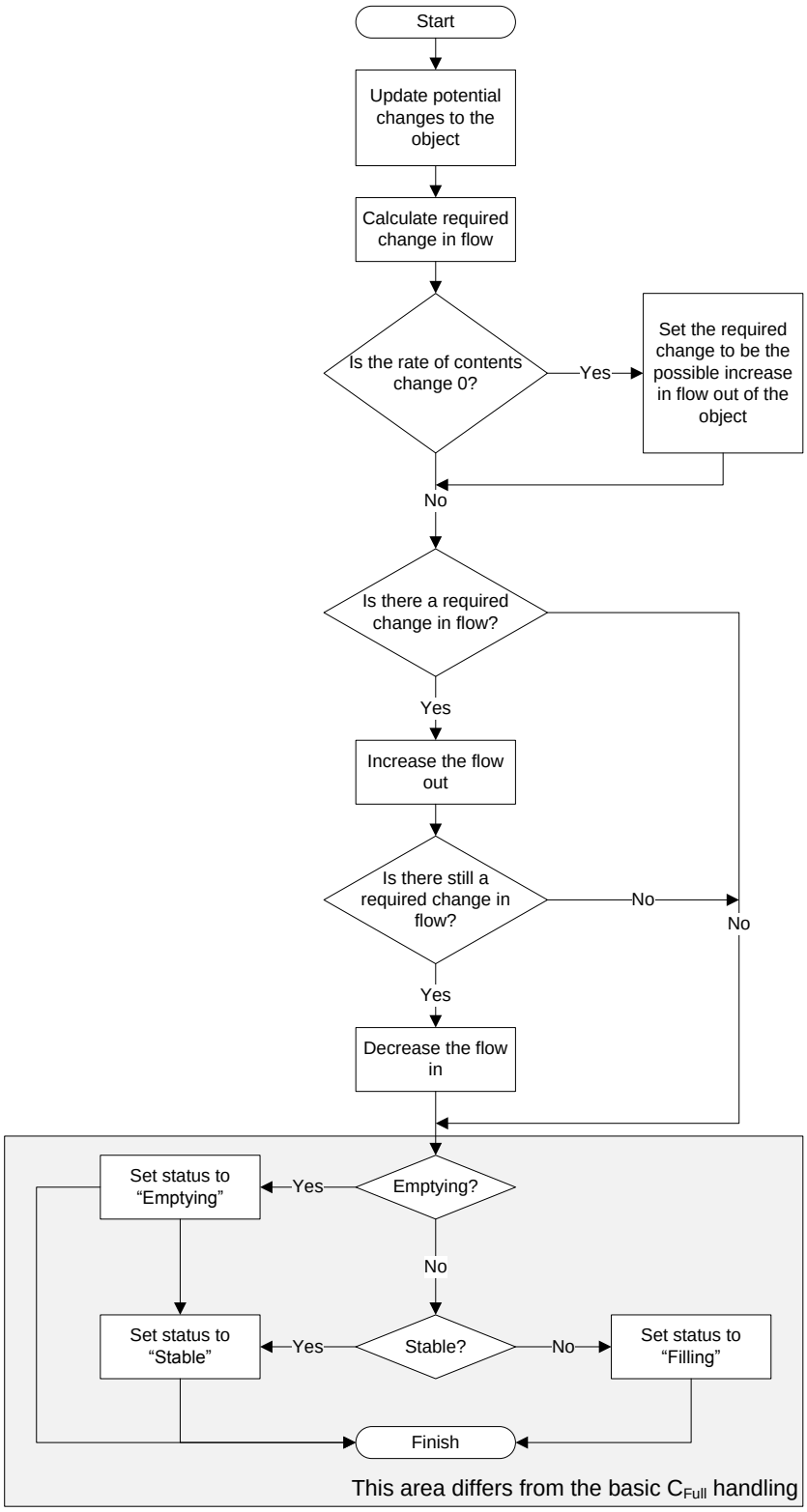

Figure 14. Handling $C_{Refilled}$

This event is handled in a very similar manner to C_{Full} – The difference is that the final state may be emptying, stable or filling.

Table 11. Code for handling $C_{Refilled}$

```
/**
 * Refilled to the normal level
 */
public void cRefilled()
{
        setStatus(FlowStatus.NORMAL);
        updatePotentialChanges();
        double required_change = getRateOfContentsChange();

        // The object is full but stable - try to increase it
        if (required_change == 0.0)
        {
                required_change = getPotentialIncreaseInFlowOutToObjects();
        }

        // If the object is actually filling...
        if (required_change > 0)
        {
                required_change = increaseFlowOut(required_change);

                // if the required change was not entirely done by
                // increasing the output, try decreasing input.
                if (required_change > 0)
                {
                        required_change = decreaseFlowIn(required_change);
                }
        }

        // Have we succeeded?
        if (isEmptying())
        {
                setStatus(FlowStatus.EMPTYING);
        }

        if (isStable())
        {
                setStatus(FlowStatus.NORMAL);
        }
}
```

4.5.3.2.5 C_{Attach}

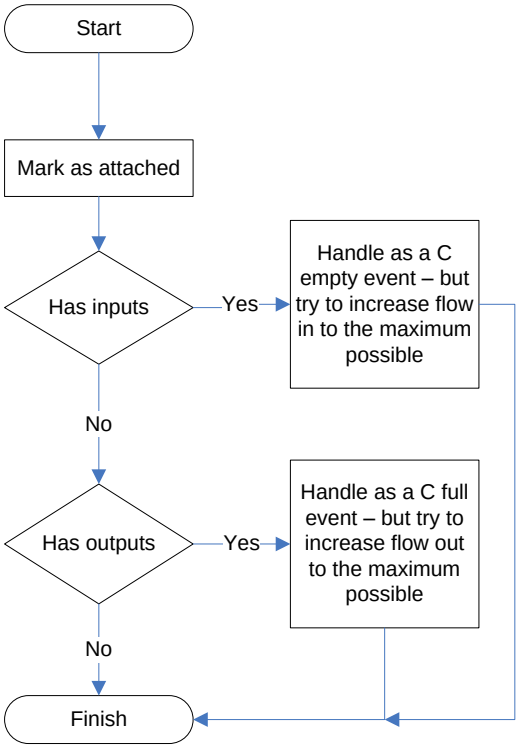

Figure 15. Handling C_{Attach}

The object is connected to the model. If it has inputs, it is handled as an empty object – which will cause flow into it. If it has outputs it will be handled as a full object – which will trigger increased flow out. In both cases the objective is to increase the flow to the maximum possible, given the maximum flow rates of the objects in question.

Table 12. Code for handling C_{Attach}

```
/**
 * Handle the object being attached to the rest of the model while it is running. For example, a
 * ship docking
 *
 * @throws GenesisRunException
 * @throws AbortException
 */
public void cAttach() throws GenesisRunException, AbortException
{
        setJustAttached(false);

        // If there are now input objects...
        if (hasInputs())
        {
                // Treat the object as if it is empty - increase the flow in, if
                // possible
                cEmptyMaxResponse();
                return;
        }

        if (hasOutputs())
        {
                // Treat the object as full - increase output
                cFullMaxResponse();
                return;
        }
}

/**
 * Handle the C Full event by increasing the flow in by the maximum amount possible
 *
 * @throws GenesisRunException
 * @throws AbortException
 */
public void cEmptyMaxResponse() throws GenesisRunException, AbortException
{
        cEmpty(getMaxFlowIn());
}

/**
 * Handle the C Full event by increasing the flow in by the maximu amount possible
 *
 * @throws GenesisRunException
 * @throws AbortException
 */
public void cFullMaxResponse() throws GenesisRunException, AbortException
{
        cFull(getMaxFlowIn() - getCurrentFlowIn());
}
```

4.5.3.3 **Ships**

This section deals with the additional & modified C event types that involve ships. In general, while connected to receiving and loading ports, ships behave just like storage tanks. Their behaviour when voyages are taken into account is more complicated – the results of the Full and Empty states are modified:

4.5.3.3.1 $C_{End\_of\_Voyage}$

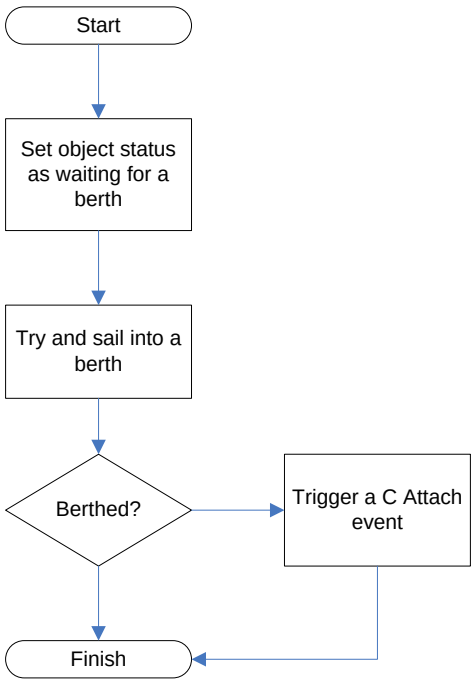

Figure 16. Handling a $C_{End\_of\_Voyage}$

The ship has arrived at a port. It tries to attach to berth and connect to the port flow structure. If it can't attach immediately, it waits in a queue for the next available berth.

Table 13. Code for handling $C_{End\_of\_Voyage}$

```
/**
 * End of laden voyage event
 *
 * @throws GenesisRunException
 * @throws AbortException
 */
public void cEndOfLadenVoyage() throws GenesisRunException, AbortException
{
        setVoyageStatus(VoyageTypes.WAITING_FOR_BERTH);

        final boolean berthed = connectToPort(getCurrentRoute().getEndPort());

        setCurrentRoute(null);

        if (berthed)
        {
                cAttach();
        }
}

/**
 * The end of unladen voyage event
 *
 * @throws GenesisRunException
 * @throws AbortException
 */
public void cEndOfUnladenVoyage() throws GenesisRunException, AbortException
{
        setVoyageStatus(VoyageTypes.WAITING_FOR_BERTH);

        final boolean berthed = connectToPort(getCurrentRoute().getStartPort());

        setCurrentRoute(null);

        if (berthed)
        {
                cAttach();
        }
}
```

4.5.3.3.2 C_{Empty}

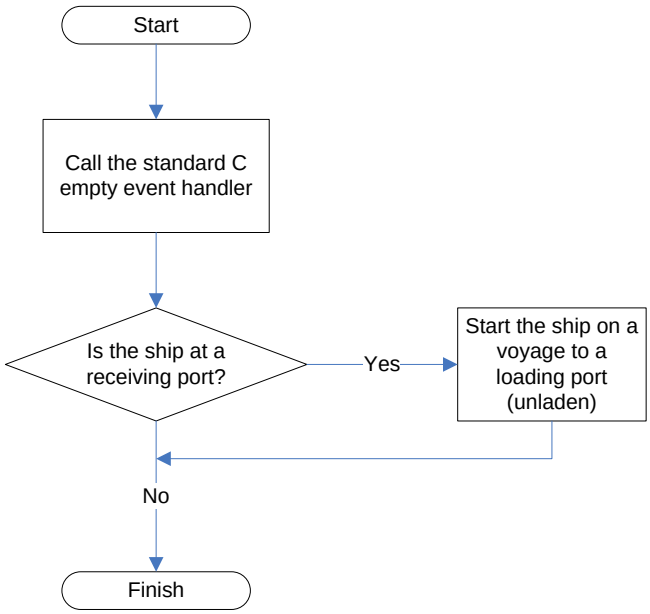

Figure 17. Handling a C_{Empty} event for a ship

If the ship is at a Loading port, try to increase the flow into the ship. If it is at a receiving port, start an unladen voyage.

Table 14. Code for handling C_{Empty}

```
/**
 * @throws AbortException
 * @throws GenesisRunException
 * @see com.stchedroff.genesis.core.structure.flow.FlowEvents#cEmpty()
 */
@Override
public void cEmpty() throws GenesisRunException, AbortException
{
	super.cEmpty();

	switch (getCurrentPortType())
	{
		case RECEIVING_PORT:
			startUnladenVoyage();
			break;
	}
}
```

4.5.3.3.3 C_{Full}

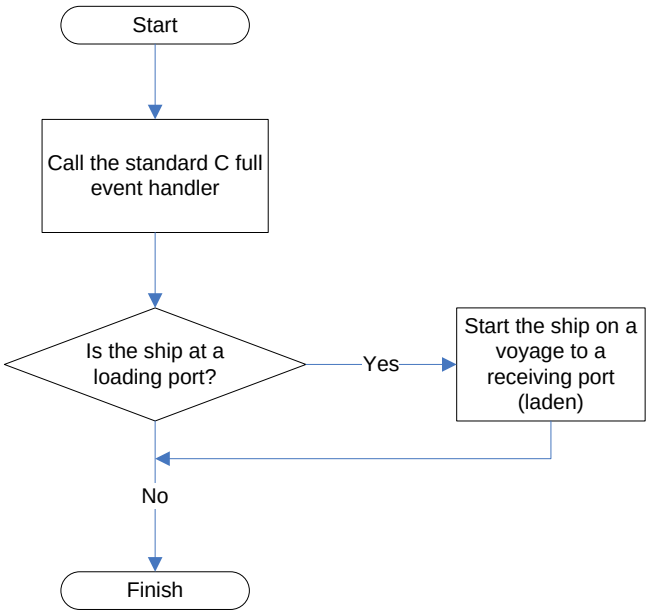

Figure 18. Handling a C_{Full} event for a ship

If the ship is at a Loading port, start a laden voyage.

Table 15. Code for handling C_{Full} For a ship

```
/**
 * @see com.stchedroff.genesis.core.structure.flow.FlowEvents#cFull()
 */
@Override
public void cFull() throws GenesisRunException, AbortException
{
        super.cFull();

        switch (getCurrentPortType())
        {
                case LOADING_PORT:
                        startLadenVoyage();
                        break;
        }
}
```

4.5.3.3.4 C_{Attach}

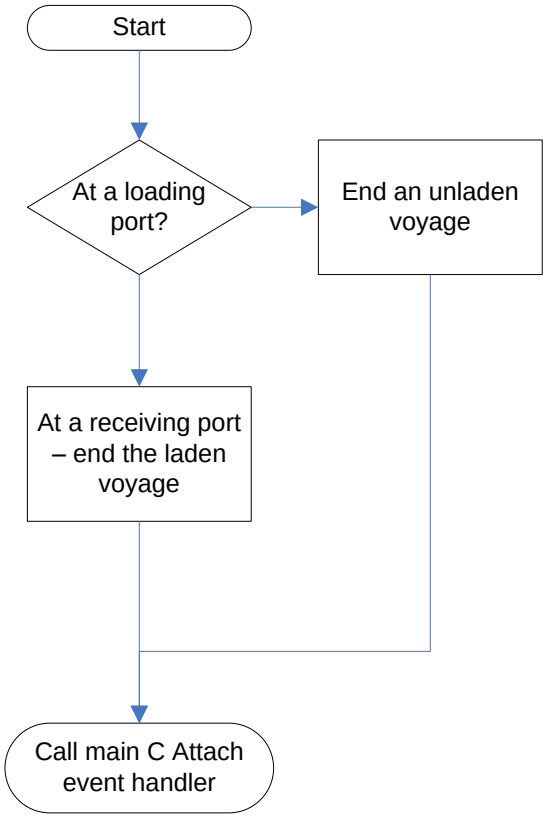

Figure 19. Handle C_{Attach} event for a Ship

The ship actually attaches to the flow structures at the ports, having found an empty berth.

Table 16. Code for handling C_{Attach} For a ship

```
/**
 * @see com.stchedroff.genesis.core.structure.flow.FlowEvents#cAttach()
 */
@Override
public void cAttach() throws GenesisRunException, AbortException
{
        switch (getCurrentPortType())
        {
                case LOADING_PORT:
                        endUnladenVoyage();
                        break;
                case RECEIVING_PORT:
                        endLadenVoyage();
                        break;
        }

        super.cAttach();
}
```

4.6 Making Flow Changes

There are four types of changes that need to be considered to implement the state changes outlined above.

4.6.1 Increasing the flow into the object

If we wish to increase the flow into the object, this can only be done if there is material in the attached objects "upstream" of the object in question. We can discount, therefore, input objects that are empty (Figure 200).

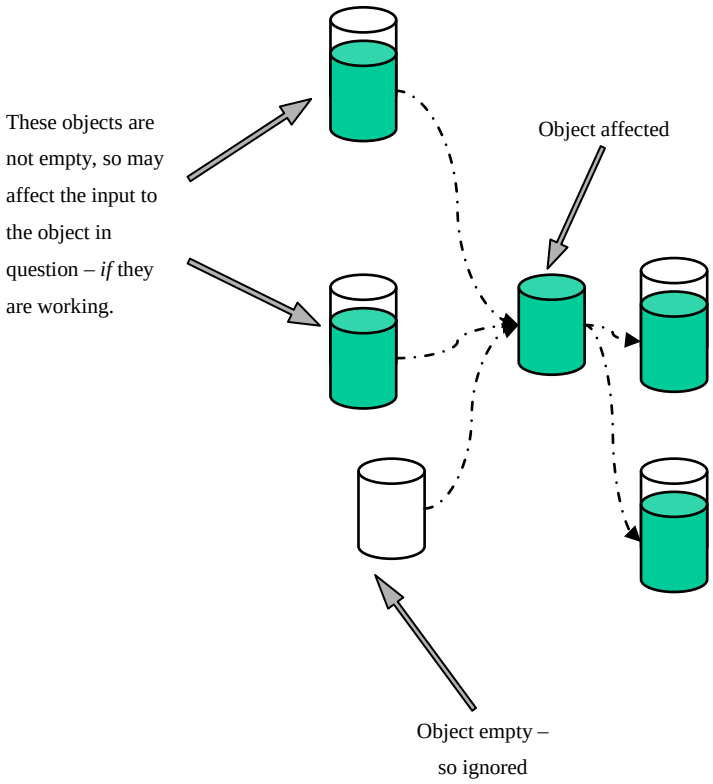

Figure 20. Empty Objects & Inputs

If the input object has some contents, then we can increase the flow from it, up to its maximum flow rate (Figure 211).

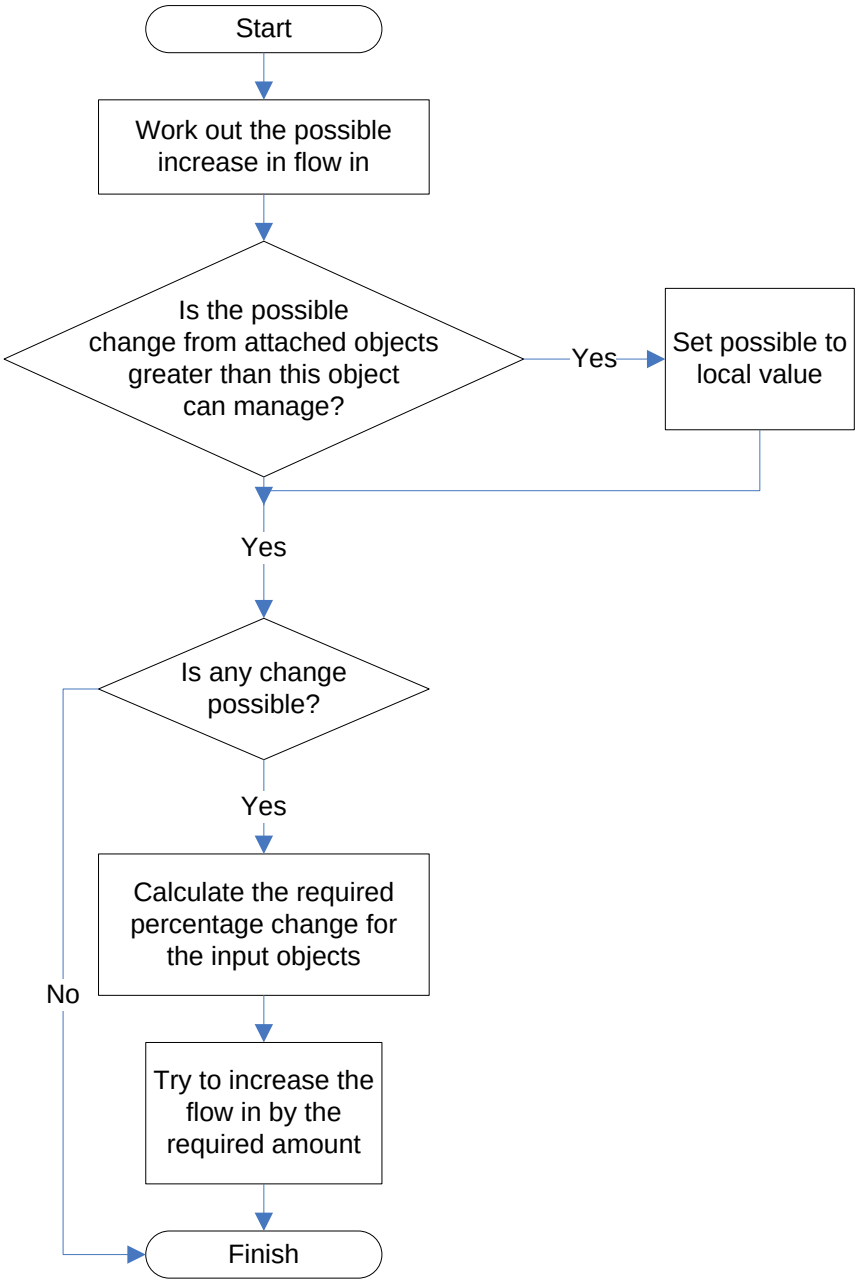

Figure 21. Increasing Flow In

The potential increase is the sum of the potential increase in flow rate.

Table 17. Code For Increasing Flow In

```
/**
 * Increase the flow into the object, if possible
 *
 * @param required
 *          The amount of extra flow required
 * @return The amount remaining
 */
protected double increaseFlowIn(double required)
{
        final double possible = getPotentialIncreaseInFlowInFromObjects();
        final double localPossible = getCurrentPotentialIncreaseFlowIn();
        final double previous = getCurrentFlowIn();

        if (required > localPossible)
        {
                required = localPossible;
        }

        // Nothing can be done
        if (localPossible == 0.0)
        {
                final double realInput = getCurrentFlowFromInputObjects();
                setCurrentFlowIn(realInput);
                return required;
        }

        // The default is to reduce by 100% of possible
        double percentageRequired = 1.0;
        double factor = 1.0;

        if (possible > localPossible)
        {
                factor = localPossible / possible;
                // If this is more than we require, reduce only by what is required
                if ((possible > required) && !Globals.aproxEqual(possible, required))
                {
                        percentageRequired = required / localPossible * factor;
                }
                else
                {
                        percentageRequired = percentageRequired * factor;
                }
        }
        else
        {
                // If this is more than we require, reduce only by what is required
                if ((possible > required) && !Globals.aproxEqual(possible, required))
                {
                        percentageRequired = required / possible;
                }
                else
                {
                        percentageRequired = percentageRequired * factor;
                }
        }

        increaseFlowFromInputs(percentageRequired);

        final double realInput = getCurrentFlowFromInputObjects();

        setCurrentFlowIn(realInput);

        return required - (realInput - previous);
}
```

4.6.2 Decreasing the flow into an object

Increases are done on the basis of what is the best that can be achieved. Decreases in input are absolute – the required decrease is always carried out

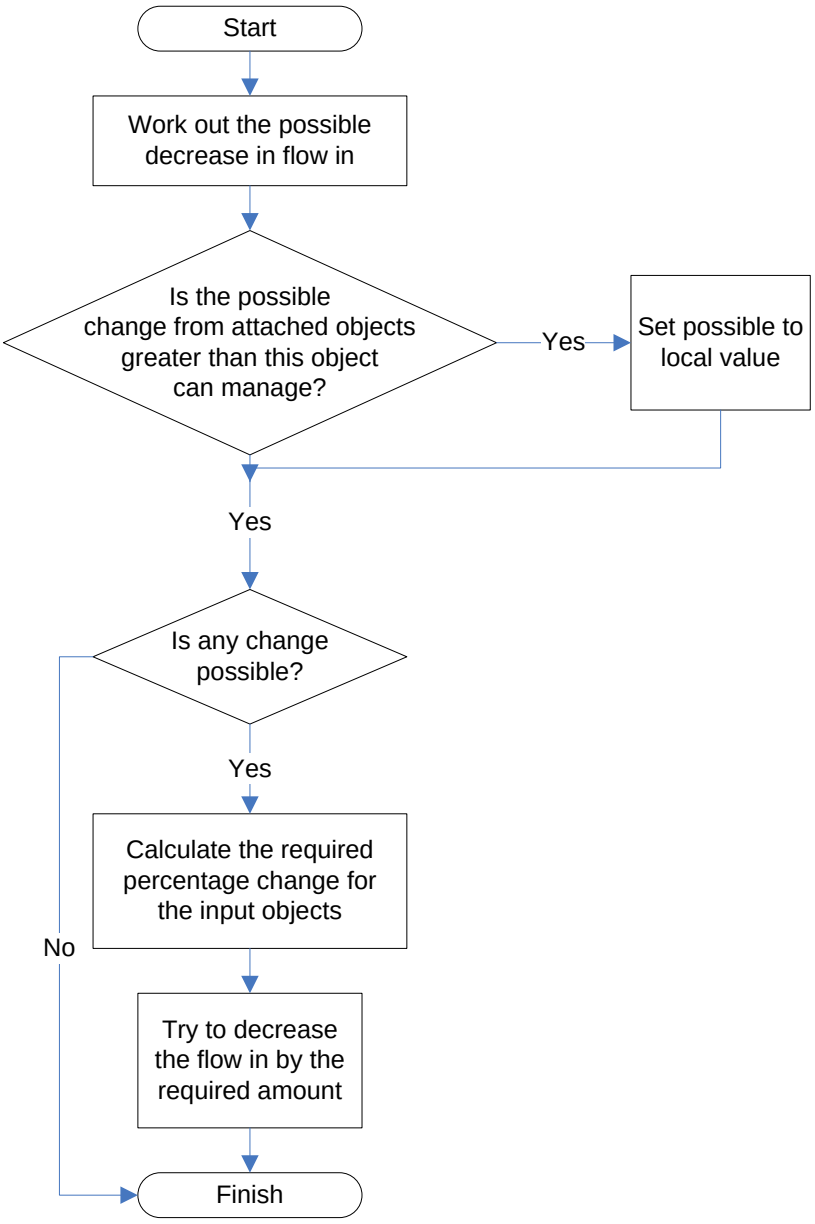

Figure 22. Decreasing Flow In

The decreases in flow can only change the rate of flow in from objects that are flowing – so again, empty objects can be ignored.

Table 18. Code For Decreasing Flow In

```
/**
 * Try to decrease flow in by a specified amount
 *
 * @param required
 *            The amount of decrease required
 * @return The amount remaining
 */
protected double decreaseFlowIn(double required)
{
	double factor = 1.0;
	double possible = getPotentialDecreaseInFlowFromObjects();
	final double localPossible = getCurrentPotentialDecreaseFlowIn();
	final double previousFlowIn = getCurrentFlowIn();

	if (required > localPossible)
	{
		required = localPossible;
	}

	if (possible > localPossible)
	{
		factor = localPossible / possible;
		possible = localPossible;
	}

	// Nothing can be done
	if (possible == 0)
	{
		final double realInput = getCurrentFlowFromInputObjects();
		setCurrentFlowIn(realInput);
		return required;
	}

	// The default is to reduce by 100% of possible
	double percentageRequired = 1.0;

	// If this is more than we require, reduce only by what is required
	if ((possible > required) && !Globals.aproxEqual(possible, required))
	{
		percentageRequired = required / possible * factor;
	}
	else
	{
		percentageRequired = percentageRequired * factor;
	}

	decreaseFlowFromInputs(percentageRequired);

	final double realInput = getCurrentFlowFromInputObjects();

	setCurrentFlowIn(realInput);

	return required - (previousFlowIn - realInput);
}
```

4.6.3 Increasing the flow out of an object

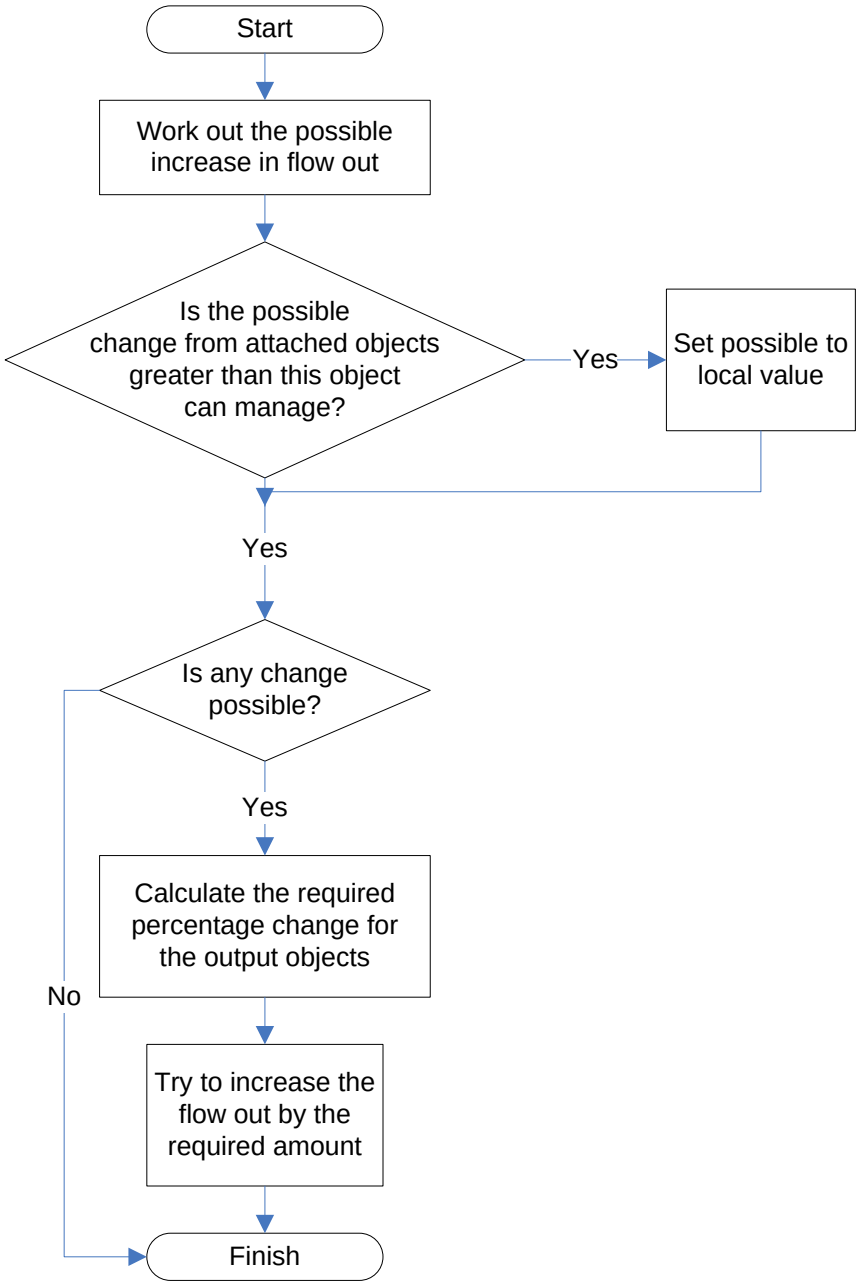

Figure 23. Increasing Flow Out

We look for all attached objects – "downstream" that are not already receiving input at the maximum rate.

Table 19. Code For Increasing Flow Out

```
/**
 * Increase flow out of the object
 *
 * @param requested
 *            The amount of increase requested
 * @return The amount remaining
 */
protected double increaseFlowOut(final double requested)
{
        double actual = requested * getConversionFactorDown();
        final double previous = getCurrentFlowOut();
        double possible = getPotentialIncreaseInFlowOutToObjects();
        final double localPossible = getCurrentPotentialIncreaseFlowOut();
        double factor = 1.0;

        if (actual > localPossible)
        {
                actual = localPossible;
        }

        if (possible > localPossible)
        {
                factor = localPossible / possible;
                possible = localPossible;
        }

        // If there is no possible change, or the request is negative/zero...
        if (possible == 0)
        {
                final double realOutput = getCurrentFlowIntoOutputObjects();
                setCurrentFlowOut(realOutput);
                return requested;
        }

        // The default is 100% of the possible amount
        double percentageRequested = 1.0;

        if ((actual < possible) && !Globals.aproxEqual(actual, possible))
        {
                percentageRequested = actual / possible * factor;
        }
        else
        {
                percentageRequested = percentageRequested * factor;
        }

        increaseFlowToOutputs(percentageRequested);

        final double realOutput = getCurrentFlowIntoOutputObjects();

        setCurrentFlowOut(realOutput);

        // setCurrentFlowOut(getCurrentFlowOut() + result);

        return (actual - (realOutput - previous)) * getConversionFactorUp();
}
```

This means that output objects that are full can be ignored:

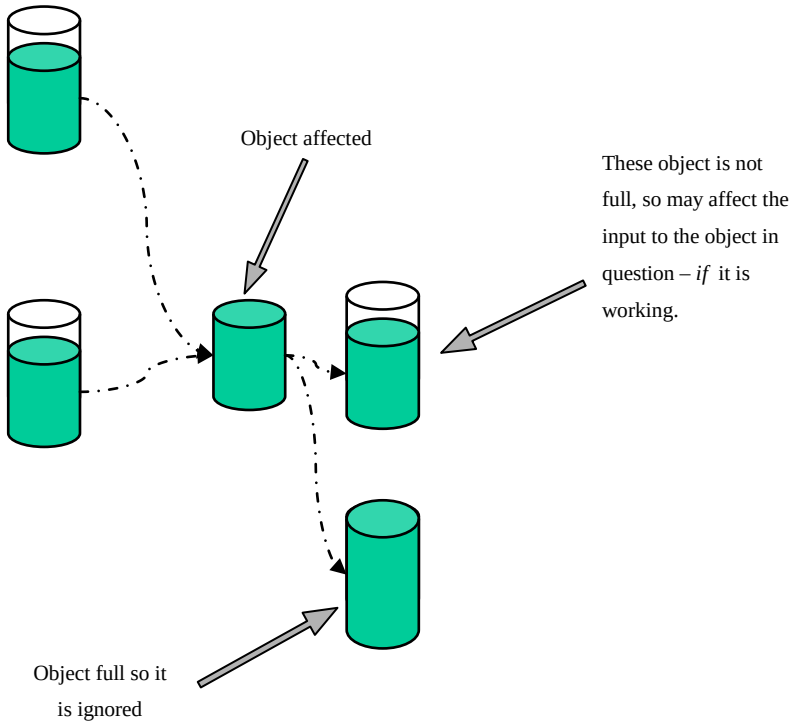

Figure 24. Effects of a full object on modifying output

4.6.4 Decreasing the flow out of an object

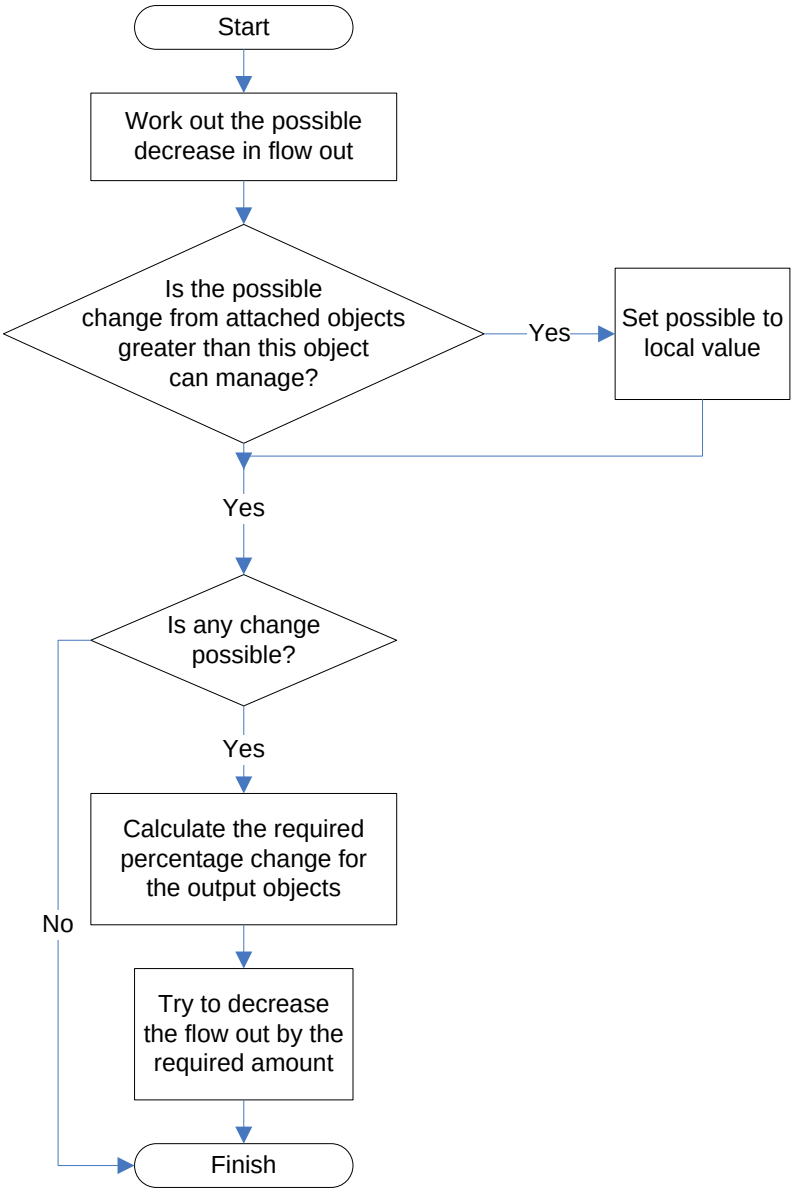

Figure 25. Decreasing Outputs

This is done as an absolute – the output from the object is reduced as required. The reduction per output object is done as a percentage of each of their existing flows. Again, full objects can be ignored, since no flow was going to them in the first place.

Table 20. Code For Increasing Flow Out

```
/**
 * Decrease the rate at which material flows out of the object (if possible)
 *
 * @param requested
 *            The amount by which we want to decrease the flow out of the object
 * @return The amount remaining
 */
protected double decreaseFlowOut(final double requested)
{
	final double previous = getCurrentFlowOut();
	double factor = 1.0;
	double actual = requested * getConversionFactorDown();
	double possible = getPotentialDecreaseInFlowOutToObjects();
	final double localPossible = getCurrentPotentialDecreaseFlowOut();

	if (actual > localPossible)
	{
		actual = localPossible;
	}

	if (possible > localPossible)
	{
		factor = localPossible / possible;
		possible = localPossible;
	}

	// If no change is possible, do nothing
	if (possible == 0) { return requested; }

	// Default is 100% of what is possible
	double percentageRequired = 1.0;

	if ((actual < possible) && !Globals.aproxEqual(actual, possible))
	{
		percentageRequired = actual / possible * factor;
	}
	else
	{
		percentageRequired = percentageRequired * factor;
	}

	decreaseFlowToOutputs(percentageRequired);

	final double realOutput = getCurrentFlowIntoOutputObjects();

	setCurrentFlowOut(realOutput);

	return (actual - (previous - realOutput)) * getConversionFactorUp();
}
```

Chapter 5. Implementation

5.1 Overview of the Structure Implementation

Initially C++ was chosen. This was latter change to Java. The reason for the change was that Java has the advantage of being platform independent, offering higher productivity in writing code and having a large library of useful structures, such as PriorityQueue. In recent versions (1.5 and higher) Java has achieved speeds very close to that of C++ for a number of tasks (Lewis & Neumann, 2004).

5.2 Basic structure

The main advantage of the object-oriented approach to structure is that functionality is developed once and then inherited by all those structure requiring it. Typically the resulting structure is tree like. The major decisions are at what level to put particular functionality in the tree – too low and duplication of capability is required, too high and object will have properties they do not require.

A subsidiary requirement was to only use single inheritance. While multiple inheritance is standard in C++ (and other languages) it can lead to structural problems and debugging issues. In particular, if a parent class is inherited twice, care needs to be taken to insure that there is only one instance and is referenced correctly - the simpler structure avoids this. This also means the

structure was re-implemented easily in Java and is compatible with virtually any object oriented language.

Analysis of the various structures to be modelled showed that there were a number of data items common to all objects in the model. From this it was an obvious step to define a basic “Object” type, common to all items in the model.

The second point from this analysis was that there was a split between the equipment that handles the gas & LNG and the meta-structures that contain them. These higher structures are the ports and the shipping routes that contain them. This leads to the idea of a “Flow Structure” from which all the equipment is derived.

The ports themselves are divided into loading and unloading. This is because different equipment is required for each port and the direction of flow is important – In a loading port the flow is always towards the jetties, and in a receiving port away from them.

The separation of structures derived from the flow structures was more difficult and underwent several revisions. The following options were considered –

1. All objects inherit the same properties – all the functionality would be placed in a “Flow Structures”.
2. Split the flow handling into two classes – “Equipment” and “Connectors”. In this paradigm the LNG and Gas types would be handled by setting flags in the objects.
3. Splitting the equipment into the three classes.

Keeping everything in the flow structure(s) was the option finally selected, after a number of revisions. The initial decision was to split the Connector type from the others. This was done in partly because of functionality issues (as above) and partly because of programming convenience. In the end, simplifying the code gave the highest benefits in terms of testing and maintenance.

Similar considerations were linked to the idea of splitting the equipment types into Gas Plant & LNG Plant – the behaviour is different and it is easier to handle it by splitting the class definitions.

5.3 Points of interest in the structure implementation

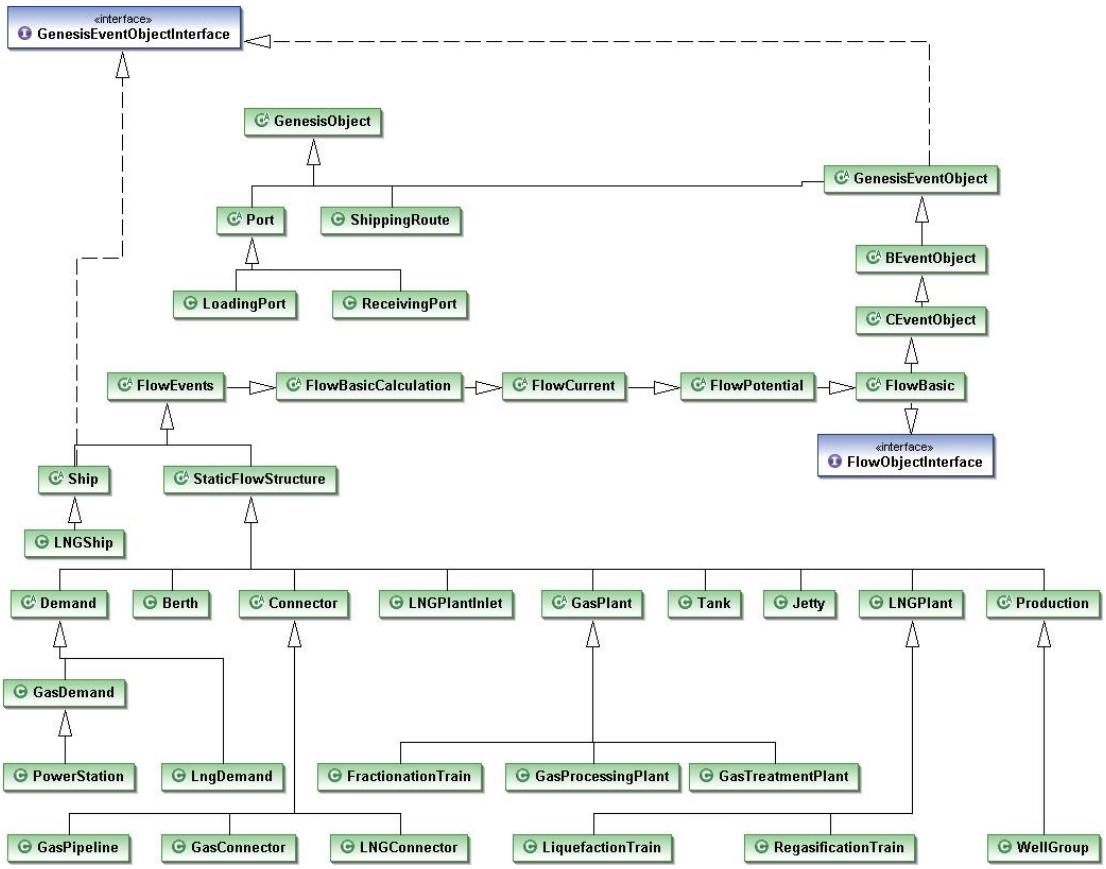

Figure 26. Overall Structure

This section provides an overview of interesting features in the implementation – much of the structure is a relatively straightforward implementation of the properties for each item. The overall structure is given in Figure 26.

5.3.1 GenesisObject

This structure holds the most basic information for each object – the object, name, dates, and the parent object (such as a port)

5.3.1.1 **The Flow Structures**

This is the set of classes titled *Flow<ClassName>*. The reason that there is more than one class, despite the linear inheritance between them, is that the amount of code became unmanageable for one class.

This level in the hierarchy provides the basic mechanisms to deal with object that have flows of gas or LNG passing through them. The highest level versions of the functions to handle C_{Normal}, $C_{EmptiedToNormal}$, $C_{Refilled}$, C_{Empty} and C_{Full} are implemented.

In addition, the Flow Structure implements the connection to Connector type objects (see below). A single Flow Structure can have several input and several output Connectors.

5.3.1.2 **LNG Ship**

The ships are treated as a type of flow structure, with overloaded functions to handle the C events C_{Attach}, C_{Empty} and $C_{Full.}$ This is in addition the functions to handle the events that are specific to LNG Ships - $C_{End\ of\ Laden\ Voyage}$ and $C_{End\ of\ \ Unladen\ Voyage}$.

5.3.1.3 **Gas Plant**

This level of structure is used to hold the properties for all the equipment that handles natural gas in its normal state – primarily the shrinkage and modification factors representing the losses incurred in transport and processing.

5.3.1.4 **LNG Plant**

This structure is used to hold the properties for all the equipment that handles natural gas in its liquid state – the fuel factor and GCV/NVC modification factors that track the reduction in energy value as lighter fractions boil off from the LNG.

5.3.1.5 **Connector**

This is another sub type of the Flow Structure – items that have only one input and output. There are three type of Connector – Gas Connectors, Gas Pipelines and LNG Connectors.

5.3.2 Shipping Route

This is a simple structure, linking two ports to each other. The only parameter of interest is the distance between the two ports. In future versions a more sophisticated version will be implemented, containing a series of voyage legs, which can each have their own weather and other probabilistic effects.

5.3.3 Port

This level in the port structures holds the links to the Shipping Routes, and the lists of structure that are common to both Loading and Receiving Ports – Jetties, Berths, Tanks, Power Stations, Gas Pipelines, Gas Connectors and LNG Connectors.

5.3.4 Loading Port

Well Groups, Gas Processing Plants, LNG Plant Inlets, Gas Treatment Plants, Fractionation Trains and Liquefaction Trains are all unique to Loading ports and are held in this structure.

5.3.5 Receiving Port

The only structure that only occurs in Receiving Ports is Regasification Trains.

5.3.6 Simulation

The highest-level structure is a “Simulation”. This structure contains a complete layout to be modelled and all the associated data – the list of ships, ports and data from the runs. In addition, the functionality to actually run a model is implemented as functions of this class. This facilitates having more than one model open and running at the same time in the application.

The executive is implemented as a set of functions in the Simulation class. The highest level is the Generate() function – this is used to generate an ADP using a fixed contingency value for each voyage, to represent probabilistic events such as weather and breakdowns.

5.3.6.1 **Start up**

The log is emptied, then it loops through all the objects in the model, running their own PreRun() functions, resetting their internal values. The runtime list of B events is generated from the lists of B events for each object in the model. The function that performs this task is CreateRunTimeBEventList(). Finally, the initial C events are calculated and placed in the queue, headed by the first to happen.

5.3.6.2 **A Phase**

Having obtained the time of the next event (B or C), the simulation clock is advanced to that point. This is unchanged from the classic three-phase model (Pidd, M. 1998).

5.3.6.3 **B Phase**

This phase of the model is a modification of the classic Three-Phase model (Stchedroff & Cheng, 2003). Because the B events have been reduced to planned, single time point occurrences, all that is required is to find the time for the start of the next B event.

5.3.6.4 **C Phase**

This is a modified version of the standard C Phase (Pidd 1998). C events that are occurring at the current time point are executed. Once done, their neighbours are added to the list of affected objects. In the event that the object is a Ship that is ready to sail on a laden voyage, the appropriate destination port is selected before it sails.

Then repeated passes are made through the objects affected by C events, to see if these have in turn caused other C events to occur. This continues until there are no more C events left to execute.

5.4 Logging the results

The data from the model is output to a variety of CSV formatted files. A main log file contains every action for the run. In addition, there are specific log files for the contents levels of the tanks, shut-ins for each of the objects, demand vs. supply for each demand point, production data per production point and ship operations on a per ship basis.

5.5 Simplifications

Compared with the full LNG supply chain modelling specification some simplifications were made. The major items were –

- The receiving port selection algorithm was simplified to sending the next ship to the port most behind with deliveries received.
- Boil off and its effects was not included
- All ships can go to all ports
- A number of fixed B events were not modelled – maintenance, dry docking and tides were not implemented
- Weather modelling was not implemented.

These items should not significantly affect the main aim of this work – to prove the 3 Phase modelling methodology for a continuous system, operating in the Generate phase.

5.6 Repeatability

An important part of the design was that all experiments are completely repeatable. Ordered structures are used for holding objects. The order of processing is randomised where required.

For example, the order in which a ship examines berths at a port when it tries to dock is determined by a pseudo random number generator. The list of berths is held in an ordered list. This means that all the logs etc from retried

runs will be identical, but that the order in which ships dock at particular berths will not be biased towards any one berth.

Chapter 6. Experimental design

A structure was devised for creating a matrix of experiments across a set of variables, given a maximum and minimum value (per variable). This collection of experiments is then implemented on the given model – the variable values being changed for each run.

6.1 Model Costing

6.1.1 Overview

The "costs" of each model configuration was considered as having two parts.

- The structural cost of the model – tank capacity, ship speed etc.
- Performance cost – demand satisfied, shut-in time etc.

Values are assigned to each of these items and a total cost for each model configuration is calculated at the end of the run.

The idea is that an optimal structure or ports and ships will ship the required quantity of LNG to the consumers without equipment being shut down (*shut-in*) or demand being missed. This tends to increase the size and number of the various components of the structure. Opposing this tendency to increased capacity/size/flow rate etc. is the cost of the model. What we are looking for is the cheapest structure that can deliver the required amount of LNG.

6.1.2 Costs

6.1.2.1 **Shut-in time**

This is the amount of time equipment spends shut down. This applies to all flow equipments upstream of the LNG tanks at each port. It is assumed that jetties, berths and their associated connectors will start and stop as ships arrive and leave. In order for the structure to be valid, it should have zero shut in time. This is taken as the full period of the model, not just the post run in period (the model is initially run for a period until the instabilities from the start up have dissipated). If there is any shut-in, the model is aborted. This matches the practise for real world systems where shut in during normal operation is generally not acceptable.

6.1.2.2 **Demand Unsatisfied**

This is the amount of LNG demanded by each receiving port but not delivered, creating a cost. Ideally this would also be zero, but in a real model, there would always be cargo despatched at the end of the run, that would not be delivered before the end date, for instance. To account for this, a threshold of ±100,000 m^3 of LNG was given for the demand unsatisfied value – below this threshold it is ignored. Beyond this, a penalty cost was applied to the overall model cost.

6.1.2.3 **Tank Trend**

An important factor in an LNG supply change is stability in the storage tanks at the ports. At the end of the model run, the trend for tank contents is calculated using the least-squares method. If the trend is changing by less than ±0.5 m^3 per hour, it is ignored. Otherwise a penalty is applied.

6.1.2.4 **Aborts**

If the model has a shut-in (see above), then it is aborted – once a shut-in has occurred; there is no point in continuing. The cost of the model is set to the extremely high figure of 10^{21} in the actual implantation.

If the model has a storage tank reach full or empty, there is also an abort. Again, the cost of the model is set to an extremely high value.

6.1.2.5 **Capacity variables**

Capacity variables represent the amount of material that an object could contain – such as a storage tank or a ship. They have a lower and upper bound. In the case of ships, if as part of an experiment, the value was lower than the lower bound, the ship would be discounted for that run. This allows evaluation of the effects of a varying number of ships as well as their size, which is a key variable in planning real world LNG operations.

6.1.3 Other Variables

Two other kinds of variables were considered in the initial work – Start Port & Slow Loading Time. These did not have any cost, so do not affect the final cost of the model.

6.1.3.1 **Start Port**

The Start Port refers only to ships and defines which port they start at.

6.1.3.2 **Slow Loading Time**

Slow Loading Time refers to the period of time at the end of the loading process that a ship loads at a very low rate. This is essentially topping up the tanks for lost material while waiting to start a voyage. This period of time is used to space out the ship voyages, allowing a model with a small number of large ships to deliver evenly spaced loads.

6.1.4 Experimental methodology

The experiments were run using the generate mode of the model – random events such as breakdowns and weather were not included. The aim was to study optimizing layouts of the ports and the number of ships between them.

The decision was made to concentrate on the number of ships, their capacity and the slow loading time. This is one of the major areas of supply chain design that is studied in the real world LNG industry.

The capacity of the storage tanks at each port was considered. However, the tank capacity is driven by the requirements of the system in real-world

operations, though there are some limits. The goal of system design is that the level in the tanks is broadly stable, but oscillating by amounts that closely related to the size of the ships coming and going from the port.

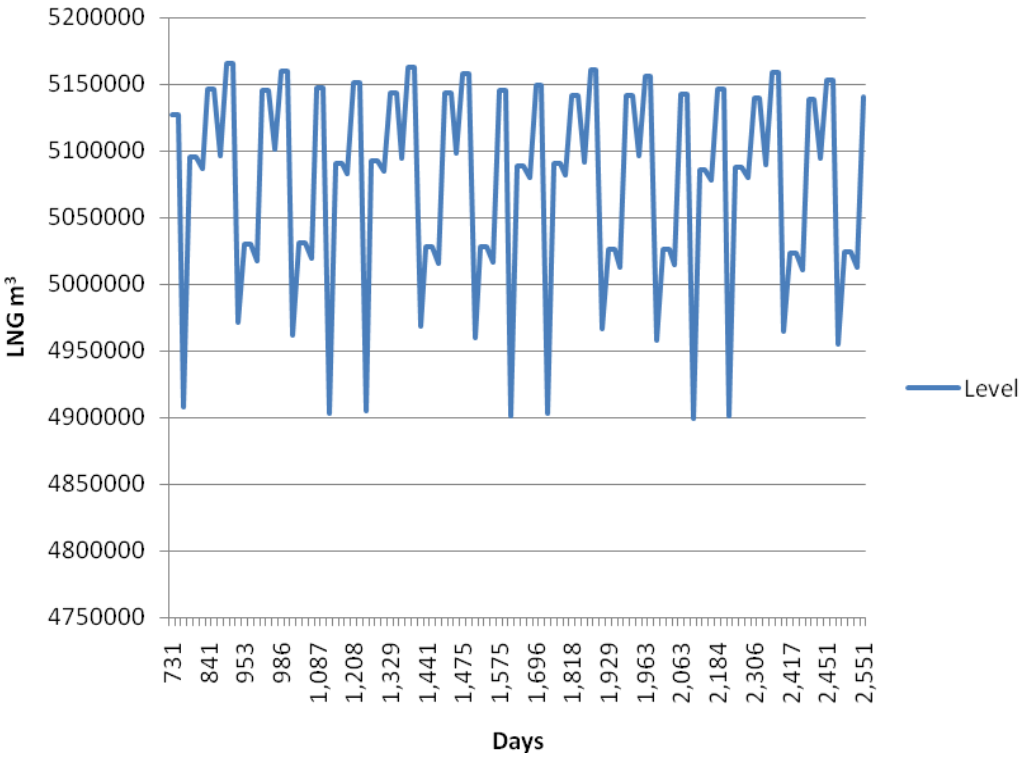

Figure 27. Typical Tank Level vs. Time (in days)

The figure (above) shows a typical tank level vs. time graph - it shows the tank level as broadly stable over the lifespan of the run, but varying by an amount that is consistent with the ship sizes for the particular model. This means that the dominant factor in sizing the tanks will be dealing with random events such as breakdowns and weather, which would be handled in the Testing Phase of modelling a supply chain.

Since we are working in the Generating Phase (at this point) it makes sense to give the tanks excess capacity (10^8 m^3 of LNG in this case) and modify the tank capacities to realistic values afterwards, based on the necessary buffer volume, deliveries and ship capacities that are optimal.

6.1.5 Optimisation

Two methods of direct search optimization were recommended (Fliege, J) – Nelder Mead (McKinnon 1999, Nelder & Mead 1965) and Multi-Directional (V. Torczon, 1991. Direct search methods can be used when either the computation of the derivative is impossible (noisy functions, unpredictable discontinuities) or difficult (complexity, computation cost). In the first cases, rather than an optimum, a not too bad point is desired. In the latter cases, an optimum is desired but cannot be reasonably found (M. H. Wright, 1996). This work falls into the first category.

The two search algorithms were implemented using the Apache common math library (Apache 1.2). Some small modifications were made; in the case of the Multi-Directional method it was found that the model could become trapped when all the values in the simplex were too similar (McKinnon 1999).

Both methods take the variables to be computed to be numbers; doubles in the case of the Apache Java implementation. In the case of the capacity variables, checking was added to catch values that were out of range. If a value was out of range, the "cost" of that configuration was given as a very large value (10^{15}), unless it was capacity for a ship, and the value was too low in which case the ship was marked as inactive and the full evaluation of the cost of that configuration was run.

The Start Port variables were handled by assigning an integer to each of the ports in the model. Generated values were fixed as these integers, and when new values were computed as part of the optimizing process, they were rounded down to the integer value.

To generate the values for the initial simplex, and subsequent additions to it, an uncorrelated pseudo-random number generator was used. The values for each variable were, of course, limited to the range for that variable.

After some initial experimentation, the variables selected for the following experiments were:

- Ship capacity – the amount of material that the ship could carry. If below the minimum, the ship is considered inactive and is ignored during the runs of the model.
- Separation between voyages – the amount of time between laden voyages.
- Start port – which port the ship starts at the beginning of the run. This was included so that the model would produce identical results for a given configuration when run and to investigate the sensitivity of the optimised delivery schedules to changes in start port.

The tank capacities in the ports were set to deliberately high, fixed values. In a real world LNG supply chain the emphasis in on reducing the number of LNG ships and their capacity. The cost of tank storage is relatively small. So, in designing facilities and planning operations, the ship operations are optimised and the tanks sized to support this configuration.
This means that for a give configuration in this series of experiments, the number of variables is 24 - the number of ships x 3.

Chapter 7. Experiments

Five main experiments were undertaken, with increasing complexity. They were used as a series of “gates” during the development of this work – each one more difficult and complex than the last

- Experiment A demonstrates the basic concepts in a simple 2 port model.
- Experiment B adds a second receiving port to test the receiving port selection and handling functionality.
- Experiment C uses a more complex design for the loading and receiving ports to test the behaviour of the model when events are triggered in a more complicated structure.
- Experiment D adds seasonal demand changes (B events).
- Experiment E demonstrates the Test model of the model.

7.1 Experiment A

A model was constructed with one loading port and one receiving port.

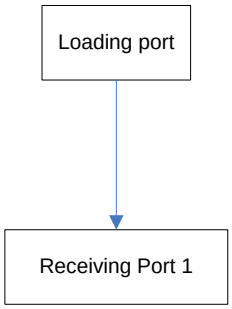

Figure 28. Experiment A overview

The loading port and receiving port were simple linear structures – a straight line of objects with one flowing into the next. The model run time was 5 years, with a 2 year run in period to allow the system to stabilise. There was a maximum of 8 ships. Variables were the number of ships, their starting ports, and their capacity and the size of the LNG tanks at each port.

The aim of this experiment was to test out the concepts involved, validate the model and assess the effectiveness of this design strategy.

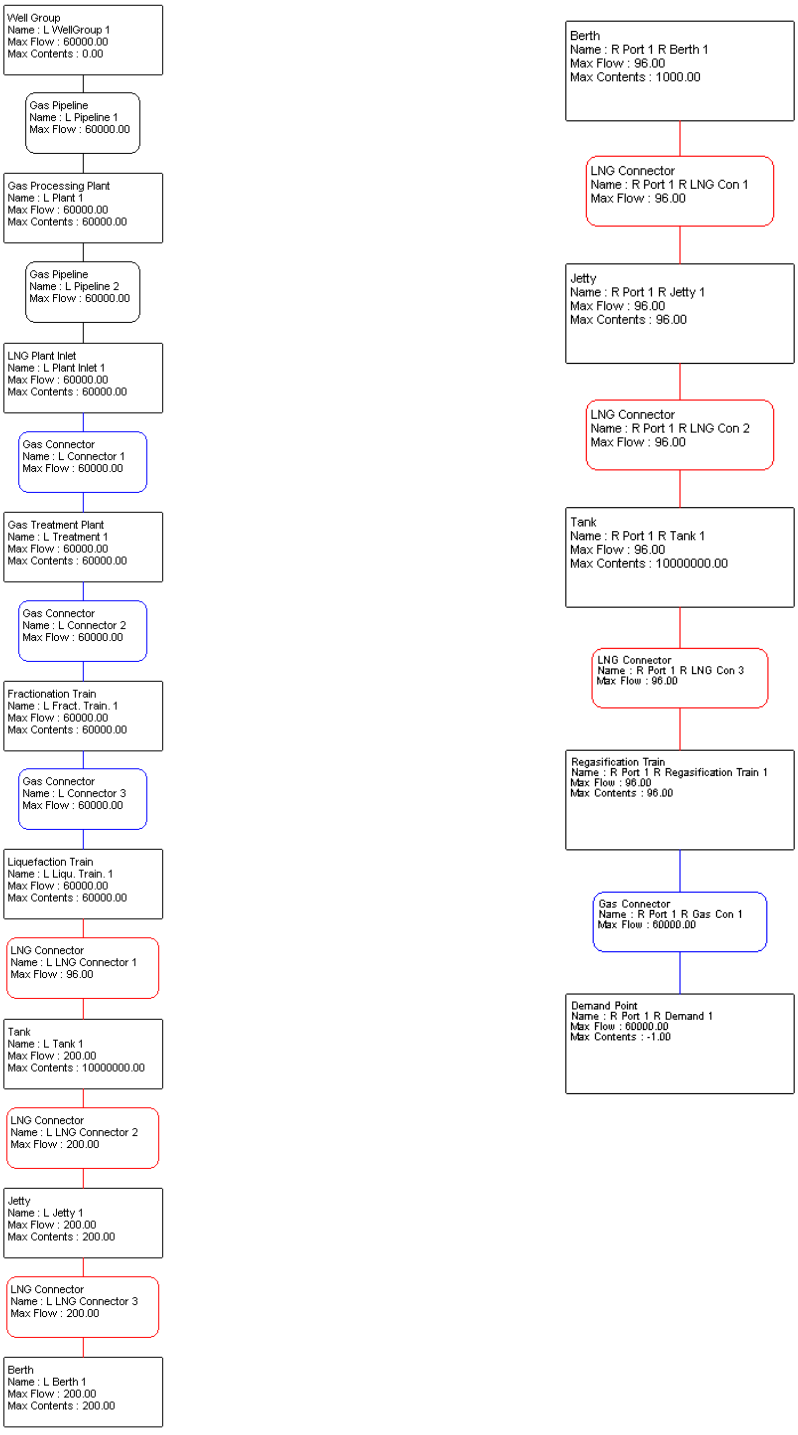

Figure 29. Experiment A: loading & receiving ports

7.2 Experiment B

A model was constructed with one loading port and two receiving ports.

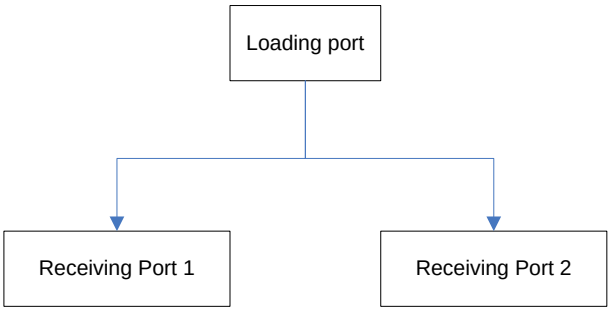

Figure 30. Experiment B overview

The loading port and receiving port were simple linear structures. Both receiving ports were identical.

The loading port was the same as the one used for Experiment A. The receiving ports are different – the flow rate between the berths and storage tanks are much higher than the rest of receiving model, allowing for a more realistic unloading pattern.

The model run time was 5 years, with a 2 year run in period to allow the system to stabilise. There was a maximum of 8 ships. Variables were the number of ships, their starting ports, and their capacity and the size of the LNG tanks at each port.

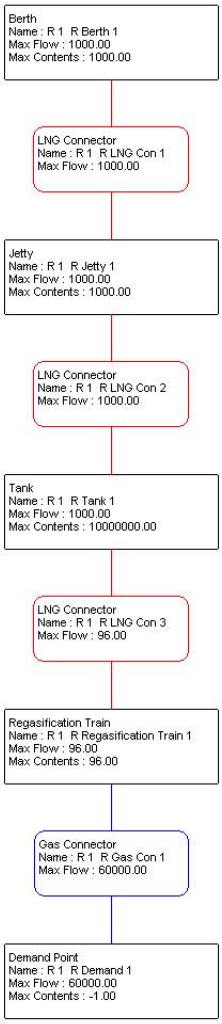

Figure 31. Experiment B receiving port

7.3 Experiment C

This experiment had one loading port and two receiving ports. The loading port was given reasonably complex structure equipment (generally) arranged in two streams with interconnection between the streams. This is representative of the complexity of real life facilities.

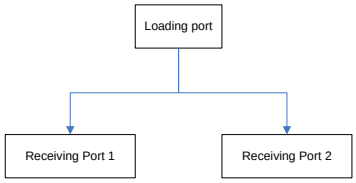

Figure 32. Experiment C - overview

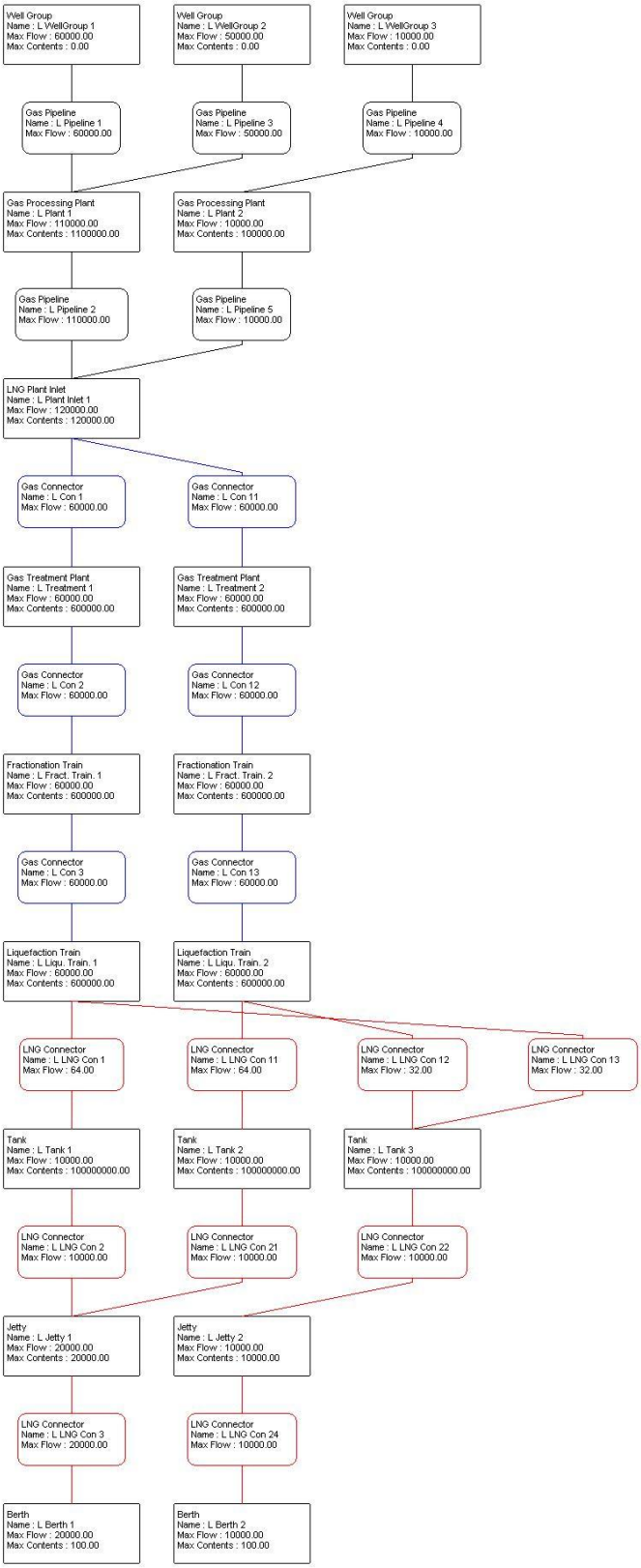

Figure 33. Experiment C – loading port structure

The receiving ports are identical, and somewhat less complex.

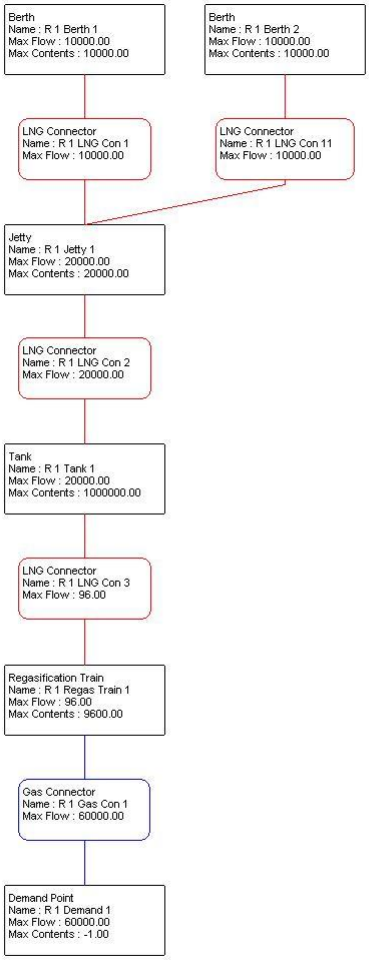

Figure 34. Experiment C – receiving port layout

Again, this is representative of real-life complexity.

7.4 Experiment D

The structure was identical to Experiment C, but demand variation was introduced. Demand was seasonally varied – for the first and last one hundred days of the year, demand was reduced by 25%. This was done with B events.

7.5 Experiment E

For this experiment, the Test mode for running the system was implemented. Firstly, contingency periods were added to end of each voyage. In the generate mode, these are a fixed amount. A delivery plan is generated (with no B events apart from demand changes) and then fed into the test mode run. In the test mode, all B events are active.

When running in the test mode, if the ship arrives late relative to the generated plan, the contingency period is shortened, so that the ship finally docks at the planned time. The figure below illustrates this process.

If the ship arrives so late that the contingency period is insufficient to cover the delay, then the delivery program has failed the test. For the purposes of this test, the contingency period was set at 3 days.

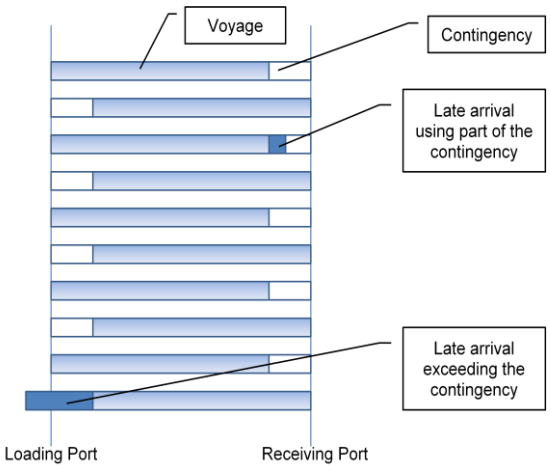

Figure 35. Contingency for ship voyages

A small modification was also made to the model methodology at this point – the delays that space the voyages out were modified from being slow loading to a delay before sailing on the laden voyage. This was done to simply the analysis of the ship movements.

The port and ship structure used for this test are identical to Experiment D. The additional events were breakdowns on all the fixed equipment port structure, apart from tanks, LNG and gas connectors and berths. Tanks and

berths are simple structures with no moving parts – the pumps that move material in and out are located in adjacent equipment in the model. Similarly, connectors are simple pipes. Failures associated with such structures in real-world situations are generally caused by the complex equipment associated with such structures, not a part of them.

The B events used were breakdowns affecting the flow through the objects concerned – 50% would mean a 50% reduction in the maximum possible flow through the object affected for the period of the event. The structures modelled in this work are generally complex in their real-life internal workings, consisting of many sub-components. Often the sub-components are duplicates organized in parallel to prevent a single failure causing a 100% loss of a capability.

The events were modelled with an exponential probability of occurring with a Mean Time Between Failure (MTBF) of 1 year. The Mean Time To Repair (MTTR) was 6 hours and the Mean Effect (ME) was a 50% failure. This is a much higher failure rate than would occur in real-world situations.

Chapter 8. Experimental Results

8.1 Experiment A

8.1.1 Nelder Mead Results

As a test of the effectiveness of the technique a test series was run for Nelder Mead; varying the number of starts and maximum number of evaluations between 50 and 2000.

The best model value found used 3 ships, with capacities of 516654, 380843 and 418728 m^3 of LNG. Interestingly the number of start seemed not to have much effect. The best result was found with 350 starts, though the number of evaluations was 1850.

A further point is that the performance is not discontinuous – suggesting that analysis of this kind could be performed by the Direct Search techniques.

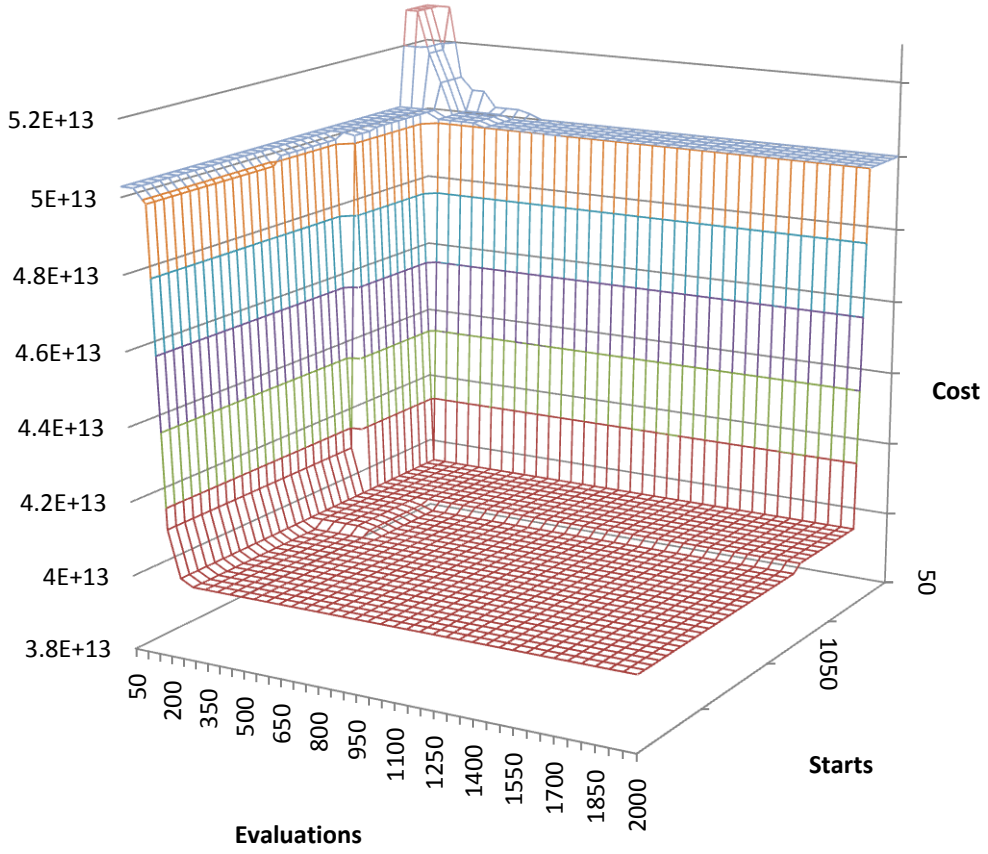

Figure 36. Experiment A - Nelder Mead for a range of values

8.1.1.1 **Best result**

The best solution showed very stable tanks for both loading and receiving ports.

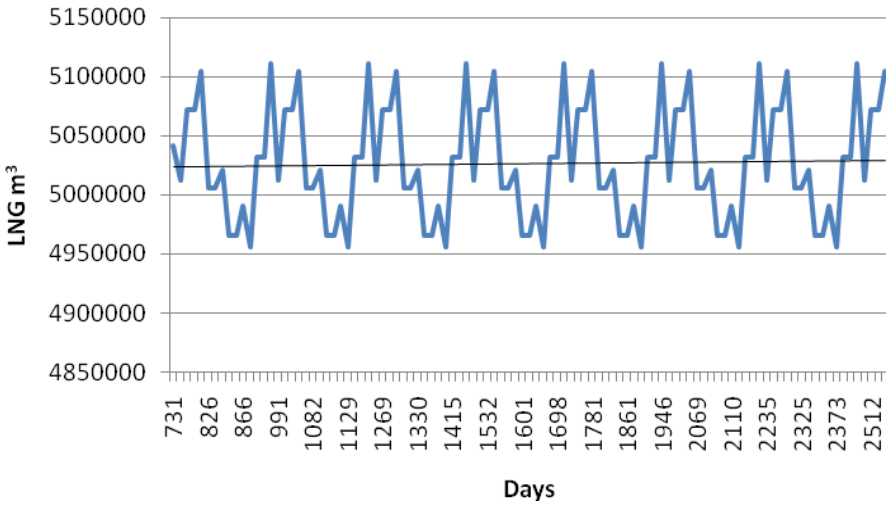

Figure 37. Experiment A - loading port tank levels

The trend line for the loading port is almost perfectly flat – an average decrease of 0.101059 m^3 of LNG per hour over the period of the model.

The level in the tank at the receiving port is completely flat – it doesn't change. This is because the model is simple - the maximum input and output values for the objects in the receiving port are all the same, and the arrival of ships at the receiving port is quick enough that the only the berth itself has fluctuations in level.

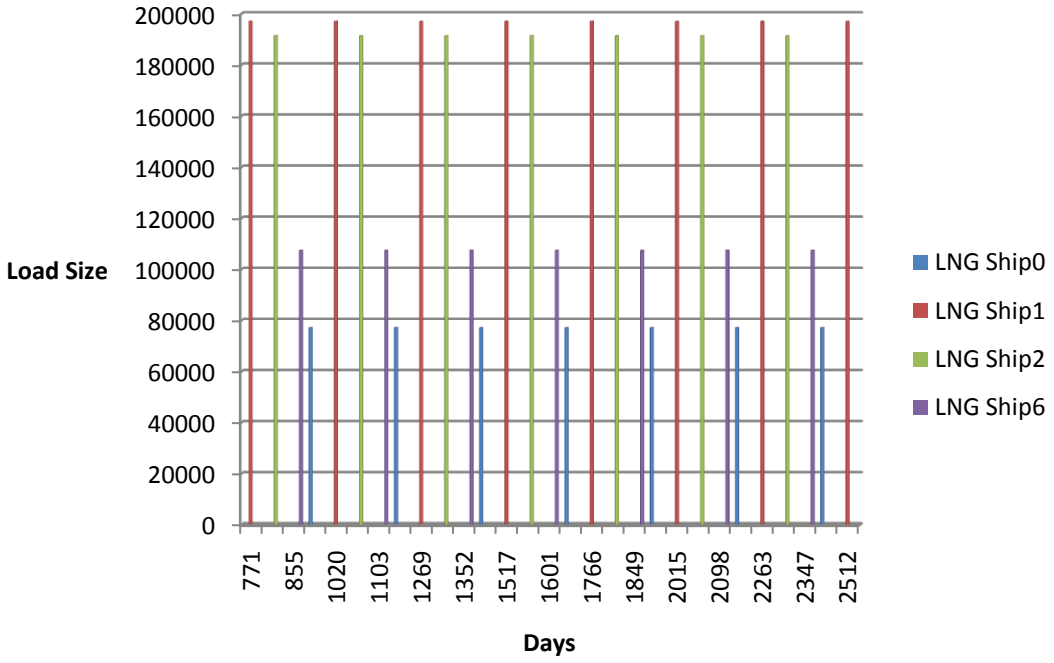

Figure 38. Experiment A – Nelder Mead delivery schedule

The delivery schedule shows a very regular sequence of deliveries being dispatched from the loading port. Demand was completely satisfied by this model.

8.1.2 Multi- Directional

An identical test series was run with maximum number of evaluations and starts between 50 and 2000, for Multi-Directional.

The best model value found used 2 ships, with capacities of 420086 and 421302 m^3 of LNG. This is considerably less capacity in total (and hence a cheaper result) than that found by the Nelder Mead (above).

The number of starts seemed not to have much effect. The best result was found with 350 starts, though the number of evaluations was 1850.

A further point is that the performance is not discontinuous – just as with the Nelder Mead method.

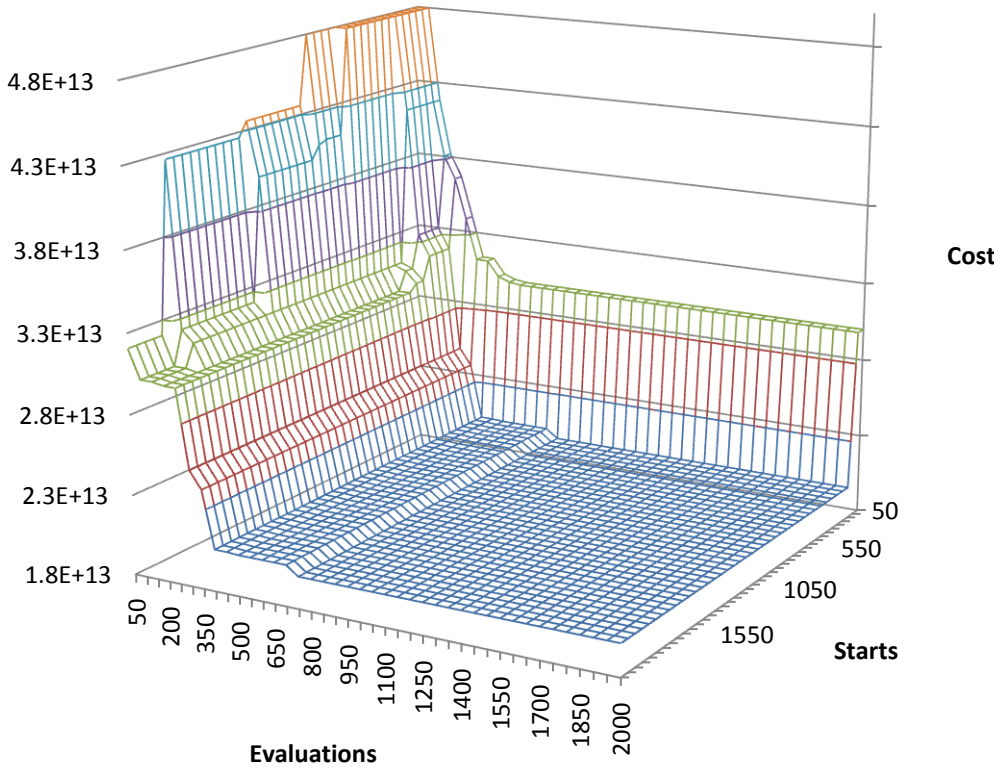

Figure 39. Experiment A - Multi-Directional for a range of values

8.1.2.1 **Best result**

The loading port contents were even more stable than for the Nelder Mead result.

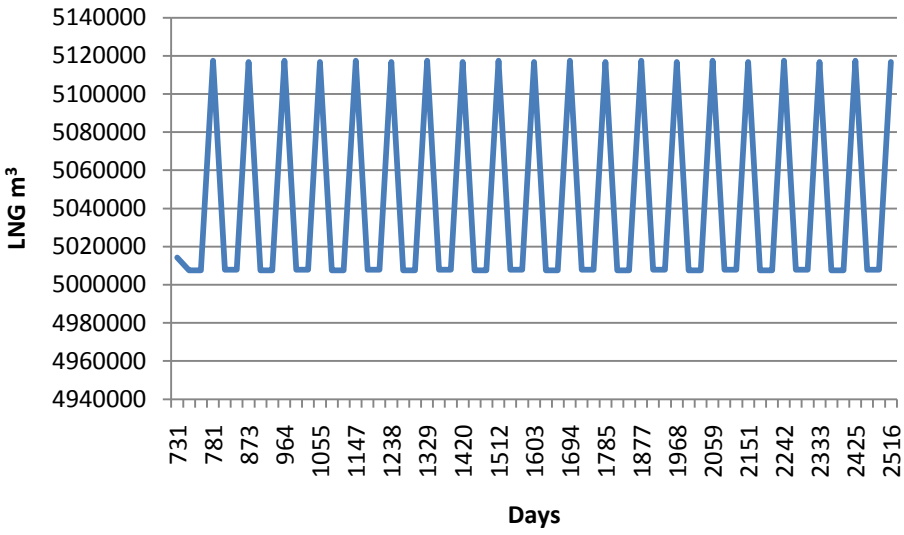

Figure 40. Experiment A - loading port tanks for Multi-Directional best result

This is due to the smaller number of ships and their equality in size. Again, the receiving port tank levels were unchanged over the run, due to the simple structure of the receiving port.

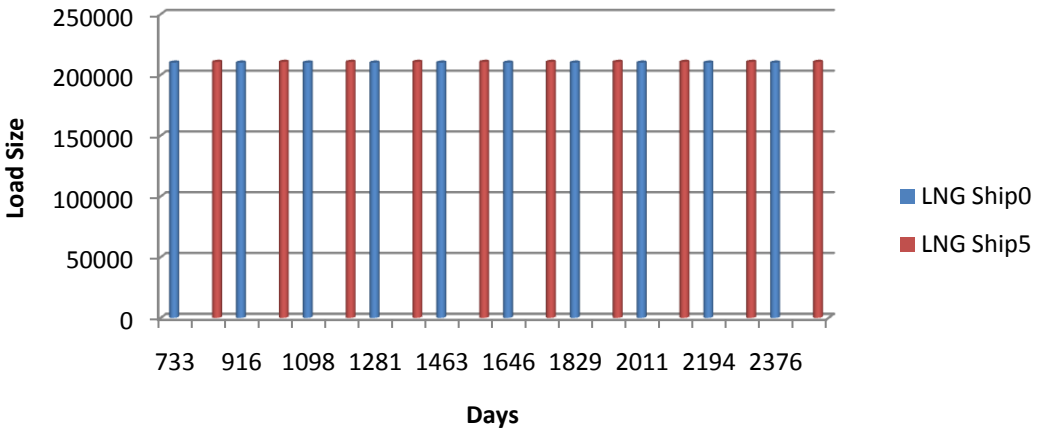

Figure 41. Experiment A – Multi-Directional delivery schedule

The delivery schedule was, again, very regular. Demand is completely satisfied.

8.2 Experiment B

8.2.1 Nelder Mead Results

A test series was run varying the number of starts and maximum number of evaluations between 50 and 2000

The best model value found used 5 ships, with capacities of 51317, 94562, 92165, 112230 and 488357 m^3 of LNG. The best result was found starting at 1150 starts and 700 evaluations (approx. since the experiment was undertaken with intervals of 50 for both starts and evaluations).

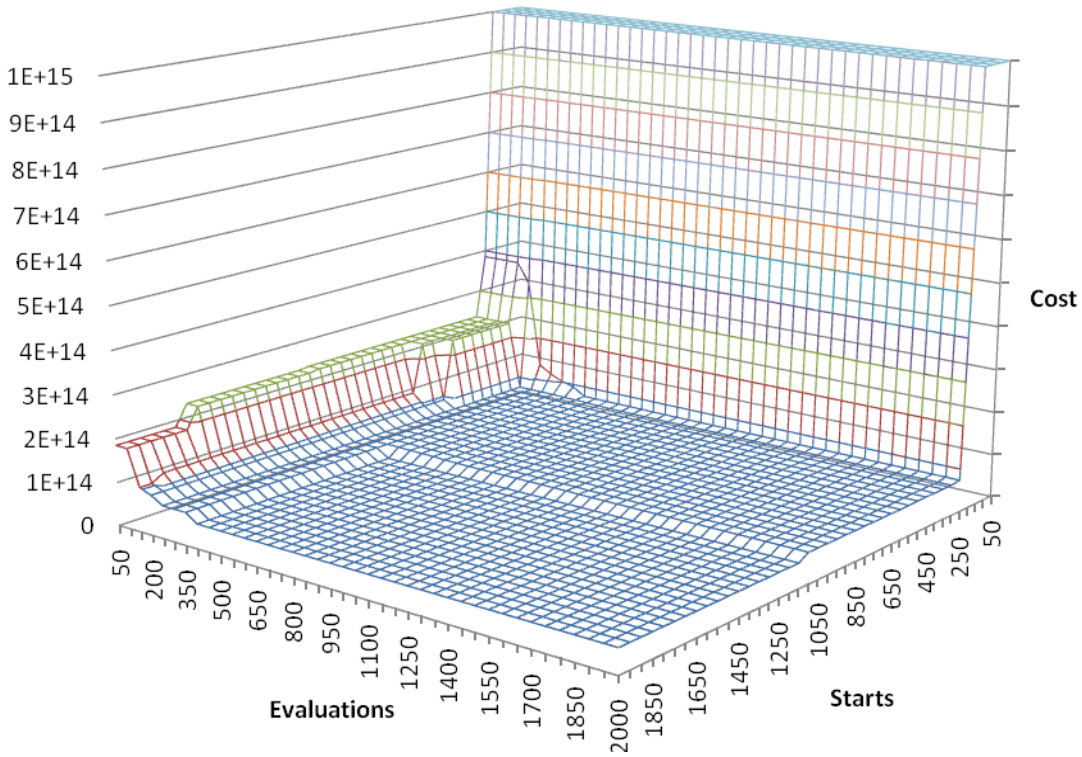

Figure 42. Experiment B - Nelder Mead for a range of values

The best value rapidly heads to a stable "floor" – indicating that further gains from increasing the number of starts and evaluations would probably not improve the results greatly.

8.2.1.1 **Best result**

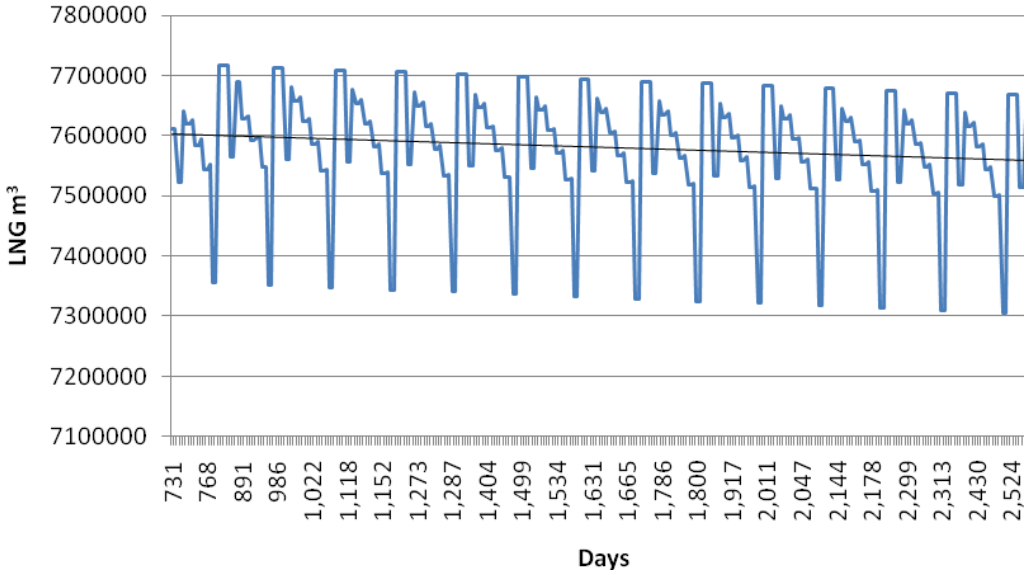

Figure 43. Experiment B – loading port levels

The loading port showed a slight downward trend in level - 0.856249 m^3 of LNG per hour on average. The two receiving ports showed slight gains, of 0.268398 m^3 per hour and 0.919182 m^3 per hour.

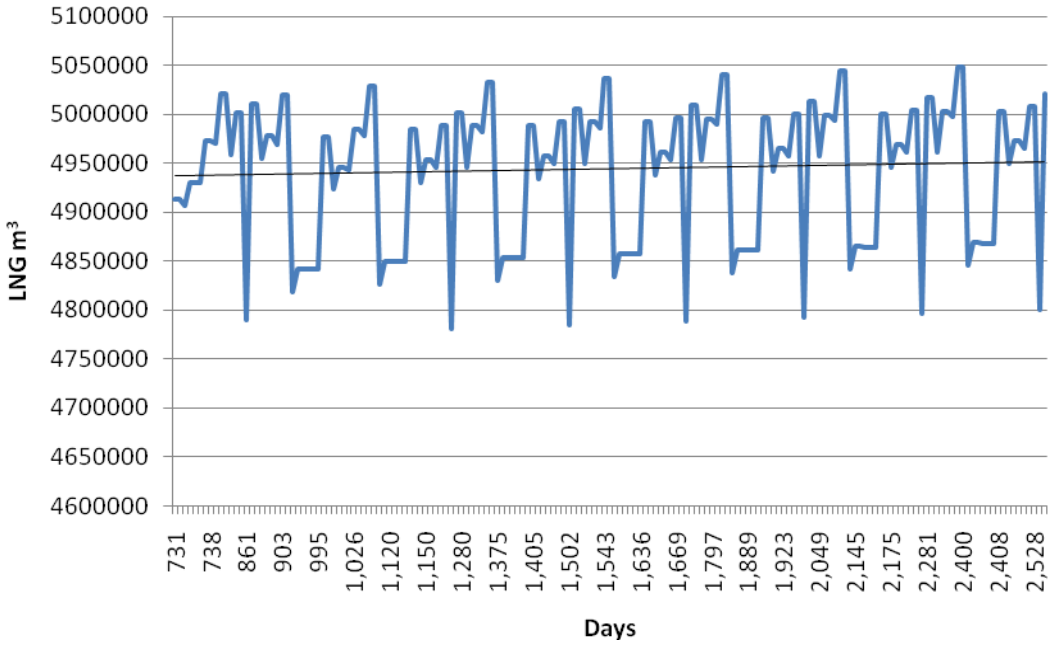

Figure 44. Experiment B – receiving port 1 levels

Overall, this shows a good level of stability – in an environment with random variables (weather etc), these small trends would be lost in the noise.

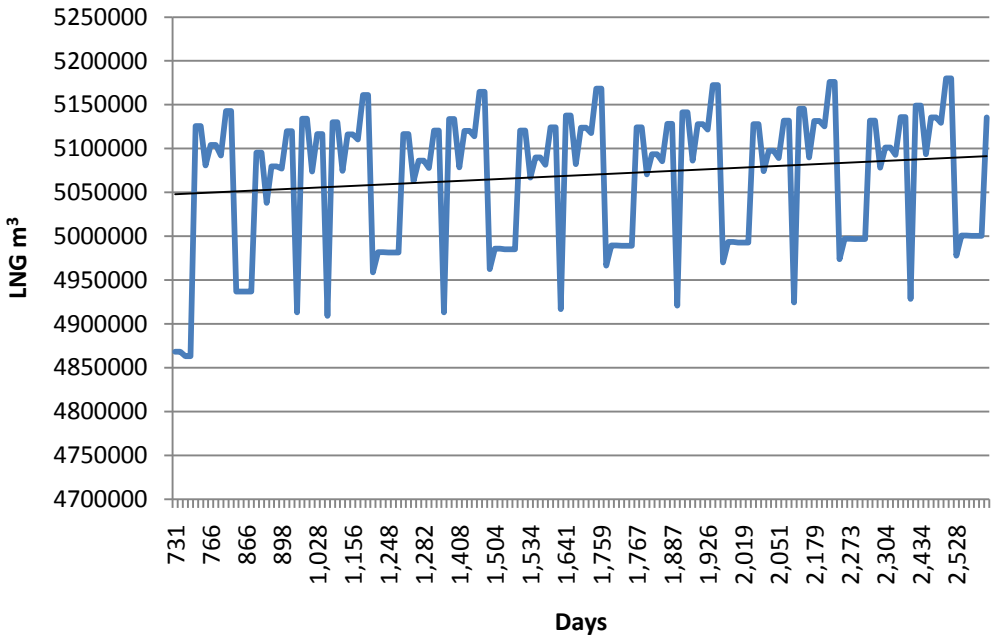

Figure 45. Experiment B - receiving port 2 levels

The delivery schedule was very regular and demand was completely satisfied. The solution picked by the tool with one large ship and 4 much smaller ones to smooth the flow is evident in the following figure.

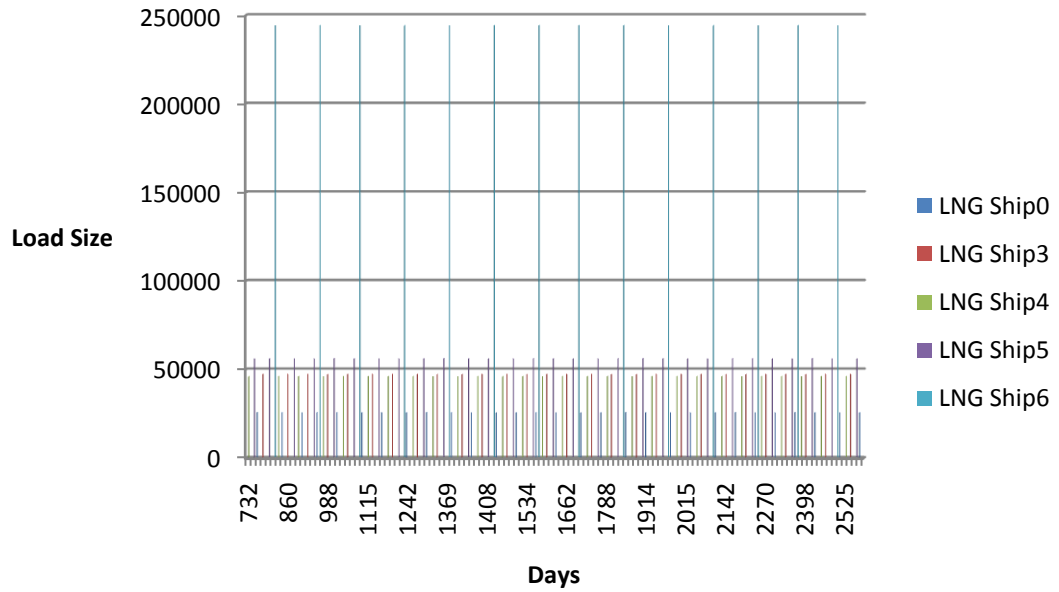

Figure 46. Experiment B – Nelder Mead Delivery Schedule

8.2.2 Multi-Directional Results

A test series was run varying the number of starts and maximum number of evaluations between 50 and 2000

The best model value found used 4 ships, with capacities of 283385, 151951, 131198, and 414444 m^3 of LNG. This is a considerably better result than for the Nelder Mead experiment above. The best result was found starting at 750 starts and 2000 evaluations (approximately since the experiment was undertaken with intervals of 50 for both starts and evaluations).

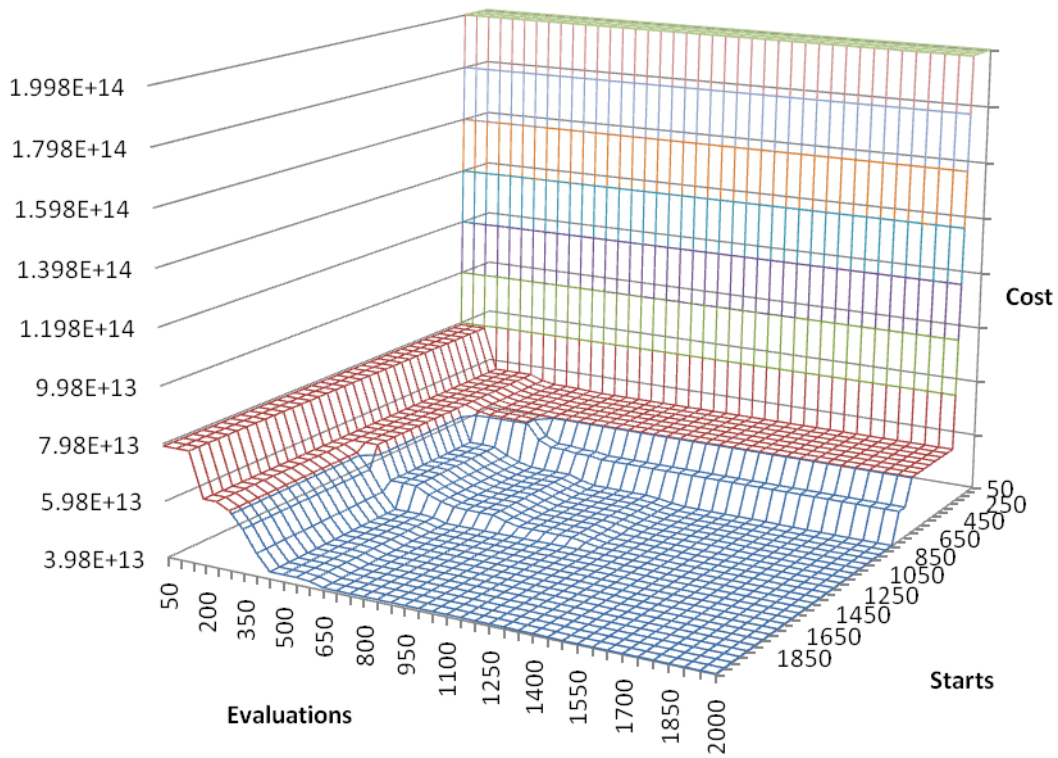

Figure 47. Experiment B – Multi-Directional for a range of starts and evaluations

Examining the "floor" of the results shows that as the experiments reached the maximum values (2000 starts and 2000 evaluations), small improvements were still being observed.

This suggests that it would be useful to examine larger values for both starts and evaluations.

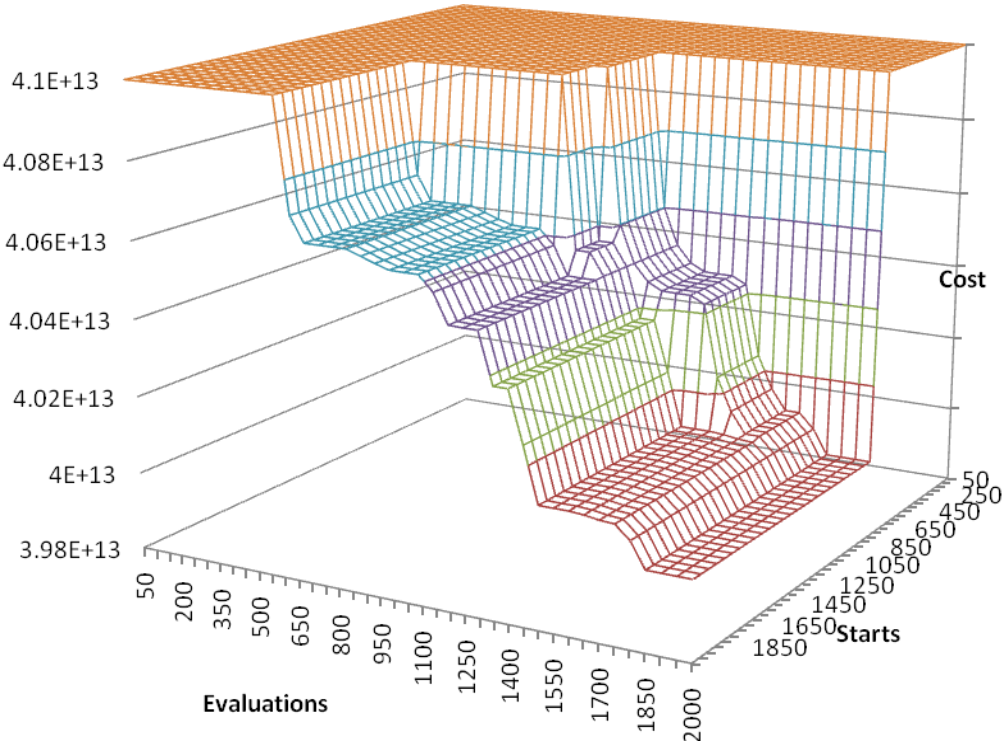

Figure 48. Experiment B – Multi-Directional range test (zoomed)

8.2.2.1 **Best result**

The loading port tank levels were very stable – an increase of 0.450228 m^3 of LNG per hour on average.

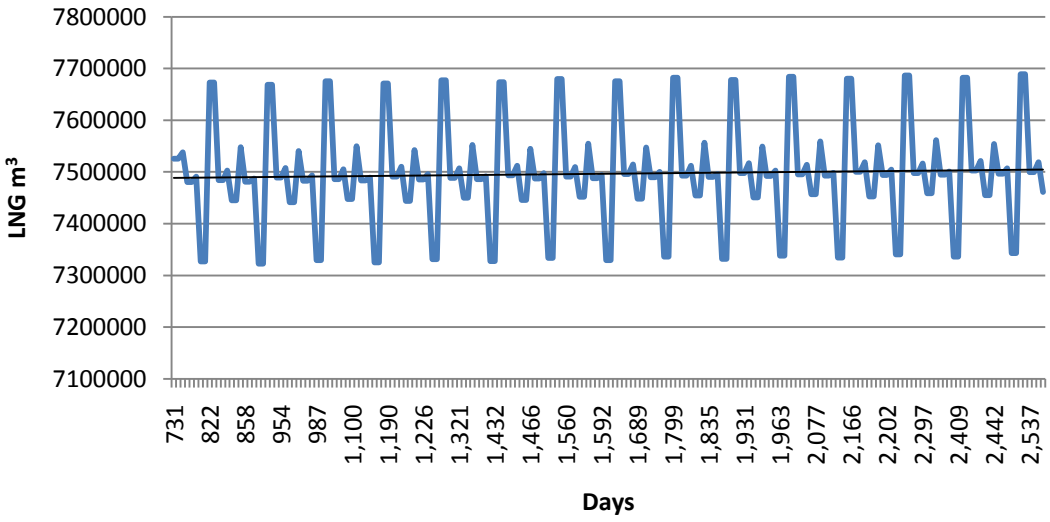

Figure 49. Experiment B – loading port tank levels

The two receiving ports were decreasing at -0.326085 and -0.347883 m^3 of LNG per hour.

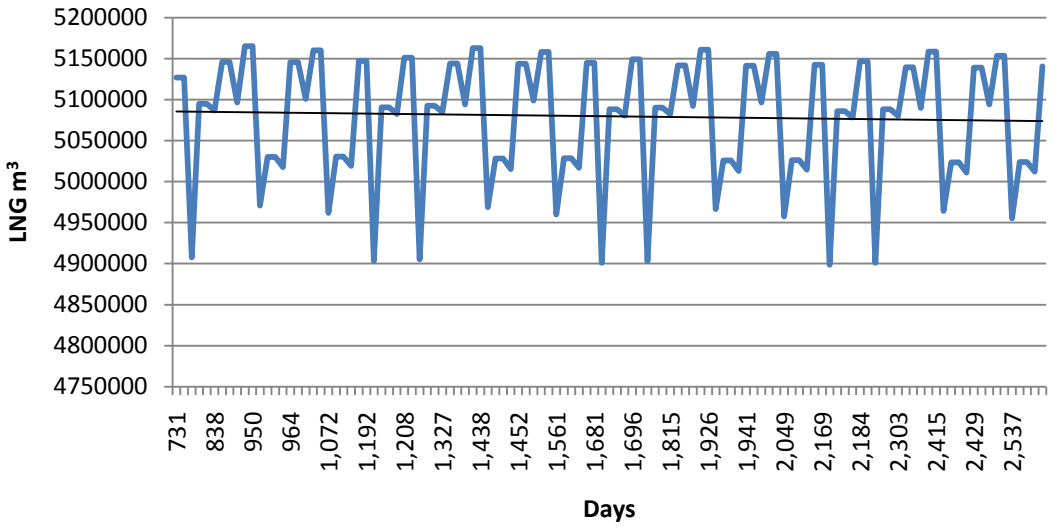

Figure 50. Experiment B – receiving port 1 levels

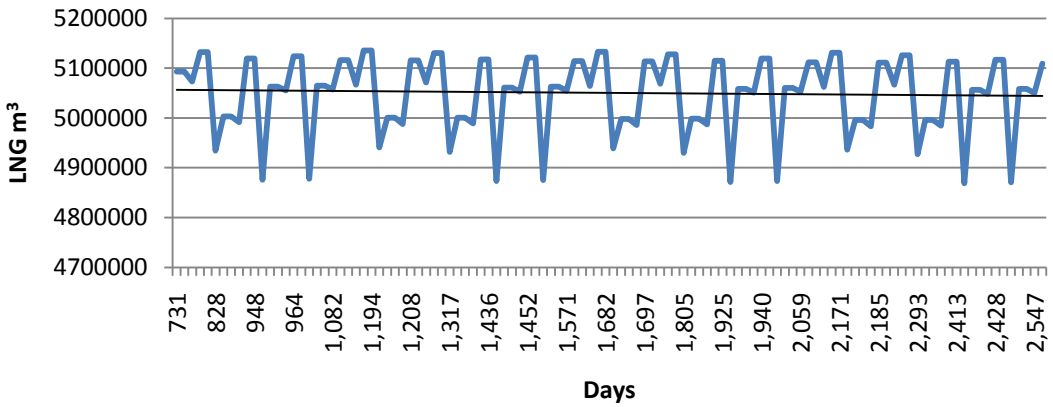

Figure 51. Experiment B - receiving port 2 levels

The delivery schedule is very regular and demand was completely satisfied. The solution picked uses two large ships and two smaller ones – a very different approach to the solution presented by the Nelder Mead algorithm.

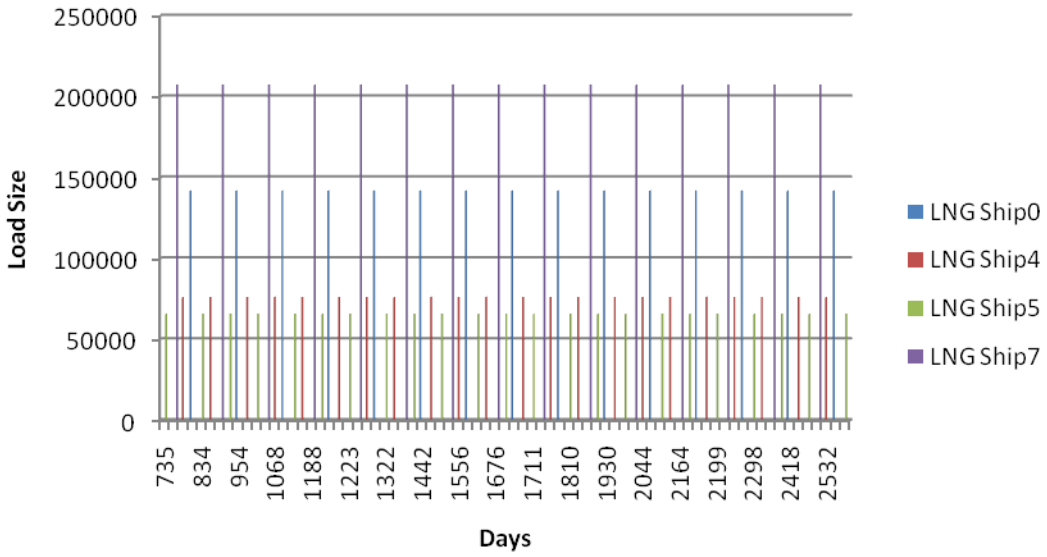

Figure 52. Experiment B – Multi-Directional delivery schedule

8.3 Experiment C

8.3.1 Nelder Mead Results

A test series was run varying the number of starts and maximum number of evaluations between 50 and 2000

The best model value found used 3 ships, with capacities of 360535, 501775 and 359501 m^3 of LNG. The best result was found with 1400 starts and 300 evaluations (approximately since the experiment was undertaken with intervals of 50 for both starts and evaluations).

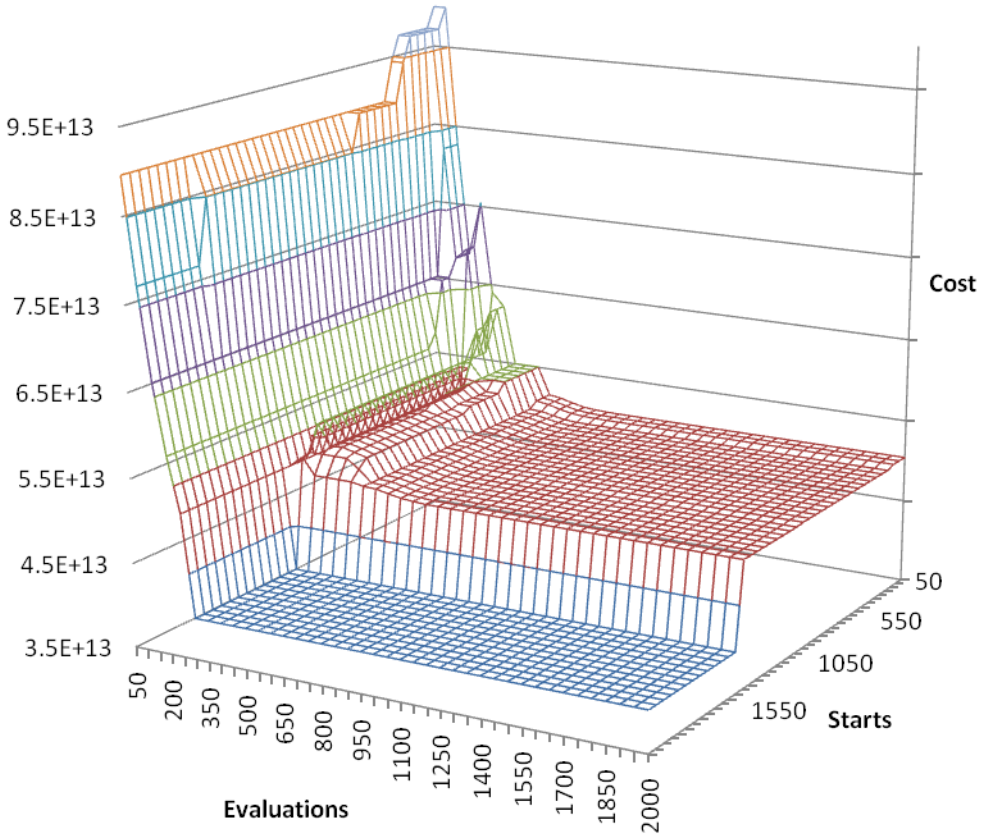

Figure 53. Experiment C – Nelder Mead results for a range of values

8.3.1.1 **Best result**

The loading port was stable – a downward trend of 7.683620 m^3 of LNG per hour

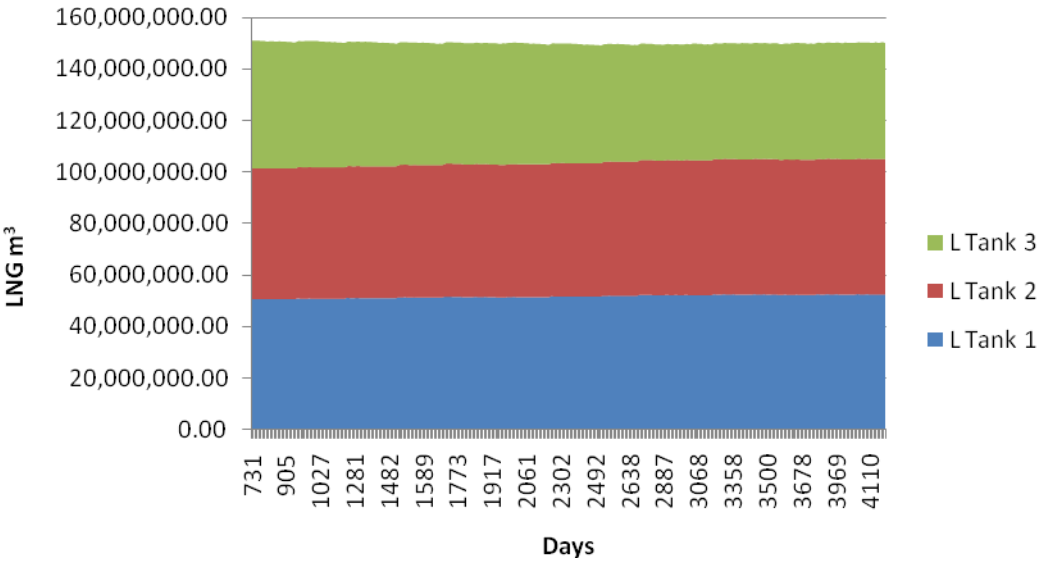

Figure 54. Experiment C – Nelder Mead - loading port levels per tank

The receiving ports were also stable with an upward trend of 4.086805 and 3.807143 m^3 of LNG per hour, respectively.

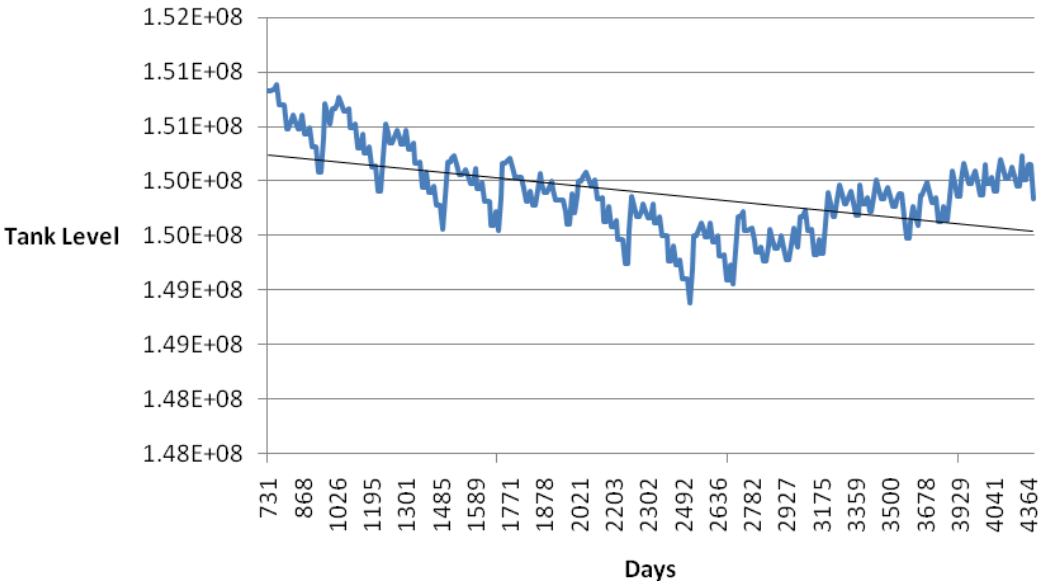

Figure 55. Experiment C – Nelder Mead - total tank levels at the loading port

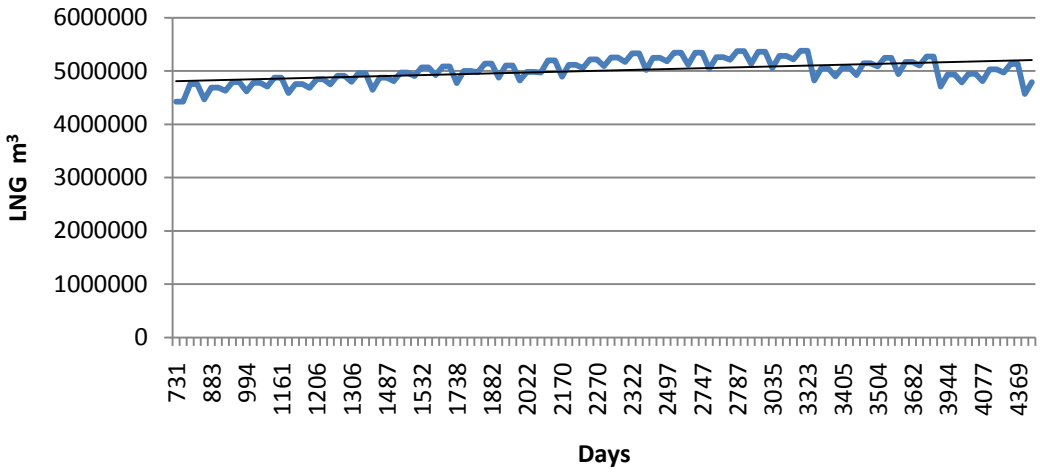

Figure 56. Experiment C – Nelder Mead - receiving port 1 tank levels

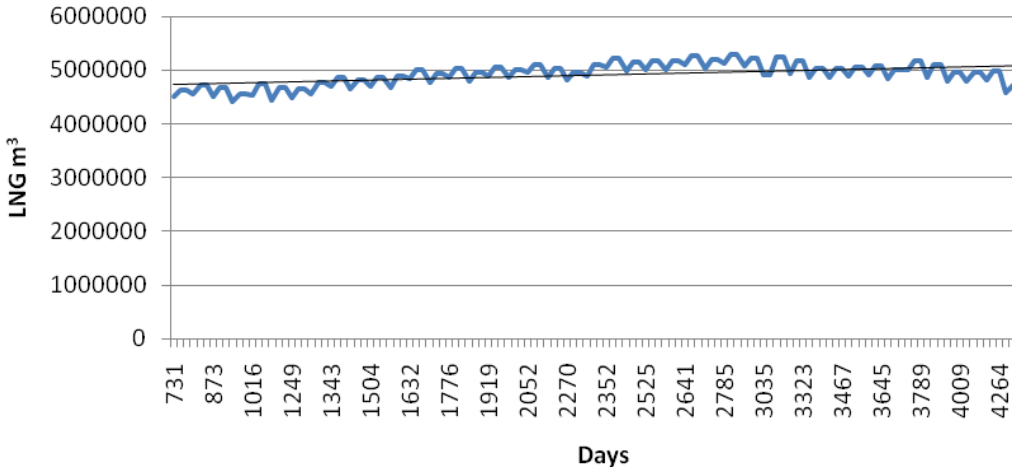

Figure 57. Experiment C – Nelder Mead - receiving port 2 tank levels

The delivery schedule was regular – one large ship and two somewhat smaller ones. Demand was completely satisfied.

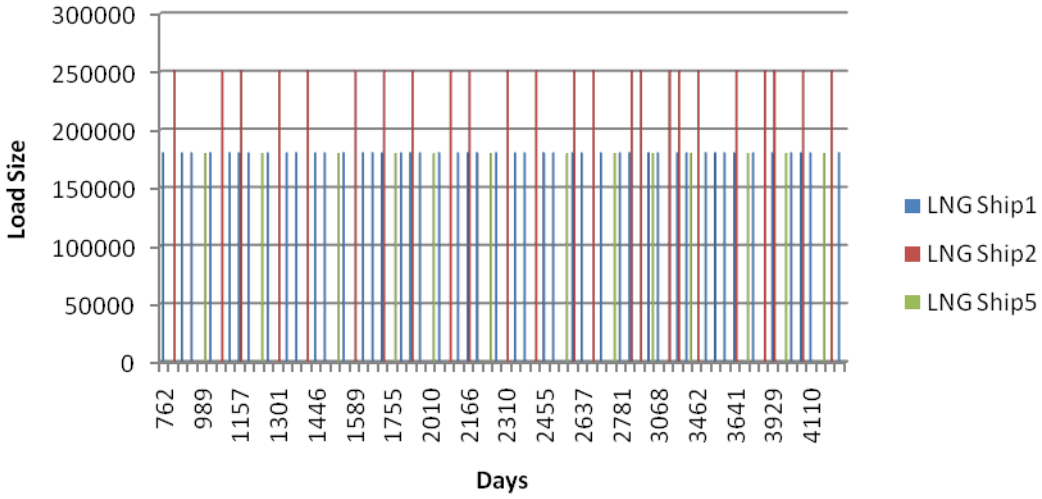

Figure 58. Experiment C – Nelder Mead - delivery schedule

8.3.2 Multi-Directional Results

A test series was run varying the number of starts and maximum number of evaluations between 50 and 2000

The best model value found used 3 ships, with capacities of 354005, 371502 and 453149 m^3 of LNG. This is a considerably better result than the Nelder Mead experiment (above). The best result was found at 1050 starts and 250 evaluations (approximately since the experiment was undertaken with intervals of 50 for both starts and evaluations).

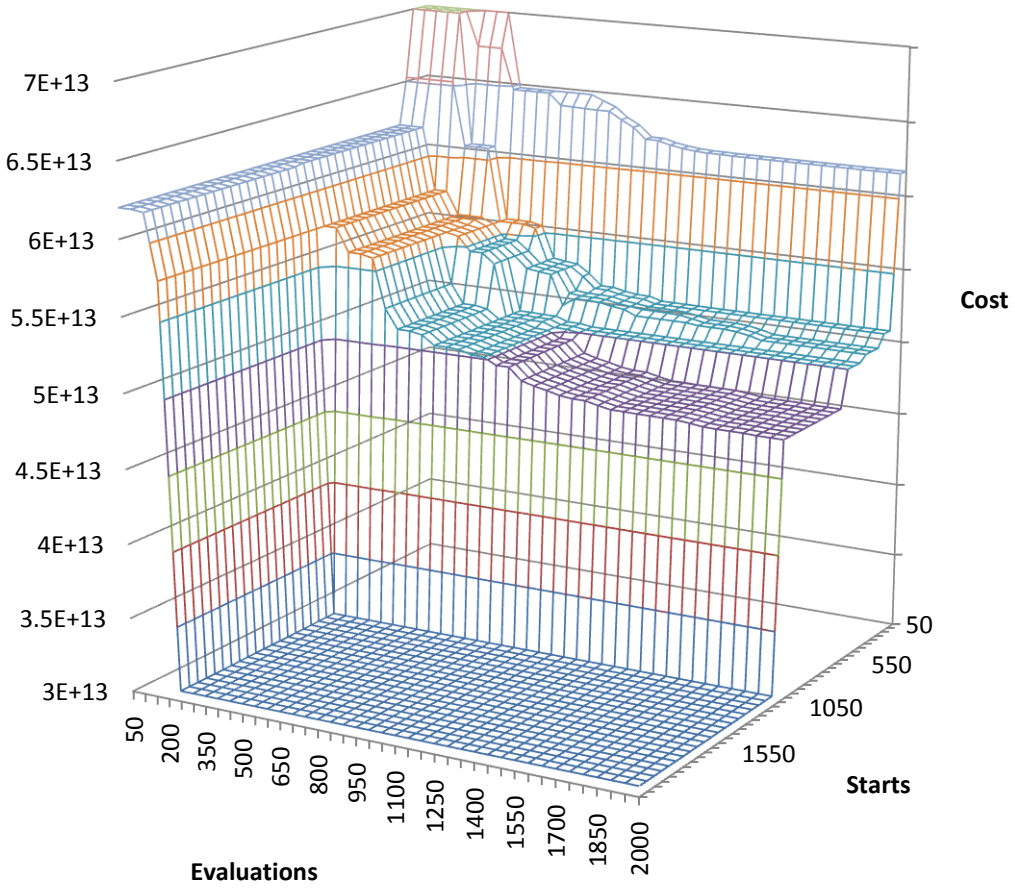

Figure 59. Experiment C – Multi-Directional - results for a range of values

8.3.2.1 **Best result**

The loading port was stable – a decrease of 0.782425 m^3 of LNG per hour

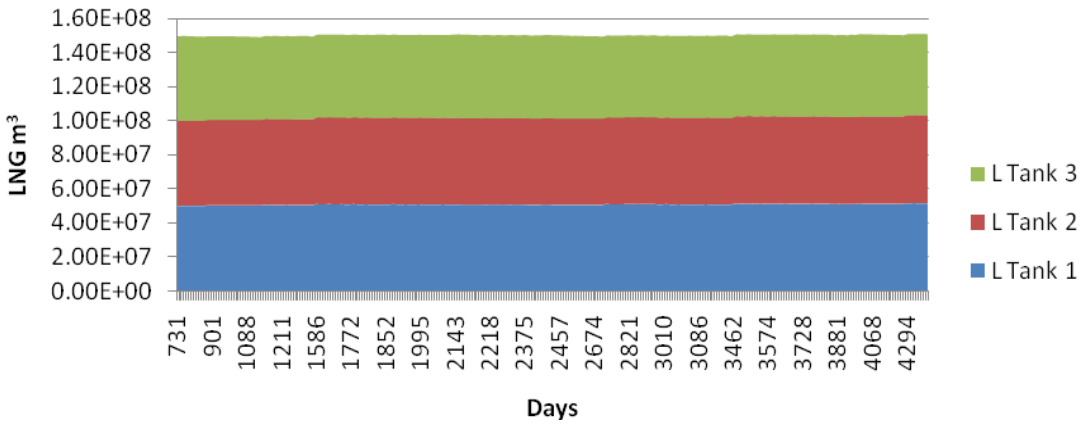

Figure 60. Experiment C – Multi-Directional - loading port tank levels

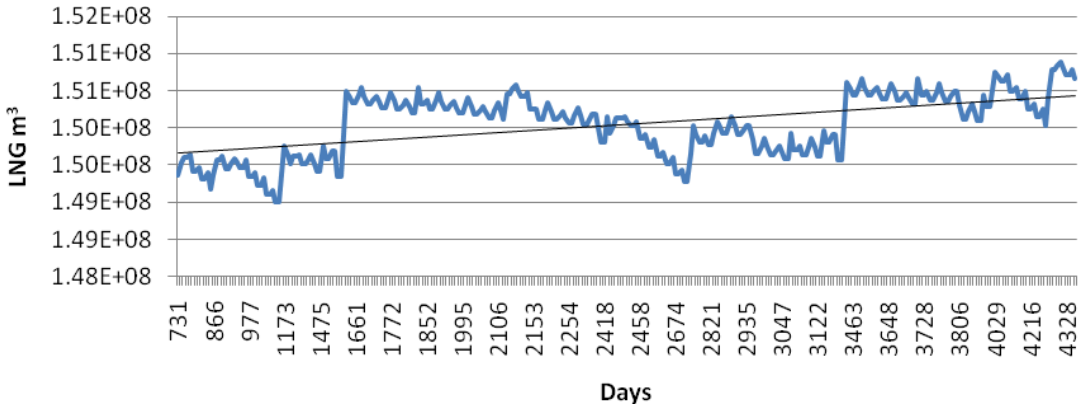

Figure 61. Experiment C – Multi-Directional - overall loading port tank levels

The receiving ports were also stable – increase of 0.120831 and 0.845390 m^3 of LNG per hour.

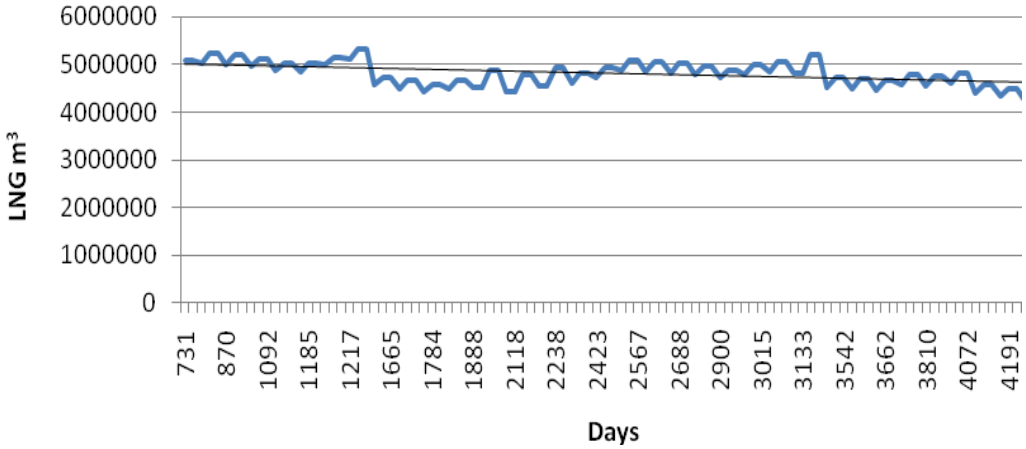

Figure 62. Experiment C – Multi-Directional - receiving port 1 tank levels

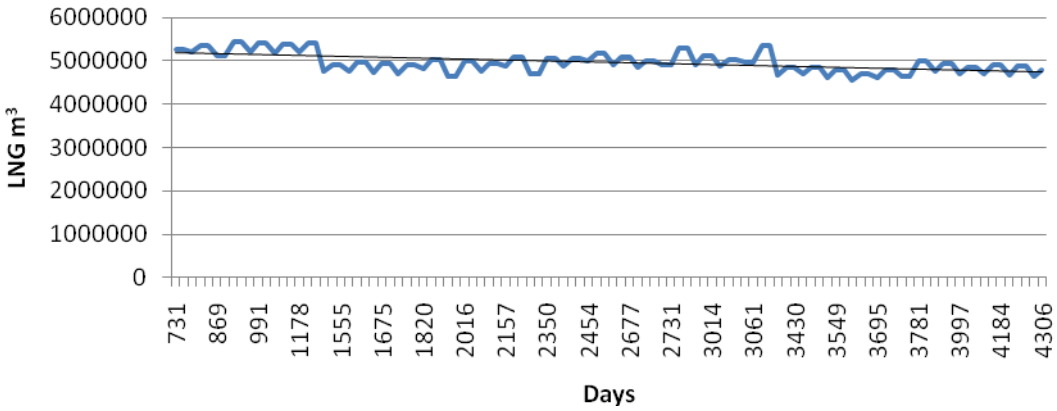

Figure 63. Experiment C – Multi-Directional - receiving port 2 tank levels

Demand was fully satisfied. The delivery schedule is regular – with 3 ships of similar size doing the deliveries.

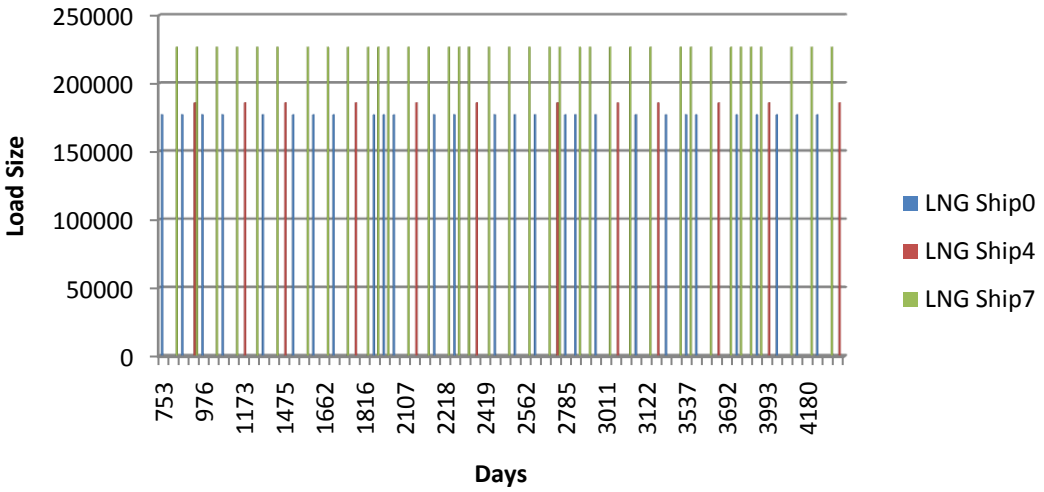

Figure 64. Experiment C – Multi-Directional - delivery schedule

8.4 Experiment D

8.4.1 Nelder Mead Results

A test series was run varying the number of starts and maximum number of evaluations between 50 and 2000

The best model value found used 5 ships, with capacities of 490316, 82450, 185310, 452968 and 526843m^3 of LNG. The best result was found at 1750 starts and 750 evaluations (approximately since the experiment was undertaken with intervals of 50 for both starts and evaluations).

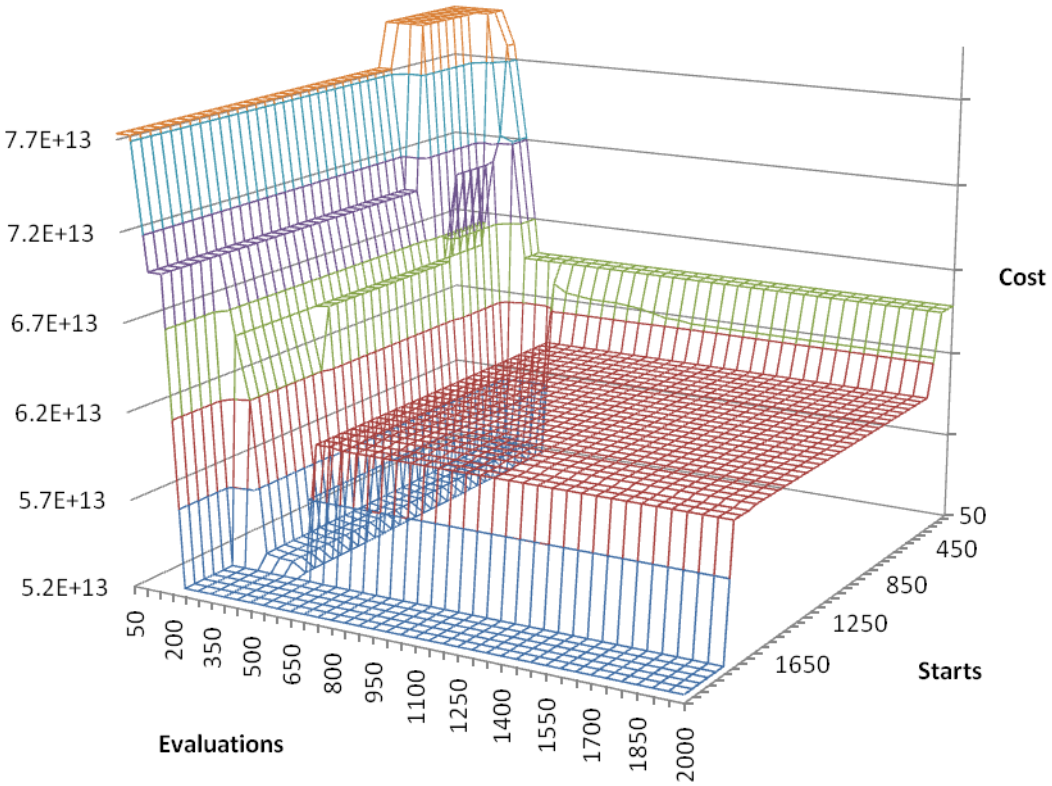

Figure 65. Experiment D – Nelder Mead - results over a range of values

The discontinuity in the effect of increasing the number of evaluations (at around 550 evaluations) is of interest and worthy of further study. It does not, however, effect reaching the lowest values, beyond 1750 starts.

8.4.1.1 **Best result**

The loading port tank farm was stable – an average gain of 1.160268 m^3 of LNG per hour.

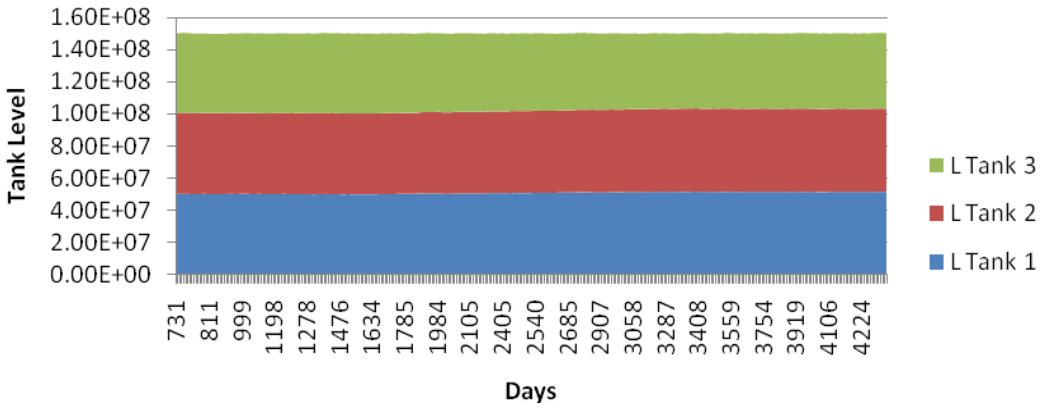

Figure 66. Experiment D – Nelder Mead - loading port tanks

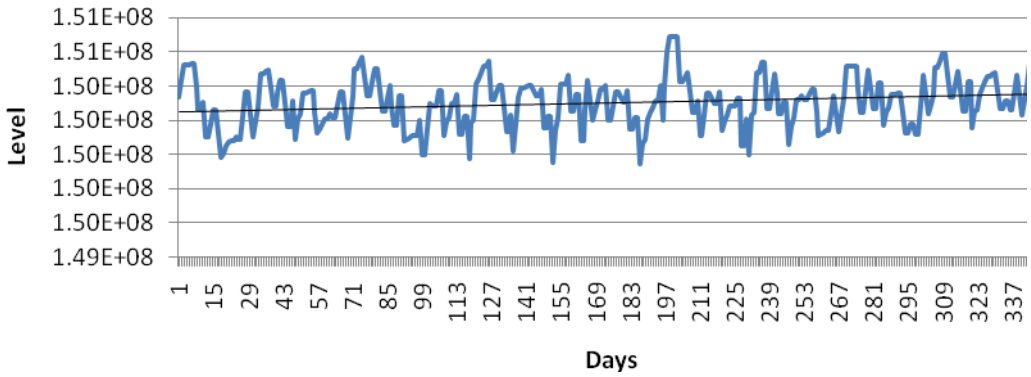

Figure 67. Experiment D – Nelder Mead - Loading port tanks overview

The receiving ports were stable as well – with falls of -0.078641 and 1.168479 m^3 of LNG per hour.

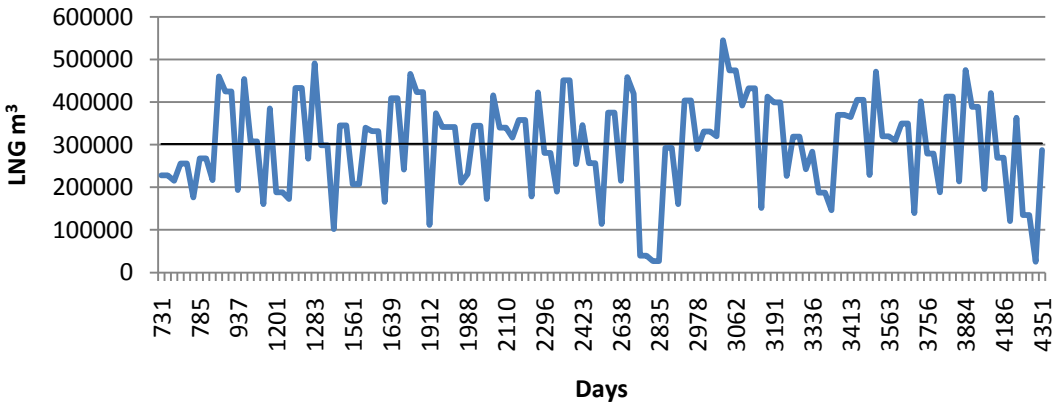

Figure 68. Experiment D – Nelder Mead - receiving port 1

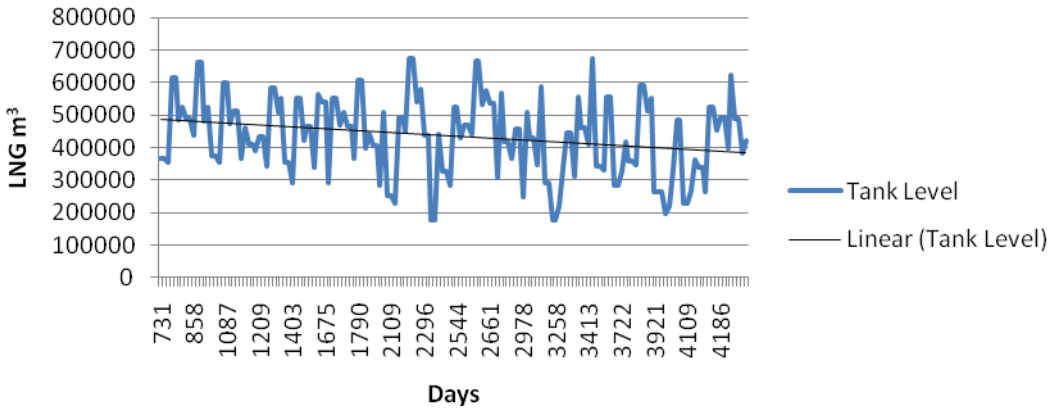

Figure 69. Experiment D – Nelder Mead - receiving port 2

Examining the data for receiving port 2 suggests that the use of a more complex trend line method might be useful – the following shows the result of fitting a 3rd order polynomial.

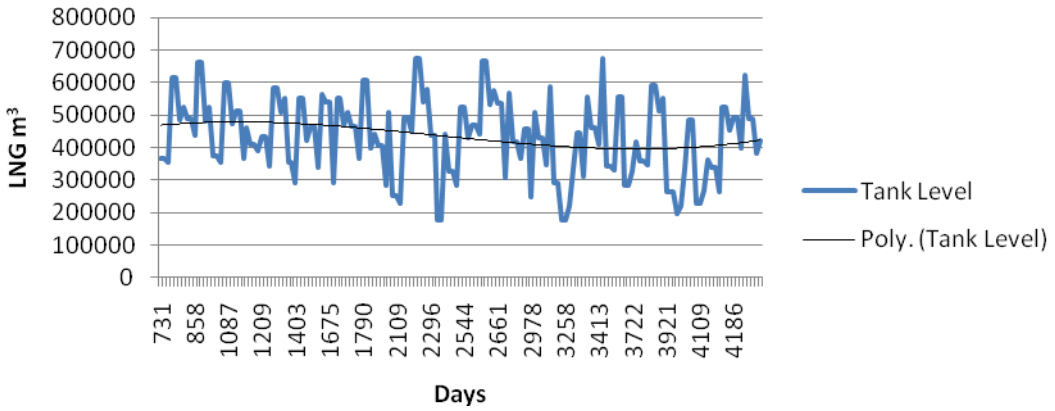

Figure 70. Experiment D – Nelder Mead – receiving port 2, polynomial trendline

The delivery schedule was regular and demand was completely satisfied. The solution picked has 3 large ships and 2 much smaller ones to smooth the flow as is evident in the following figure.

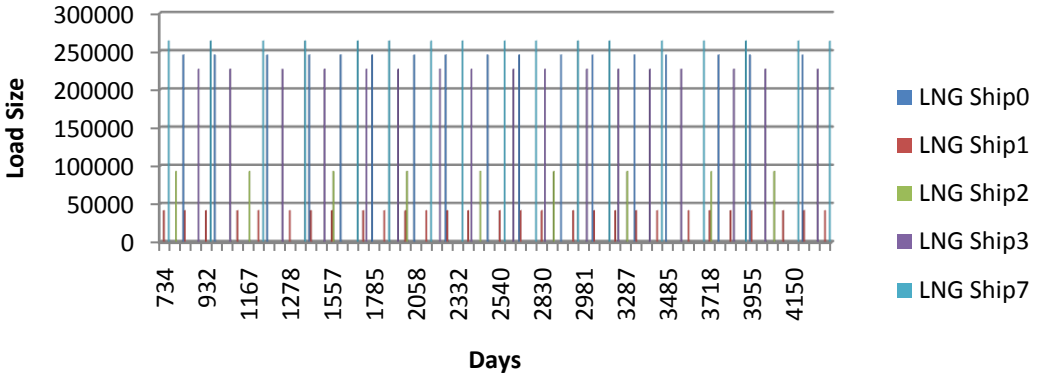

Figure 71. Experiment D – Nelder Mead - delivery schedule

8.4.2 Multi-Directional Results

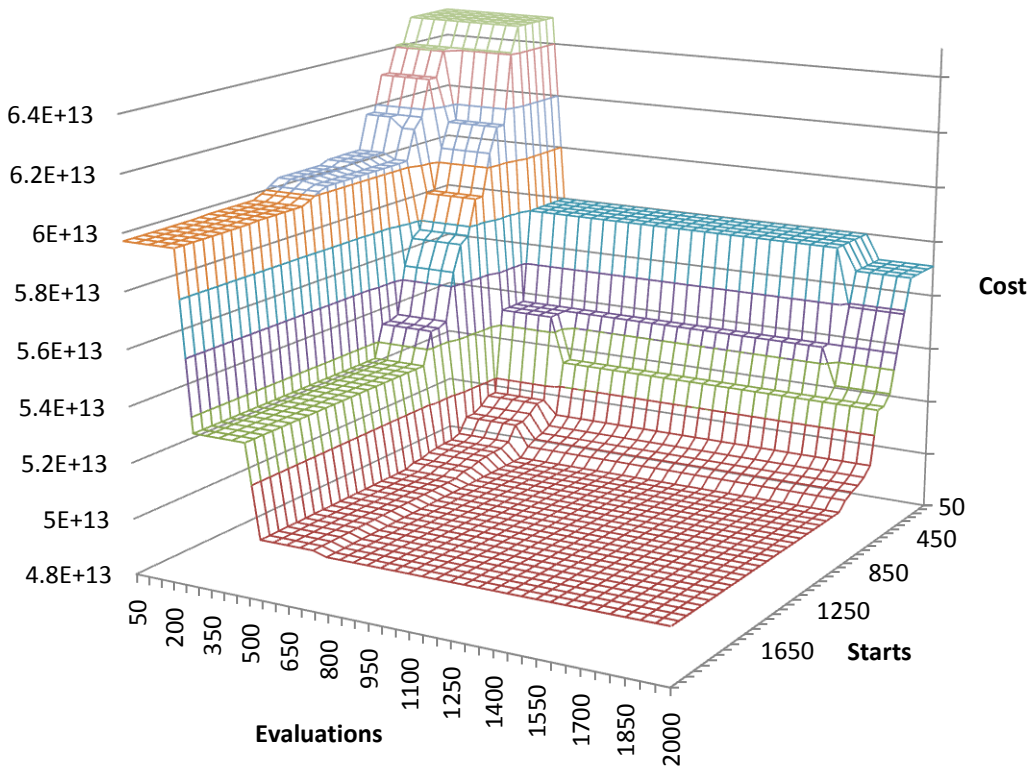

Figure 72. Experiment D – Multi-Directional - results for a range of values

A test series was run varying the number of starts and maximum number of evaluations between 50 and 2000

The best model value found used 5 ships, with capacities of 196459, 102193, 324112, 458100 and 343157m^3 of LNG. The best result was found at 800 starts and 2000 evaluations (approximately since the experiment was undertaken with intervals of 50 for both starts and evaluations).

8.4.2.1 **Best result**

The loading port was stable – with an overall increase of 0.122971 m3 of LNG per hour on average.

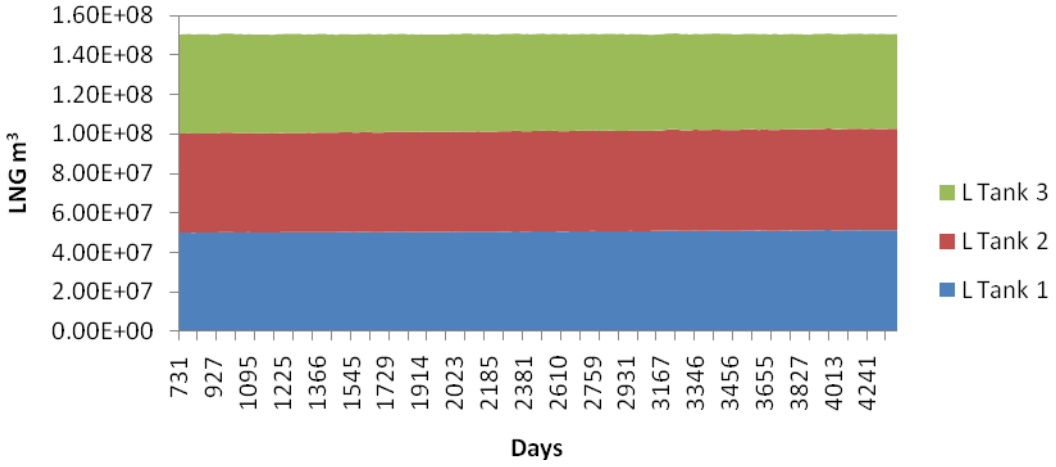

Figure 73. Experiment D – Multi-Directional - loading port tank levels

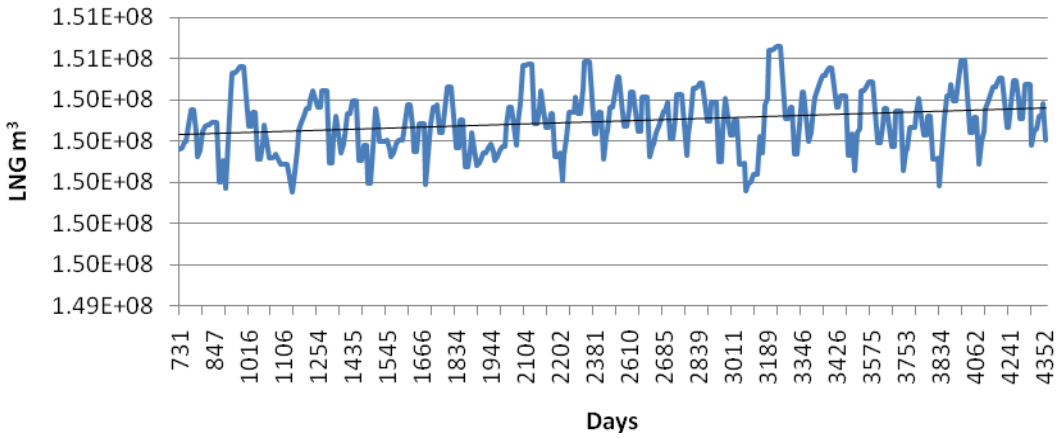

Figure 74. Experiment D – Multi-Directional - overall loading port tank levels

The receiving ports were also stable – reducing by 0.077138 and increasing by 0.000232 for Ports 1 and 2 respectively.

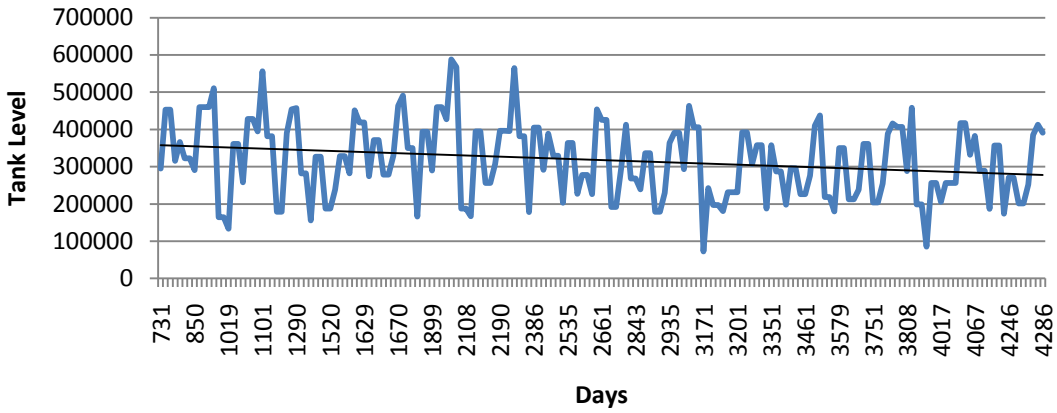

Figure 75. Experiment D – Multi-Directional - receiving port 1 tank levels

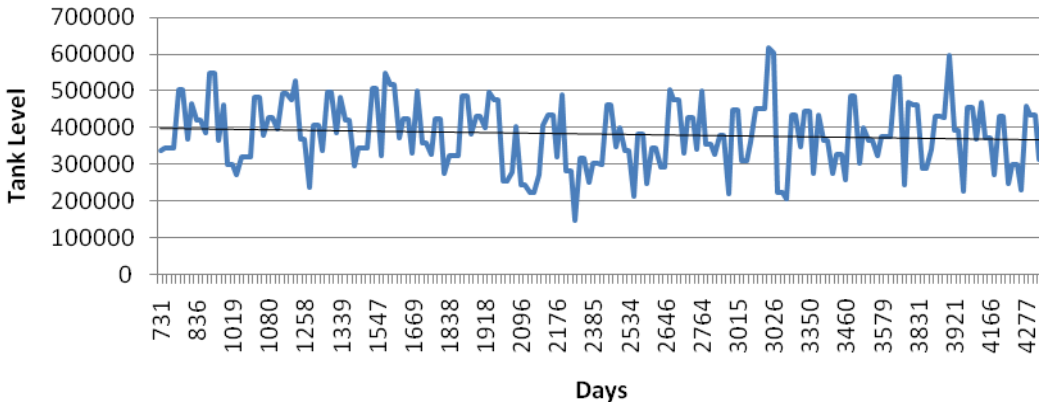

Figure 76. Experiment D – Multi-Directional - receiving port 2 tank levels

The delivery schedule was regular and demand was completely satisfied. The solution picked – 5 ships of a range of sizes is evident in the following figure.

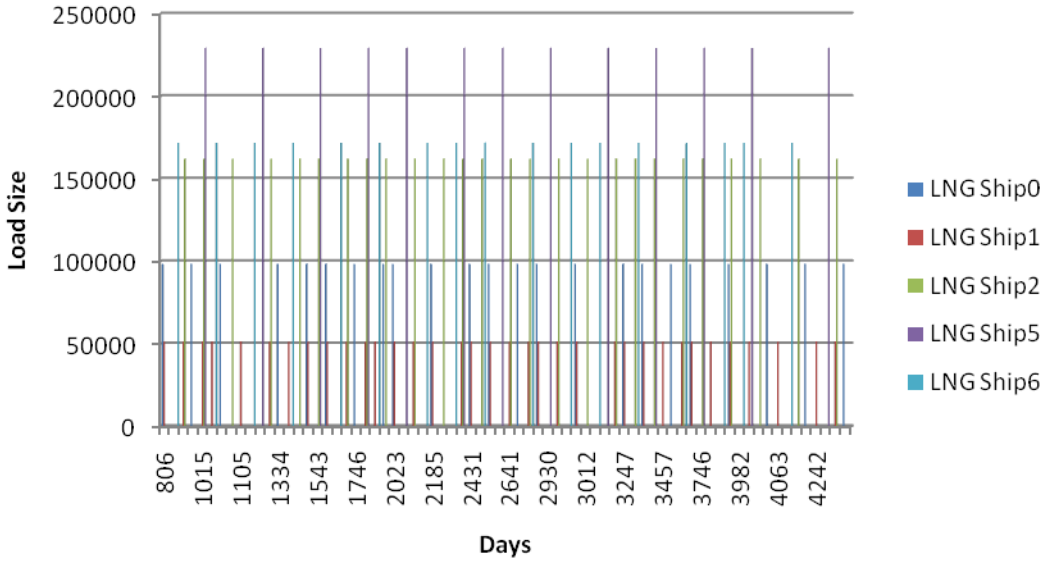

Figure 77. Experiment D – Multi-Directional - delivery schedule

8.5 Experiment E

The model was run in generate mode and then 10,000 times in test mode. All the tests passed. The largest amount of contingency used was 2 days, 3 hours and 46 minutes.

8.6 Results Summary

Table 21. Comparison of results

Exp.	A	A	B	B	C	C	D	D
Type	**MD**	**NM**	**MD**	**NM**	**MD**	**NM**	**MD**	**NM**
Starts	150	1250	750	1150	1050	1400	800	1750
Evals	1950	300	2000	700	250	300	2000	750
Total Ship Capacity	421302	991484	697594	787316	1178658	1221812	1227564	1247573
Ships	2	4	4	5	3	3	5	5
Loading Port tank trend	0.1461	0.1548	0.4502	-0.8562	-0.7824	-7.6836	0.1230	1.1603
Receiving port R1 tank trend	0.0000	0.0000	-0.3261	0.2684	0.1208	4.0868	-0.0771	-0.0786
Receiving port R2 tank trend	N/A	N/A	-0.3479	0.9192	0.8454	3.8071	0.0002	-1.1685
The model cost	2.00E+13	4.00E+13	4.00E+13	5.00E+13	3.00E+13	3.00E+13	5.00E+13	5.00E+13
Total Cost	2.00E+13	4.00E+13	4.00E+13	5.66E+13	3.07E+13	3.99E+13	5.00E+13	5.25E+13

8.6.1 Generating delivery schedules

The table above gives a summary of the relative results of the various experiments and methods. In general, it seems that the Nelder-Mead method requires more starts and fewer evaluations to reach the optimum point and that the Multi-Directional method has the reverse requirement. In each case,

the Multi-Directional method produced better results. This is particularly clear for experiment C where Nelder-Mead produced a result with trends for the tank levels considerably greater than the 0.5 m^3 of LNG per hour limit and Multi-Directional did considerably better.

8.6.2 Testing schedules

Experiment E successfully demonstrated the concept and implementation of the B event portion of the executive. The aim of this test was to prove this and show that performance in this mode was not excessively affected by the extra events. The performance achieved is discussed in the following section.

8.6.3 Performance

All tests were carried out on a computer with an Intel 6400 2 Core CPU, clocked at 2.13 GHz, with 3 GB of memory. The JVM was allocated 1 GB of memory to ensure that the allocation was not exceeded.

8.6.3.1 **Generate mode**

A test series was run for Experiments C and D for 2000 starts and 2000 evaluations, to establish the performance of the modelling environment. The number of configurations is interesting – it suggests that the difference in performance noted above for the Multi-Directional and Nelder Mead methods is due to the larger number of configurations evaluated by the MD method.

Table 22. Time taken to perform experiments

	Multi-Directional	Nelder Mead
C	3 hours 31 mins 30 secs for 2179065 configurations.	1 hour 46 mins 29 secs for 1145832 configurations.
D	1 hour 48 mins 24 secs for 940680 configurations.	44 mins 52 secs for 456139 configurations.

The average time for evaluating each configuration is given below. It should be noted that a large number of the configurations will have aborted due to tanks becoming empty or full, so that many of the configurations will not have run for the full period.

Table 23. Time per configuration evaluation (ms)

	Multi-Directional	Nelder Mead
C	5.809831281	5.550551913
D	6.888633754	5.787709448

The best case result from Multi-Directional in experiment D was run 10000 times in generate mode – multiple runs of this took an average of 8.85 milliseconds. This gives the time taken for a non-aborting run for a model with realistic complexity.

This level of performance is orders of magnitude than earlier models that the author is familiar with – milliseconds instead of seconds.

8.6.3.2 **Test Mode**

The best case result from Multi-Directional in experiment E was run 10,000 times in generate mode. Multiple runs of this took an average of 18.3 milliseconds. This demonstrates that the additional cost of the probabilistic B events is manageable – especially since the number of breakdowns in question is much higher than would occur in a real-world modelling situation.

Chapter 9. Conclusions

The following objectives were formulated for this work –

To select a suitable simulation methodology

Following a detailed examination of the field, a modification of the Three Phased Approach was chosen.

To investigate the flexibility and performance of the modelling method

The approach chosen was implemented and demonstrated using representative problems. It has higher performance than existing industry standard models, for realistic levels of complexity. Simulating 10 years of operation can take a similar number of milliseconds, for problems with a real-world level of complexity. Previously, such work would have taken a number of seconds. This means that millions of configurations can be considered in a few hours (for a basic desktop computer) – making automated design practical for the first time

Select experimentation methods

Given the performance achieved, and the nature of the experiments required, the Nelder-Mead and Multi-Directional methods were selected and have been successfully implemented.

Investigate the performance of the methods and the possibilities for automating design of the supply chain structure.

Using these two techniques (above), a range of experiments was carried out. These validated the concept and indicated the success of the automated design methods proposed.

Both generate and test modes have been implemented and used. This means that the complete concept has been demonstrated. In both modes representative problems can be modelled and tested in a few hours on a single computer. This represents a very useful level of performance.

The capabilities inherent in this new system would be invaluable to planners working on continuous supply chain problems. In the past, performance has placed severe restraints on design of LNG supply chains – work has been done manually and largely by intuition, rather than by a systematic search through the solution space.

The testing mode has also been evaluated. The performance of the model is only moderately affected by the extra events, making the use of this mode equal in practicality, compared to the generate mode.

9.1 Future Steps

9.1.1 Modelling other systems

The methodology has been designed to be generic – any continuous (or partially continuous) system with a similar type of structure should be able to be modelled with it. Oil supply chains, water distribution & treatment and ore handling are a few examples of suitable candidates.

9.1.2 Complete automated design?

If we can build a schedule in a matter of a few hours and test it in minutes, then it makes sense to combine the techniques. A combined system might test (by running in the **"Test"** mode) some of the valid delivery schedules (created in the **"Generate"** phase) as a part of the design process. The

schedules that would be selected for testing would be those that show a significant level of performance - the top 1000, say. For Experiment E, it would be possible to test each one 1,000 times and complete the whole test in less than 5 hours, for example. This assumes a single thread running on the hardware used for these tests. Using parallelisation, discussed below, could reduce this time substantially and allow more sophisticated strategies for testing. Testing each successful delivery schedule as it is generated, within the design cycle, might become possible.

9.1.3 Parallelisation

The direct search techniques used are trivially parallelisable – each start is independent of each other. For testing, a single thread per delivery program being tested logically suggests running in parallel. The obvious method for such parallelisation is to use a grid. Another possibility is using GPU (Graphical Processing Unit) computing. Recently, vendors such as NVIDIA have begun emphasising the capabilities of their technology for solving more generalized problems than merely graphical display. A C1060 (http://www.nvidia.com/object/product_tesla_c1060_us.html) card, for example, contains 240 processors. A standard desktop high performance computing setup consists of 4 such cards in high end PC. This gives a total of 960 processors running at 1.3 GHz.

A major limiting factor with such equipment is moving data on and off the memory on the GPU cards (Albanese, 2008). Since the model itself is not large, and the data it produces (the best results) are small in comparison to the data requirements of the financial problems that Albanese describes, this should a solvable issue.

If it is possible to adapt the model described in this work to work in a GPU computing environment problems could be run in a fraction of the time described in the section on performance. The challenge would be to code the model and its data to fit within the constraints of the GPU hardware and software environment.

Bibliography

Albanese, C. “ GPU Computing for Financial Engineering”, seminar at King's College London, 2008

Andersson, M. & Olsson, G. "A Simulation Based Decision Support Approach for Operational Capacity Planning in a Customer Order Driven Assembly Line", Proceedings of The Winter Simulation Conference, 1998

Apache Commons Maths Library, http://commons.apache.org/math/

Banks, J; Carson, J.S; Nelson, B.L. & Nicol, D.M. "Discrete-Event System Simulation, Third Edition", Prentice Hall, 2000

Beck, U & Nowak, J. W. “The Merger of Discrete Event Simulation with Activity Based Costing for Cost Estimation in Manufacturing Environments”, Proceedings of The Winter Simulation Conference, 2000

Bruzzone, A. G; Giribone, P. & Revetria, R. "Operative Requirements and Advances for the New Generation Simulators in Multimodal Container Terminals", Proceedings of The Winter Simulation Conference, 1999

Burgsteden, M. C; Joustra P. E; Bouwman M. R. & Hullegie, M. “Modeling Road Traffic on Airport Premises”, Proceedings of The Winter Simulation Conference, 2000

Daum, T & Sargent R. G. "Scaling, Hierarchical Modeling, and Reuse in an Object-Oriented Modeling and Simulation System", Proceedings of The Winter Simulation Conference, 1999

Finn, A.J, Johnson G.L & Tomlinson, T.R. “LNG Technology For Offshore And Midscale Plants”, 79th Annual GPA Convention, Atlanta, 2000

Fliege, J. - Prof. Dr. Jörg Fliege, FORS, Director of CORMSIS, University of Southampton. Personal communication to author.

Golkar, J; Shekar, A. & Buddhavarapu, S. "Panama Canal Simulation Model", Proceedings of The Winter Simulation Conference, 1998

Huang, Y & Iyer, R. K. "An Object-Oriented Environment for Fast Simulation Using Compiler Techniques", Proceedings of The Winter Simulation Conference, 1998

Ioannou, P. G. "Construction of a Dam Embankment with Nonstationary Queues", Proceedings of The Winter Simulation Conference, 1999

Joines, J. A. & Roberts, S. D. "Simulation in an Object-Oriented World", Proceedings of The Winter Simulation Conference, 1999

Kilgore, R. A. & Burke, E. "Object-Oriented Simulation of Distributed Systems Using Java® and Silk®", Proceedings of The Winter Simulation Conference, 2000

Kiran, A. S; Cetinkaya, T & Og, S. "Simulation Modeling and Analysis of New International Terminal", Proceedings of The Winter Simulation Conference, 2000

Kyle, R. G. & Ludka, C. R. "Simulating the Furniture Industry", Proceedings of The Winter Simulation Conference, 2000

Law, A.M. & Kelton, W.D. "Simulation Modeling and Analysis", Third edition, McGraw-Hill, 2000

Lewis J.P. & Neumann U. "Performance of Java versus C++", University of Southern California, 2004, http://www.idiom.com/~zilla/Computer/javaCbenchmark.html

Marzouk, M. & Moselhi, O. "Optimizing Earthmoving Operations Using Object-Oriented Simulation", Proceedings of The Winter Simulation Conference, 2000

McKinnon, K.I.M. "Convergence of the Nelder-Mead simplex method to a non-stationary point", SIAM J Optimization, 1999, vol. 9, pp148-158.

Mitriani, I. "Simulation Techniques for Discrete Event Systems", Cambridge University Press, 1982

Nelder, J.A. & Mead, R. "A simplex method for function minimization", Computer Journal, 1965, vol. 7, pp 308-313

Perumalla, K & Fujimoto, R. "Efficient Large-Scale Process-Oriented Parallel Simulations", Proceedings of The Winter Simulation Conference, 1998

Pidd, M & Castro, R. B. "Hierarchical Modular Modeling In Discrete Simulation", Proceedings of The Winter Simulation Conference, 1998

Pidd, M. "Computer Simulation in Management Science", Wiley, 1998

Russell, E. C. "SIMSCRIPT II.5 Programming Language", CACI, La Jolla, C. A. 1987

Schriber, T. J. & Brunner, D. T. "Inside Discrete-Event Simulation Software: How It Works and Why It Matters", Proceedings of The Winter Simulation Conference, 2000, 1999, 1998, 1997

Schunk D. & Plott B. "Using Simulation to Analyze Supply Chains", Proceedings of The Winter Simulation Conference, 2000

Shi, J. J. "Object-Oriented Technology for Enhancing Activity-Based Modeling Functionality", Proceedings of The Winter Simulation Conference, 2000

Stchedroff N. & Cheng, R.C.H. "Modelling A Continuous Process With Discrete Simulation Techniques And Its Application To LNG Supply Chains", Proceedings of The Winter Simulation Conference, 2003

Swedish, J. A. "Simulation of an Inland Waterway Barge Fleet Distribution Network", Proceedings of The Winter Simulation Conference, 1998

Takakuwa, S. "A Practical Module-Based Simulation Model for Transportation Inventory Systems", Proceedings of The Winter Simulation Conference, 1998

Takakuwa, S; Takizawa, H; Ito K. & Hiraoka, S. "Simulation and Analysis of Non-Automated Distribution Warehouses", Proceedings of The Winter Simulation Conference, 2000

Torczon, V. "On The Convergence Of The Multidirectional Search Algorithm", 1991

Trone, J; Guerin, A. & Clay, A. D. "Simulation of Waste Processing, Transportation, and Disposal Operations", Proceedings of The Winter Simulation Conference, 2000

Wright, Margaret H. "Direct search methods: Once scorned, now respectable", 1996