DESIGN OF AN EXPANDABLE MANUFACTURING SIMULATOR THROUGH THE APPLICATION OF OBJECT-ORIENTED PRINCIPLES

PETER DAVID BALL

Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

July 1994

The University of Aston in Birmingham

Design of an expandable manufacturing simulator through the application of object-oriented principles

Peter David Ball **Doctor of Philosophy 1994**

Summary

In analysing manufacturing systems, for either design or operational reasons, failure to account for the potentially significant dynamics could produce invalid results. There are many analysis techniques that can be used, however, simulation is unique in its ability to assess detailed, dynamic behaviour. The use of simulation to analyse manufacturing systems would therefore seem appropriate if not essential.

Many simulation software products are available but their ease of use and scope of application vary greatly. This is illustrated at one extreme by simulators which offer rapid but limited application whilst at the other simulation languages which are extremely flexible but tedious to code. Given that a typical manufacturing engineer does not possess in depth programming and simulation skills then the use of simulators over simulation languages would seem a more appropriate choice.

Whilst simulators offer ease of use their limited functionality may preclude their use in many applications. The construction of current simulators makes it difficult to amend or extend the functionality of the system to meet new challenges. Some simulators could even become obsolete as users demand modelling functionality that reflects the latest manufacturing system design and operation concepts.

This thesis examines the deficiencies in current simulation tools and considers whether they can be overcome by the application of object-oriented principles. Object-oriented techniques have gained in popularity in recent years and are seen as having the potential to overcome many of the problems traditionally associated with software construction. There are a number of key concepts that are exploited in the work described in this thesis: the use of object-oriented techniques to act as a framework for abstracting engineering concepts into a simulation tool and the ability to reuse and extend object-oriented software.

It is argued that current object-oriented simulation tools are deficient and that in designing such tools, object-oriented techniques should be used not just for the creation of individual simulation objects but for the creation of the complete software. This results in the ability to construct an easy to use simulator that is not limited by its initial functionality. The thesis presents the design of an object-oriented data driven simulator which can be freely extended. Discussion and work is focused on discrete parts manufacture.

The system developed retains the ease of use typical of data driven simulators, whilst removing any limitation on its potential range of applications. Reference is given to additions made to the simulator by other developers not involved in the original software development. Particular emphasis is put on the requirements of the manufacturing engineer and the need for the engineer to carry out dynamic evaluations.

Keywords Manufacturing simulation, Expandable simulator, Object-oriented, Manufacturing system design, Modelling.

Acknowledgements

I have many people to thank, without whom none of this would have been possible

Firstly I would like to thank my partner, Hilary Green, for her patience and support during the research and subsequent preparation of this thesis.

There are many members of the research group at Aston, both past and present, who deserve credit. In particular I must thank my supervisor, Doug Love, for setting up the research project and for subsequent support for this research and other activities at Aston. I should also thank Nick Boughton for his helpful comments prior to the completion of this thesis.

Finally I would like to acknowledge my sponsors, Lucas Engineering & Systems Ltd, for their help, support and suggestions that gave this research an important industrial dimension.

Table of Contents

Appendices

List of Figures

1. Introduction

This chapter details the context in which this thesis is based. Some of the background concepts will be introduced. The format of the subsequent chapters will also be outlined.

1.1. The background

The work addressed in this thesis was carried out to satisfy the long term requirements of the research group at Aston University and the short term requirements of Lucas Engineering & Systems Ltd. (LE&S). Simplistically, the Integrated Design and Manufacture research group at Aston University had a requirement for a software architecture to support their Whole Business Simulation (WBS) (Love et al., 1992) project whilst LE&S had a requirement for an easy to use software tool for use in their manufacturing system design work.

Typically manufacturing systems designs are assessed in isolation with little consideration given to the effects on other parts of the business. Such assessments neglect some important dynamic influences; other parts of a business are equally complex and dynamic in nature. The Integrated Design and Manufacture Research Group at Aston University is investigating the concept of Whole Business Simulation (WBS) (Love et al., 1992). This concept recognises that the manufacturing system is not the only part of a business that could benefit from the application of simulation. The concept of WBS would allow assessment of changes to the manufacturing system or other parts of the business in terms of the business as a whole. Assessment would include the cash flow and profit and loss as well as lead times, stock levels and utilisations.

The WBS software would be large and complex. To be able to apply such a tool to a wide range of businesses it would need the flexibility for modification and tailoring. This leads to the first objective of the work covered in this thesis: to provide a number of key mechanisms by which a large simulation software tool could be constructed and managed.

LE&S assist in the process of manufacturing system design (MSD) and redesign (MSR) for companies inside and outside of the Lucas Group. The design process is that

proposed by Parnaby (1986) and uses a top down approach to reorganising manufacturing systems. One of the stages of the MSD/MSR involves assessment and provision for the dynamics. The process promotes the use of methodology before technology hence involves designing an effective manufacturing system before the introduction of new technology. Typically the MSD/MSR is carried out over a period of 6 months and does not involve highly automated manufacturing systems. The use of any software tool to assist in the process must therefore meet the following criteria:

- quick to apply;
- easy to apply;
- able to assess manufacturing system dynamics;
- recognises the human (as well as technological) element of their designs;
- recognises the diversity of manufacturing systems;
- recognises that additional modelling capabilities may be required in the future.

The second objective of the work covered in this thesis was therefore to provide a software tool that had the potential for use in the manufacturing system design process.

1.2. The thesis

This thesis examines the provision of software to assist manufacturing engineers in the analysis and design of manufacturing systems. The complexity and dynamics of typical manufacturing systems is such that it is difficult for individuals to understand them fully. The use of models and computer software has potential to assist the engineer by providing a means by which complex, dynamic behaviour can be represented and evaluated quickly.

Simulation is a means by which dynamic behaviour can be examined (Chaharbaghi, 1991). It is argued in this thesis, however, that current simulation software does not match the needs of engineers; software either has limited application or is difficult to use. It is argued that there needs to be an improvement in the capabilities of simulation software.

Object-oriented software techniques have grown in popularity in recent years. Object-oriented techniques to software construction can provide a means by which software can be modified and reused easily (Booch, 1991). It is proposed that an object-oriented approach could be employed to analyse, design and implement a manufacturing simulator. This would overcome some of the limitations of existing simulation software. A working simulator has been built and used to demonstrate that the proposal can be achieved.

1.3. The chapter format

Chapter 2 examines the characteristics of manufacturing systems and argues that simulation techniques are well suited to detailed, dynamic analysis.

Chapter 3 argues that the ultimate users of simulation tools are likely to be manufacturing engineers. Simulation software should therefore be built for use by engineers and not those with extensive programming and software experience.

Chapters 4, 5 and 6 examine the limitations of current simulation software and propose that object-oriented software construction techniques should be used. The use of object-oriented techniques has potential to provide ease of use as well as range of application.

Chapters 7, 8 and 9 detail the application of these techniques to build what has been termed the Advanced Factory Simulator (AFS). AFS is shown to provide a means by which manufacturing engineers can build simulation models easily whilst software developers can add new functionality. This is achieved using two modes of operation: model building and software development. In the latter mode a software developer can design and implement new functionality using the existing source code and software development environments. This effectively gives rise to a new version of the software. In the model building mode manufacturing engineers will use the data-driven user-interface to build models. The engineer is unable to modify the program code, this can only be carried out separately by the software developer. As new versions are introduced by the software developer the engineers would make use of the increased

range of functionality. It should be noted that the term "software developer" describes other developers as well as the author. The process is illustrated in Figure 1.1. below.

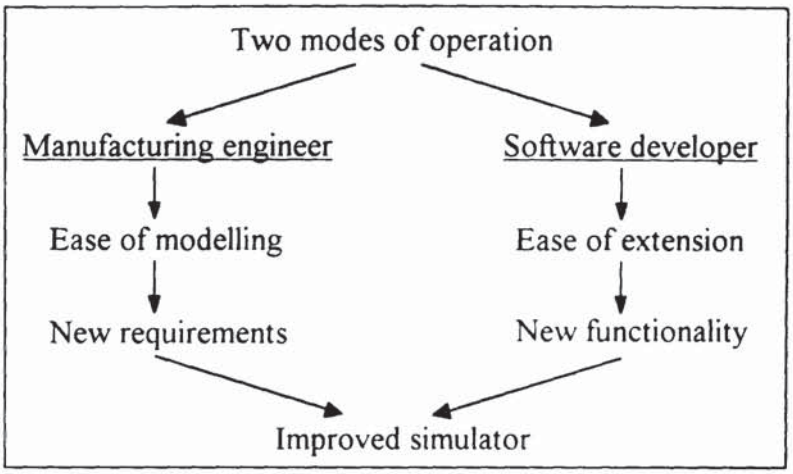


Figure 1.1. Two modes of operation of the Advanced Factory Simulator

AFS therefore has the potential to overcome the compromise typically associated with simulation software; ease of use vs. range of potential application

Chapter 10 discusses the potential of the Advanced Factory Simulator. The simulator has the potential for a wide range of functionality as well as novel ways of viewing a simulation model. The simulator architecture also has the potential to support detailed modelling not just of the manufacturing system but of the whole business. The ability to model the whole business would enable assessment of models in terms of business as well as local measures.

Whilst theory directly relevant to the thesis has been included, more general theory and supporting material has not. Elementary knowledge of manufacturing systems and simulation is therefore assumed. This also applies to some of the object-oriented concepts. Where such concepts are introduced in the thesis extensive references are made to a range of explanatory texts. It is considered that this approach is justified to preserve clarity and to avoid repetition of established theory.

1.4. The software

This thesis argues the case for a different approach to the analysis, design and implementation of simulation tools. The nature of the work necessitates the

development of software. A simulation tool was therefore developed using the principles outlined in this thesis. Discussions in later chapters show that the software can provide both ease of use and potential range of application.

Computer hardware is continually being developed and this has resulted in hardware which is relatively cheap, able to store vast quantities of information and able to carry out computations quickly. There is no reason why hardware speed and storage capabilities will not continue to increase. As a result of hardware developments and user requirements software is also continually being developed and enhanced. As the capabilities of software increase so too do size and complexity. This applies both to software for carrying out tasks such as word processing and data manipulations and to new software being developed.

The simulator developed to support this thesis has made use of the latest software development tools available for the personal computer (PC). This has resulted in a simulator that is both large and complex. The development approach used is such that a significant element of commercial software is required for support. Various commercial software is used as an integral part of the simulator as well as being used to provide an environment in which it can run.

Source code and detailed documentation of the simulator is not covered in this thesis. There are two reasons for this. Firstly full source code and full documentation of a simulator of the size created would require a document many times the size of this. Secondly full documentation would be incomplete due to the extensive use of commercial software (such as the Microsoft Windows Applications Programming Interface (API)). A clear overview and summary documentation are, however, provided.

In order to demonstrate the value of the software individuals from both Lucas Engineering and Systems Ltd. and Aston University were asked to become involved. Individuals from both organisations were asked to build models whilst individuals from Aston were involved in adding new functionality. Clear references are made to this work in the body of the thesis and further details are given in the appendices.

2. Simulation for manufacturing system analysis

> *"Discrete event simulation has emerged as one of the most appropriate modelling tools to plan the design and operation of production systems."*
>
> Chaharbaghi (1991)

This chapter will examine the suitability of modelling techniques for the analysis of manufacturing systems. Characteristics of typical manufacturing systems will be detailed. Following this the ability of a range of analysis techniques to assess those characteristics will be examined. Conclusions will be drawn as to which technique is most suitable. Assessments will be based on the suitability of *techniques*; reference to current *tools* will be avoided where possible.

2.1. Complexities of manufacturing systems

> *"The modern manufacturing system has extraordinary complexity and few individuals understand more than a small part."*
>
> Parnaby (1986)

This section will briefly discuss the complexity of manufacturing systems, thereby identifying the properties required by any suitable analysis technique.

2.1.1. Dynamics

Dynamics is the term used to describe time varying behaviour. Variations over time may be either short term or long term. Coyle (1977) provides the following illustration:

> *"All business firms, economies and social organisations show DYNAMIC BEHAVIOUR. That is, as time passes, the variables by which we measure their condition (such as sales, profits, stocks, balance of payments, unemployment, and many others), fluctuate noticeably, sometimes alarmingly as when cash reserves fall, and sometimes, of course, gratifyingly such as when profits rise."*

There are a number of sources of uncertainty which can affect the performance of a manufacturing system. Such sources of uncertainty contribute to the dynamics. The uncertainty can be categorised as internal or external.

Internal influences relate to the behaviour of plant and personnel. Sources of uncertainty can affect both the local area in which they are situated as well as other areas. For example, machines may be affected by breakdowns, setting, quality variations, etc. Also operators may be absent, affect the performance of the machines, delay the transfer of batches, etc. This will result in variations of the performance of the local area as well as the other areas. For example breakdowns within a cell will result in lower utilisation of machines concerned as well as requiring personnel to carry out repairs. This in turn could result in downstream cells receiving unpredictable deliveries of work.

External influences mainly relate to customer and supplier actions. The pattern of demand volume and mix will vary. Such variations will include long term trends as well as short term changes including order cancellations or adjustments. Depending on the product structure and supply policies, customer orders may involve varying specifications of a base product. This in turn will result in a variation of demand for components that are used for the product. The ability of suppliers to deliver quality products on time is another external influence. External influences could also include, to varying extents, the supply of skilled personnel, credit terms with the banks, timing of payments by customers and access to new technology. The external influences will in turn contribute to the variation of loading on the manufacturing system's resources.

Manufacturing systems are typified by complex dynamics. Such dynamics are closely related to the performance of manufacturing systems. Understanding the dynamics of a manufacturing system is therefore an important part of understanding the manufacturing system as a whole.

2.1.2. Control systems

Manufacturing businesses will employ material and information control systems to support, among other things, the physical production system. Whilst all planning and control systems (referred to hereafter as control systems) aim to manage the process of

converting the customer orders to customer deliveries, different control systems achieve this in different ways.

One form of categorisation of material control systems is push (for example Manufacturing Resource Planning or MRP II (Wight, 1983)) and pull (for example Kanban (Monden, 1981)). In operation pull systems will produce a gradual upstream effect on the production system as attempts are made to replenish stocks. Push systems on the other hand will produce a downstream effect as stock passes through the various production processes before being matched against customer orders. Push and pull systems vary in the way that orders are released and use different mechanisms to cope with the responsiveness of the physical production processes.

The various types of control system are able to respond (in varying degrees) to the behaviour of the manufacturing system. For example, with MRP II it is possible to reschedule orders based on overloading at particular work centres. Control systems can therefore be used to respond to the dynamics of the physical production processes. Control systems also influence the dynamics of the production processes (Gooden, 1988). For example, the release of an order may result in changes to operator deployment, loading at a particular work centre and the requirement for tooling changeovers. This in turn may affect other parts of the production system. For example, the necessity to change over machines may result in starvation of work centres downstream.

Control systems are therefore very tightly coupled with the physical production processes. Control systems could both dampen or accentuate the dynamic behaviour of the production system. Any analysis method that is applied to the physical production processes should also consider the control systems under which it operates. Failure to take into account the control systems could result in an analysis method that lacks validity and may result in incorrect decisions being made.

2.2. Levels of analysis

A manufacturing system can be viewed in a number of ways. For manufacturing systems operation, personnel at different levels of a business will view the manufacturing system

differently. For example the manufacturing system could be viewed by a cell leader or supervisor as a collection of machines, operators and batches; by a works engineering manager as a collection of machines; by a production control manager as a collection of work centres. Typically a cell leader would only be interested in part of the manufacturing system whereas a production control manager would be interested in the whole.

In the case of design, manufacturing engineers would view the manufacturing system at an 'appropriate' level of complexity. For example, a design may involve the configuration of a small number of machines, operators, transportation systems, etc. In this case the design could be considered at one detailed level. If the task involves the design of the whole manufacturing system then several levels of detail may be considered. With increasing detail it is likely that less of the manufacturing system will be considered. For example, the systems engineering approach proposed by Parnaby (1986) views the business first at a macro (whole system) level then later at a micro (individual cell) level. In a survey conducted by Love & Bridge (1988) it was suggested that the design process should be split into three distinct stages: rough-cut, macro-level and micro-level.

It could be suggested that any view of a manufacturing system, whether design or operations related, is part of a hierarchy. Hence a view could be either broken down to provide more detail or joined with others to provide less detail. There is nothing, however, to suggest that there is a single hierarchical structure that encompasses all views. Given there is such a wide variation in company structures, management styles and design methods it is unlikely that a single hierarchical structure could exist. Hence any form of analysis must be able to cope with a wide variation of views.

2.3. Modelling

Kreutzer (1986) states:

> *"A model is an 'appropriate' representation of structures and processes of a miniworld, instantiating some aspects of theory."*

Physical models can be built to gain understanding of some real system (proposed or actual) without having to incur full cost building of that real system. For example, a scale model of a passenger aircraft may be built to investigative the aerodynamics of different designs. Another type of model is a mathematical or logic model (Pidd, 1992c). Such models can be used to assist in the analysis of manufacturing systems.

Models can be built that capture the complexity and interrelationships. Models can either represent existing systems or new designs. Experimentation with the models can then be performed to assess the effects of inputs on the outputs. For example, the levels of work-in-progress and operator utilisation can be compared for different levels of order volume and mix. Manufacturing engineers can then use the outputs of models to assess the relative merits of different designs or changes to an existing manufacturing system.

Models of a particular system can contain varying levels of detail. These levels of detail can match the model users' view of the manufacturing system. For example, a model of a manufacturing system may be aggregate or detailed and therefore may be viewed as a collection of manufacturing cells or a collection of machines and operators respectively. It is important that models are built to the appropriate level of detail (Kreutzer, 1986; Pidd, 1992c). When analysing a system, a model must contain enough detail to enable valid conclusions to be drawn. The detail must also be limited to avoid wasted modelling effort and prevent the introduction of complexity that could erode the clarity of the model. Modelling techniques, and therefore modelling tools, for use in manufacturing should therefore allow models of the 'appropriate' level of detail to be produced.

2.4. Techniques for analysis

Jackman & Johnson (1993) derive a spectrum of modelling techniques for modelling manufacturing systems, see Figure 2.1. Whilst the figure provides a guide to the relative cost and complexity of modelling techniques, on examination of specific tools exceptions may occur. For example Huettner & Steudel (1992) use a combined spreadsheet, queuing and simulation analysis approach to modelling manufacturing systems. In their approach the development of the spreadsheet model took longer than the queuing model. This would imply that the spreadsheet model cost more to build.

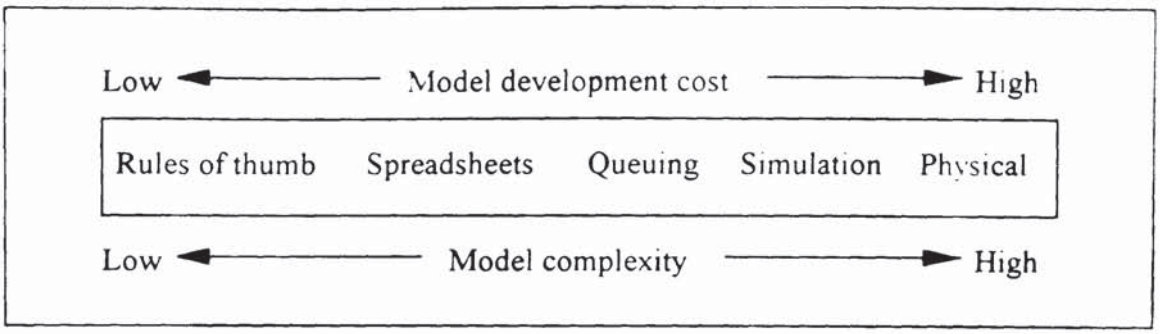


Figure 2.1. Spectrum of manufacturing systems modelling techniques (from Jackman & Johnson, 1993)

The following sections discuss a number of modelling techniques. Examination of the techniques will be in the context of modelling manufacturing systems. Emphasis will be on how well the techniques can conceptually model the detailed dynamic behaviour of a manufacturing system. The term 'conceptually' is used since the following discussion will focus on the advantages and disadvantages of *techniques* rather than how well they have been embodied in current *tools*.

2.4.1. Static mathematical modelling techniques

Static mathematical modelling techniques are perhaps the simplest of the quantitative techniques. Static techniques provide a quick and easy means of examining a system by making a number of simplifying assumptions. Applications may include capacity analysis or lead time calculation in which times are aggregated. For example, for capacity analysis of a machine the machining times of all batches are summed and compared with the total time available. Such calculations are often performed using spreadsheets.

Whilst static techniques use simplifying assumptions to enable quick analysis such assumptions limit their application. Failure to account for the dynamic behaviour of real manufacturing systems may produce an optimistic view. For example, losses at a bottleneck may assume constant availability of material from upstream operations. Whilst on 'average' this assumption may be a good one, in that the upstream operations can normally produce faster than the bottleneck, a breakdown at a feeder machine could cause temporary bottleneck starvation and thus reduce its capacity to less than the calculated figure. Furthermore the inherent random nature of manufacturing systems (such as absenteeism and scrap production) and the effects of competition for resources

(such as allocation of machines for a task or the deployment of operators) are ignored (Jackman & Johnson, 1993).

2.4.2. IDEF modelling techniques

One approach to modelling manufacturing systems is the use of structured analysis techniques to develop a series of hierarchical models. IDEF (ICAM Definition or Integrated Computer Aided Manufacturing Definition) (see Bravoco & Yadav, 1985) employs such techniques. There are three different views that can be adopted:

- Functional view ($IDEF_0$);
- Information view ($IDEF_1$);
- Dynamics view ($IDEF_2$).

In practice models are developed from the functional and information views, compared and used to develop a model of the dynamics view (Bravoco & Yadav, 1985). The notation used in the $IDEF_0$ modelling technique is shown in Figure 2.2.

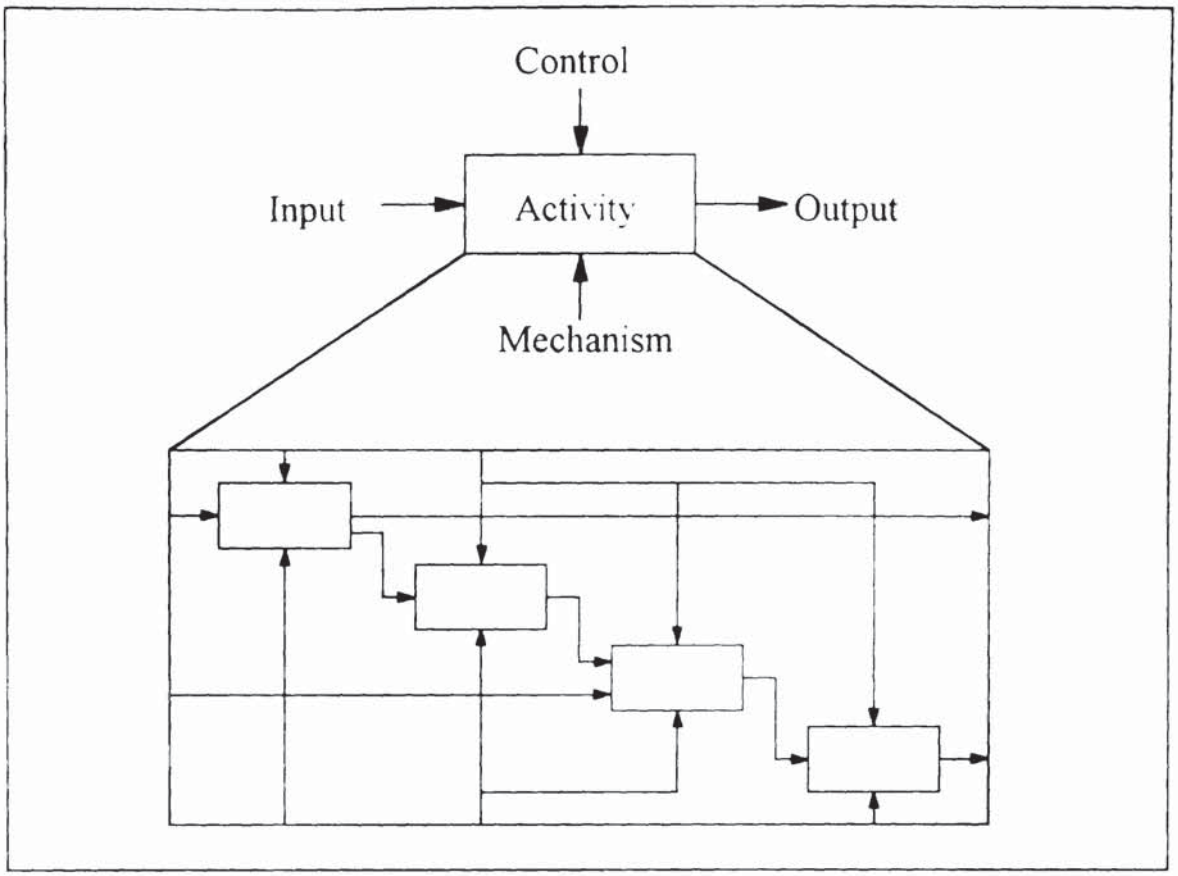


Figure 2.2. The notation used in $IDEF_0$ (from Busby & Williams, 1993)

Busby & Williams (1993) describe the $IDEF_0$ notation:

> *Thus, as a basic building block, each activity to be modelled is represented as a box, labelled with a verb phrase that describes the activity. Attached to the box is a series of arrows, known collectively as constraints. These are labelled with suitable nouns. Constraints that enter on the left are the activity's inputs, which are subsequently transformed into outputs. This transformation is determined or modified by the existance of controls, which enter at the top of the box, and is performed by mechanisms, which enter at the bottom. A model is constructed by building a hierarchy such that an activity at one level is decomposed into a number of more elementary activities at the next lower level.*

The use of hierarchical techniques guides the process of adding detail and provides a structure in which detail is only added when necessary.

As the categories suggest, $IDEF_0$ (functional) and $IDEF_1$ (information) ignore the dynamics of the system being examined. The previous section on static mathematical techniques suggested that ignoring dynamics will produce unrealistic results. Furthermore the notation used for $IDEF_0$ does not employ any quantitative measures and although values can be added the technique does not assist in any analysis (Busby & Williams, 1993). Busby & Williams voice a number of other concerns regarding the use of $IDEF_0$ that question its usefulness.

$IDEF_2$ is specifically aimed at modelling the dynamics of a system. Whilst this would seem useful, the technique merely documents the dynamics and does not provide any form of analysis (see Bravoco & Yadav, 1985). The information can be used as input to a tool that could carry out dynamic analysis. In this light the technique should be compared with information gathering techniques and not analysis techniques. Wang et al. (1993) and Mujtaba (1994) give examples of the use of IDEF techniques to develop dynamic models using another analysis technique. In both examples simulation is used as a means of building and evaluating the dynamic models. These examples should therefore be seen as a means of building simulation models, not as a different form of dynamic analysis. Simulation will be discussed later.

2.4.3. Queuing theory

Queuing theory (for example, Solberg & Nof, 1980) goes some way towards enabling the dynamic behaviour of systems to be investigated. In order to apply queuing theory the user must become involved in programming, however, software tools such as MANUPLAN (Suri & Diehl, 1985) are available to greatly ease the task of creating models. Queuing theory is sometimes referred to as a rapid modelling technique since models can be built and evaluated quickly and easily (Huettner & Steudel, 1992).

The dynamic analysis possible using queuing networks is based on steady state operation and therefore ignores transient behaviour. Transient behaviour can be used to refer to the change of a manufacturing system from a functional layout to a cellular layout or to activities such as the movement of batches. Jackman & Johnson (1993) believe that the

analysis of transient behaviour is crucial since few manufacturing systems ever reach steady state.

Jackman & Johnson (1993) detail a number of deficiencies of queuing theory. Deficiencies include:

- Assumption of infinite capacity of queues / buffers;
- Difficulties of modelling assembly production;
- Flow time and work-in-progress estimates;
- Inability to model scheduling and shop-floor control strategies.

Some of these deficiencies are recognised by the originators of MANUPLAN (see Suri & Diehl, 1985) and others (Chaharbaghi, 1991; Shimizu, 1991; Huettner & Steudel, 1992).

Manufacturing control systems will attempt to both overcome and contribute to the dynamics of a manufacturing system. For example, the timing of the release of work orders, the mechanisms by which operators choose the next batch to work on and the batch sizes are all closely related to manufacturing system dynamics. The failure of queuing theory to model the operation of control systems in detail will limit its application.

It is well recognised that the use of queuing theory is not well suited to the detailed analysis of manufacturing systems (Suri & Diehl, 1985; Jackman & Johnson, 1993; Huettner & Steudel, 1992). It is, however, recognised that queuing theory has potential for carrying out more aggregate, strategic analysis.

2.4.4. System dynamics

Originally referred to as Industrial Dynamics (Forrester, 1958), system dynamics is a modelling technique that views systems as an interacting web of feedback or control loops. System dynamics models are able to capture some of the dynamics and control missing from techniques discussed above. Whilst it can be applied in an industrial

context (for example, Love, 1980) it can be used in other areas such as biological studies.

As with queuing theory, system dynamics models can be created using high level programming languages or using specialist software such as DYNAMO or STELLA (see Kreutzer, 1986; Pidd, 1992c).

Whilst system dynamics can be used to model businesses in an aggregate way (such as modelling the supply chain) its application to detailed modelling is limited; the use of equations for constructing models results in difficulties in representing the flow of individual batches, the effect of machine breakdowns and the problem of effective deployment of operators. System dynamics is therefore an inappropriate medium through which to model detailed manufacturing system designs or the adjustments to operational policies.

2.4.5. Discrete event simulation

The term simulation can be used to cover a range of approaches to modelling from discrete event to continuous simulation (Carrie, 1988). Continuous simulation models are usually based on mathematical equations and therefore suffer similar deficiencies to queuing theory and system dynamics. Since many manufacturing processes are not continuous in nature Wu (1992) believes that:

> *"... discrete simulation is perhaps the most important type of computer simulation used in the design and analysis of modern manufacturing systems ..."*

Discrete event simulation models can be built up by using or creating entities and logic to govern the behaviour of those entities. In a manufacturing context, entities could represent machines, operators, trucks, parts, cells, departments, etc. Evaluation of simulation models is carried out in a time-phased manner. For more detailed descriptions of discrete event simulation see Carrie (1988), Kreutzer (1986), Law & Kelton (1991), Pidd (1992c) and Wu (1992).

Among the advantages that can be claimed for discrete event simulation (referred to hereafter as simulation) when compared with other modelling techniques are the ability to model:

- a wide range of manufacturing systems;
- transient behaviour;
- a variety of levels of detail from modelling detailed behaviour of individual machines to modelling manufacturing cells in an abstract way;
- complex material and information flows including assembly;
- resource allocations such as which operators use which machines and when;
- constrained buffers and therefore buffer related control systems such as Kanban.

Christy & Watson (1983) found that in the early 1980s simulation was regarded as one of the most effective management science methods available. Although the application of simulation in manufacturing remains relatively low (Simulation Study Group, 1991), simulation is one of the more commonly used analytical techniques (Coccari, 1989). Whilst the Simulation Study Group (1991) found a relatively low use of simulation they found high satisfaction rates in instances where it had been used and estimate that millions of pounds are being lost annually through the failure of firms to employ the technique.

One of the commonly quoted drawbacks of simulation is the relatively long time required to build and evaluate simulation models. Such criticisms are often made when comparing simulation to other techniques such as queuing theory (or queuing network models, QNMs). Whilst in some cases this may be true such comparisons have been questioned:

> *"Certainly, the time and effort needed to learn and become efficient with a general purpose simulation language, such as SIMAN, is not trivial. However, there are many good simulators such as ProMod, Simfactory, and XCELL+ which can address the same types of issues as MANUPLAN or OP. The learning curve for such systems is quite short, thus we cannot unequivocally say that building a QNM of a*

system is faster than building a simulation model of that same system."

Jackman & Johnson (1993)

Numerous software tools are able to support the development of discrete event simulation models (see Banks, 1993). There are numerous instances of the development of detailed simulation models (for example, Shimizu (1991) and Townsend & Lamb (1991)) as well as the development of more aggregate models (see Bridge, 1990). Of those documenting the application of simulation significant benefits are typically claimed.

This chapter has examined some of the characteristics of manufacturing systems. A number of modelling techniques have also been discussed. Whilst manufacturing systems are typified by complex interaction and dynamics few modelling techniques are able to provide detailed assessment. Simulation is able to assess detailed dynamics and would therefore appear well suited for application in manufacturing.

Whilst this section has discussed the advantages of simulation techniques over other techniques little has been said about the software tools that can be used to build models. Subsequent chapters will discuss ways in which models can be built. Of particular interest is how well current software tools are able to support the needs of manufacturing engineers.

3. Modelling tools for analysis of manufacturing systems

> *"In our view, a very small percentage of systems that could benefit from simulation are actually simulated, and the primary reason for this is the high level of effort required to employ simulation technology successfully."*
>
> Collins & Watson (1993)

This chapter examines the process of building simulation models. A number of requirements of a tool that best meet the requirements of the user are discussed. In the light of these requirements different approaches to building simulation models are discussed. It is argued that the data driven approach to model building best suits the needs of engineers.

3.1. Requirements of the engineer

3.1.1. The engineer as the model builder

Engineers will typically be involved in the process of improving manufacturing systems. Such activities may involve changes to the manufacturing system design or changes to the policies that the manufacturing system operates under. Authors, such as Parnaby (1986), advocate a systems approach for the design of manufacturing systems. The process of manufacturing system design (MSD) requires the evaluation of factors such as work-in-progress, lead times and schedule adherence. Operational or policy decisions will be made to control such factors These factors and others are directly related to the dynamic (that is time varying) behaviour of the manufacturing environment. It follows, therefore, that failure to account for dynamic behaviour will lead to inadequate design and policy decisions.

Earlier it was argued that simulation was the most appropriate technique to use to assess the dynamics of a manufacturing system; simulation has been well proven in the examination of the detailed, dynamic behaviour of manufacturing systems. Whilst advantages and disadvantages of different types of techniques were discussed, the discussion did not extend to how the techniques would be applied.

The application of simulation to manufacturing systems requires the use of simulation software. The range of simulation software systems will be discussed in detail later, however, it is relevant here to state that such software is typically complex and difficult to use. The nature of many types of simulation software is such that manufacturing engineers are not able to build models without support from simulation experts.

The assessment of the dynamics of a manufacturing system should not be the only criteria on which to base decisions. Many other factors must be taken into account such as possible additional operator training requirements or changes to the design of the tooling. Many of the factors are interrelated. For example, training operators will provide increased flexibility but will cost more. Assessment of these interrelating factors will involve one or more personnel who have detailed knowledge of the products (components, assemblies, etc.) and tooling. Building simulation models often requires detailed knowledge of the manufacturing system and knowledge of what is and what is not important. Ideally, therefore, those directly involved in the decision making process should be involved in the process of building and simulating models:

> *"We think that simulation should be employed as much as possible by manufacturing engineers who have education and experience in design of production systems."*
>
> Ulgen & Thomasma (1990)

Not only would manufacturing engineers, rather than specialist simulation personnel, be able to build more appropriate models but manufacturing engineers would benefit from involvement in the process; model building highlights assumptions made and is able to provide insight which would not be achieved from examination of the simulation results alone.

3.1.2. Building models quickly

Whether simulation is used in design and operational decision making, the time taken to build a model is an important consideration. For operations, assuming a model has been built, the model will have to be changed over time to reflect changes in the manufacturing system. For design, a new model will need to be created or an existing one radically altered. Models often require large quantities of detailed data and therefore

the model building process cannot start until the design process is well underway. The time taken to build a model has a direct impact on the modelling system's utility in the design process (Love & Bridge, 1988). The time taken to carry out manufacturing systems designs varies but would typically be of the order of months. If the time required to evaluate a design using modelling equals or exceeds the time allocated to carry out the overall design task then the model created can only be used to check the resulting design and cannot be used as an integral part of the design process. In summary, failure to allow rapid model building may preclude the use of simulation or reduce the possible benefits.

3.1.3. Building models easily

One factor that has a major impact on the time taken to build simulation models is the ease of use of the software tool. Ease of use of a simulation tool will also influence its use in the design process; the easier a tool is to use the greater potential there is to explore alternative models. There are two aspects to ease of use, firstly the amount of training required to be able to use the tool and secondly the ease by which models can be created and modified:

> *"We believe that the key to making simulation technology more widely used is to make the tools significantly easier to learn and use."*
>
> Collins & Watson (1993)

The amount of training required to use a simulation tool will depend on the knowledge required of simulation and of the tool's user-interface. A typical manufacturing engineer will have minimal knowledge and experience of simulation; a survey carried out in the U.K. showed knowledge of the benefits of simulation to be low (Simulation Study Group, 1991). Worse still, the survey showed that approximately 40% of decision makers in small to medium sized enterprises (SMEs) were unaware of simulation techniques. Additionally it is difficult to find engineers who can actually build models of complex systems (Ulgen & Thomasma, 1990). To account for this lack of knowledge, tools should be easy to learn and use.

The ease by which models can be built will depend on the type of steps necessary (for example programming may be required) and the ease by which a design can be

translated. The latter point is particularly important. The better the match between the model created using the simulation tool and the conceptual view of the manufacturing system held by the user of the tool the easier the translation is likely to be. The translation may refer to either the creation of a model of a design or the implementation of a design using the model.

The ease of model building is best measured by how well the software is able to represent the view required by the user rather than by how well the software is able to present a view that the user can utilise. For example, whilst a person may be able to use a given item of software, the person will not necessarily find this process easy. The ideal situation is where the software matches the requirements of the user. There may be many potential users of a simulation software tool each of who may view the manufacturing system in different ways. For example, one user may be interested in an aggregate view of the whole system whilst another may be interested in the detailed operation of one small part of it. A simulation tool must therefore enable models of varying levels of detail and scope to be built.

3.2. Programming simulation tools

3.2.1. Range of simulation tools

Simulation models of manufacturing systems can be extremely complex. The enormous number of steps that must be carried out to evaluate a single model for a single set of operating conditions necessitates the use of computers. There is a wide range of approaches to build simulation models using computer software (see Law & Kelton, 1991). The approaches used to develop simulation models can be categorised as:

- Programming approaches;
- Data driven approaches.

Within these two categories there is a whole range of approaches that could be adopted, see Figure 3.1.

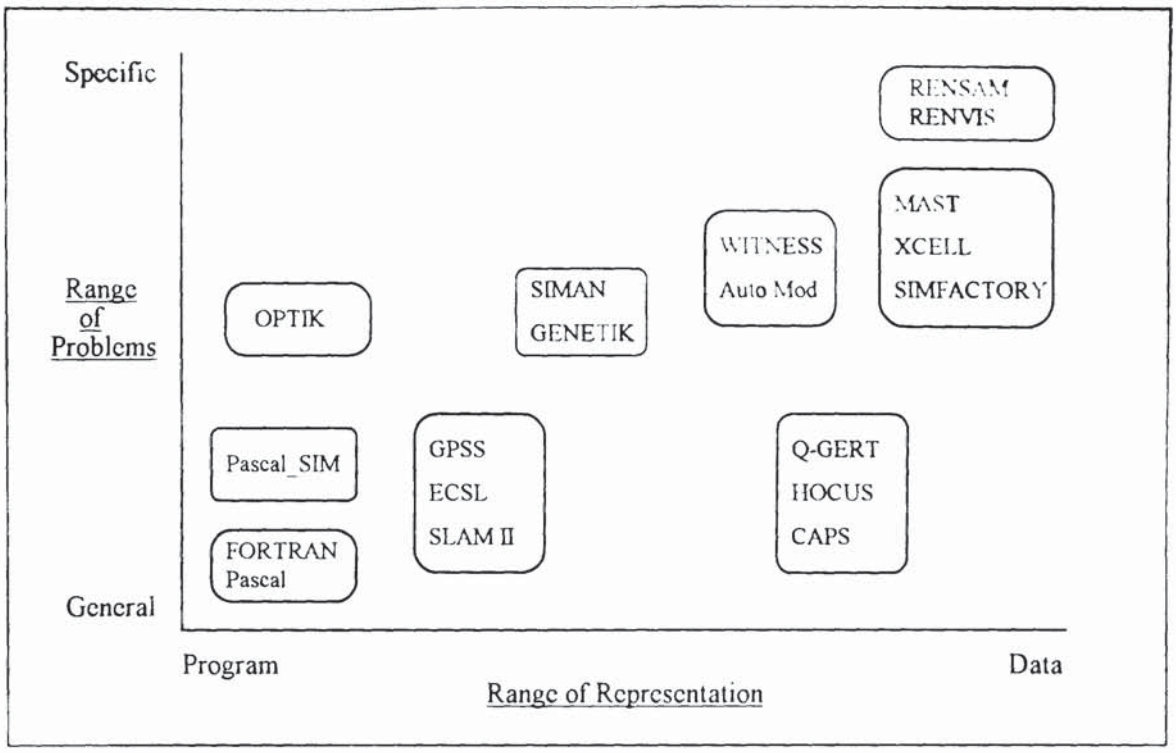


Figure 3.1. Various simulation languages and packages (from O'Keefe & Haddock, 1991)

This section deals specifically with programming approaches, data driven approaches will be discussed later.

3.2.2. Flexibility of languages

The programming approach to creating simulation models involves the end user, or more likely simulation specialist, building a model specific to a manufacturing system. The model builder will require programming skills and experience to implement the model. The extent of the skill and effort required will depend on the programming system being used. Categories of programming approaches include:

- Use of high level language, e.g. Smalltalk, C, C++, Pascal & Fortran;
- Use of a simulation language, e.g. SIMAN, SLAM, ECSL, GPSS & Simula.

Some simulation software allows the model builder to program using the particular simulation language in conjunction with the underlying high level language. Involvement of the model builder in programming activities is the oldest approach to building simulation models with the use of simulation languages being far more common than the

use of high level languages. Perhaps the main advantage to the use of programming languages for building simulation models is flexibility; languages allow models to be built that closely match the needs of the user (Banks et al., 1991; Shewchuk & Chang, 1991). The ability to use languages to closely match the needs of the user has led to a very wide range of applications. For example, the SIMAN simulation language can be used to simulate a diverse range of systems ranging from manufacturing to hospitals to traffic (Pegden, 1985).

3.2.3. Requirement for programming skills

Whilst the use of languages would appear to match the needs of potential users in terms of flexibility the means by which the languages achieve that flexibility constitutes their main drawback. Shewchuk & Chang (1991) in their discussion of simulation tools present a well established view:

> *"General-purpose simulation languages are very flexible and can be used to simulate practically anything, but are difficult and time-consuming to learn and require a good degree of programming skill."*

The use of languages not only requires knowledge of the syntax of the particular language being used but also requires knowledge and skill in software design irrespective of the medium used. This situation is further compounded by the nature of the situation that the languages are being applied to; the complexity of the manufacturing system being modelled necessarily requires the construction of complex software.

Such programming skills are typically confined to those involved in full time development of software and not manufacturing engineers who are required to have a broad range of skills and knowledge in manufacturing. It is possible to train engineers in software skills, however that takes time and effort to attain the level of competence necessary to embark on complex software construction projects. To repeat the point made earlier, it is not easy to find engineers able to build simulation models of complex systems (Ulgen & Thomasma, 1990).

3.2.4. High model building time

Even with training in simulation languages relatively long periods of time are required to construct simulation models when compared to other model building approaches (O'Keefe & Haddock, 1991). Building simulation models using simulation languages involves two stages: firstly the preparation of the data and secondly the conception and expression of the logic to manipulate the data (Ball & Love, 1993). Even though systems are available to ease the task of conversion of the logic into program code the process of conception and expression of the logic is a lengthy one (Love & Bridge, 1988; O'Keefe & Haddock, 1991). To further this the use of simulation languages typically involves a lengthy debugging activity (Law & Kelton, 1991; O'Keefe & Haddock, 1991). Love & Bridge (1988) found that teams undertaking manufacturing system design projects where time scales where short (less than six months) felt, in most cases, that there was insufficient time to attempt simulation studies.

The person requiring the results of a simulation study is unlikely to possess the time and skill necessary to build a model. Often model building will be carried out for the user by an experienced simulation expert. Splitting the roles of model builder and model user presents a number of potential drawbacks:

- the assumptions made by the builder could be incorrect;
- the lead time between the user requesting a change and the change being implemented could be high;
- the opportunity for the user to experiment with alternatives may be limited.

3.3. Data driven simulation tools

The relative merits and drawbacks of simulation languages are well summarised by Shewchuk & Chang (1991):

> *"General-purpose simulation languages are very flexible and can be used to simulate practically anything ...*
>
> *... but are difficult and time-consuming to learn and require a good degree of programming skill."*

The following sections examine the other category of software used to build simulation models, namely data driven.

3.3.1. The no-programming approach

Compared to simulation languages, data driven simulation tools lie at the other extreme of figure 3.1. Whilst simulation languages offer flexibility at the expense of ease of use it is generally considered (Banks, et al., 1991; Carrie, 1988; Pidd, 1992a) that data driven simulation tools offer ease of use at the price of reduced flexibility. This section will discuss the advantages of data driven software highlighting, where necessary, key differences between different implementations.

Data driven simulation tools, commonly referred to as simulators, enable simulation models to be constructed using data only, without the need to undertake any formal programming (Pidd, 1992a). They are able to do this because much of the simulation logic is built-in (Carrie, 1988). The built-in logic will have been tested and verified during the construction of the simulator. By providing parameters or data items the user is able to construct a simulation model. In most cases the model can be run immediately. Many simulators are able to check elements of the model hence the test and debug cycle is reduced if not removed. Usually simulators possess graphical user interfaces that make the task of entering data easier.

Since models can be built without the need to program, users are able to build models relatively quickly. The absence of the need to program will also remove the time consuming iterative development, refinement and debugging associated with producing code (O'Keefe & Haddock, 1991). In some cases the amount of data that needs to be specified by the user may be very low; hence a model can be built very quickly. For example, a model of a manufacturing cell may be built up by specifying the number of machines, a number of parts and their routings. Additional data such as standard times and machine load and unload times may be added.

The absence of the need to program removes a significant barrier to the use of simulation techniques by those not possessing programming skills. Therefore it is possible for manufacturing engineers without programming skills or training to use simulation tools.

3.3.2. User-interface to generating code

Some simulation tools combine the concepts of data driven tools with programming systems. The user is able to specify much of the model using data and will resort to programming in instances where the built-in logic is insufficient. Data driven tools such as ProModel (Harrell & Leavy, 1993) provide a data driven approach and combine this with an internal programming language. Others such as SmartSim (Ulgen & Thomasma, 1990) and Arena (Collins & Watson, 1993) are program generators. With the program generator approach the software tool uses data specified by the user to generate a simulation program. In instances where the software tool is unable provide the necessary logical constructs the user can resort to modifying the generated code.

The extent of user programming required by program generators will vary between different tools and between different models. This approach to model building can offer a powerful, efficient simulation tool (Paul & Chew, 1987). However, the requirement for programming skills may preclude its use by some users. Additionally the time taken to build models may be greater than if the same model could be built using data only. Despite this, time savings of two to three times have been achieved when using program generators when compared to traditional languages (Ulgen & Thomasma, 1990).

3.3.3. Quick and easy model building

Data driven simulation tools possess menus for entering data and will typically include a graphical display for purposes of editing and animation. Many of these tools utilise a 'windows' style user interface (for example ProModel (Harrell & Leavy, 1993); Micro Saint (Hood et al., 1993) and Arena (Collins & Watson, 1993)). Since there are numerous other types of packages (for example databases, spreadsheets and word processors) that possess a similar style of interface navigation through the simulation tools' menus and dialogs can be considered natural and straightforward.

The ease of entering data to build a simulation model will depend on how well concepts and terminology present in the menus and dialogs correspond to reality; the greater the match the easier the process of model building will be. Pidd (1992b) observes:

> *"Design engineering and modelling go hand in hand. The difficulty appears to be the translation of the physical design into the concepts required by a simulation package."*

For example, ATOMS (Bridge, 1990) contains menus and dialogs that allow routings to be created by specifying operation numbers, work centres and standard times and operators to be created by specifying shift times, skills and efficiencies. Since ATOMS possesses concepts and terminology found in reality the process of model building can be achieved efficiently with minimal training. Such observations apply to other data driven tools with similar user-interfaces.

Not only must a simulation tool be able to provide a natural translation between concepts found in reality and those of the tool but be able to present a view of reality consistent with that of the user. This infers that simulation tools should be able to support any level of modelling detail required by the user. The ability of a simulation tool to support multiple levels of modelling detail will allow the user to select the level of detail that is most appropriate. A number of data driven tools have been produced that offer such facilities. For example, MAST (Lenz, 1989) combines queuing theory with simulation to provide two levels of modelling detail whilst ATOMS (Bridge, 1990) combines a single queuing theory modelling level with three different simulation modelling levels.

3.4. Data driven software for manufacturing system analysis

The above discussion on simulation examined the two extremes of creating simulation models; the use of data driven tools and the use of programming languages. Whilst a whole variety of tools exist between these extremes it was argued that the data driven approach offers the most appropriate means by which manufacturing engineers can build models. Data driven tools have the following advantages:

- relatively fast model building;

- relatively low skill requirements;
- potential match of real world concepts and terminology in simulation tool.

The concepts of data driven tools match the requirements of engineers actively involved in the improvements of manufacturing systems, either by design or by changes to operational policies. A recent U.K. survey (Simulation Study Group, 1991) reported that whilst the benefits of simulation were seen to be significant the actual use of simulation was low when compared to other engineering software such as CAPP, MRPII and CAD. It suggested that emphasis was needed on the development of application methodologies and improvements in the education and training of potential users. These conclusions are drawn from a number of responses from simulation users such as 'lack of skill', 'difficulties in data definition' and 'slow model building'. An alternative view might be that existing systems are too difficult to use and efforts should be directed to improving the usability of simulation systems.

On the basis of the above discussion it would appear that data driven tools are well suited to the needs of engineers. Current tools, however, have significant drawbacks. Perhaps the biggest drawbacks are limitations in the range of application and ease of use; tools can only be applied to a small number of situations and skills of abstraction are often required. The drawbacks of data driven simulation tools will be discussed in more detail in the next section.

4. Limitations of current simulators

> *"Where data are readily available generating the input data is easy if the paradigm of the package is shared by the user, but may be difficult otherwise."*
>
> O'Keefe & Haddock (1991)

It has been argued that the concept of data driven simulation tools (typically referred to as simulators) best matches the needs of manufacturing engineers. The previous chapter suggested that the relatively low adoption of simulation tools compared to other engineering software was due to deficiencies in the tools. This chapter will examine a range of data driven simulation tools. Numerous limitations will be highlighted. Subsequent chapters will examine ways of overcoming them.

4.1. Limitations in ease of use and flexibility of simulators

Simulators are data-driven simulation systems that contain pre-written and tested pieces of code to represent machines, operators, etc. (Pidd, 1992a). Through the user interface it is possible to construct models. There are numerous simulators commercially available. Some of the most common are: ProModel (Harrell & Leavy, 1993), SIMFACTORY (CACI, 1992) and XCELL (Conway & Maxwell, 1986).

4.1.1. Data, conceptual and simulation models

The term simulator has been applied to a wide range of simulation systems that use data to construct models. Whilst using data to construct models would seem a user-friendly approach it is important to identify to what the term 'data' refers. Data from the real world (for example, routing files taken from the company database) can be used to create a data model of part of the company or new design (for example, a selection of routings). Data not present in manufacturing databases can be obtained from observations and approximations. For example, machine breakdowns may not be recorded and therefore experience of manufacturing personnel must be used to arrive at initial breakdown patterns. This data model must then be converted into a format capable of being input into the simulation system. Once entered into the simulation system the model may be stored in yet another format. This process is shown in Figure 4.1

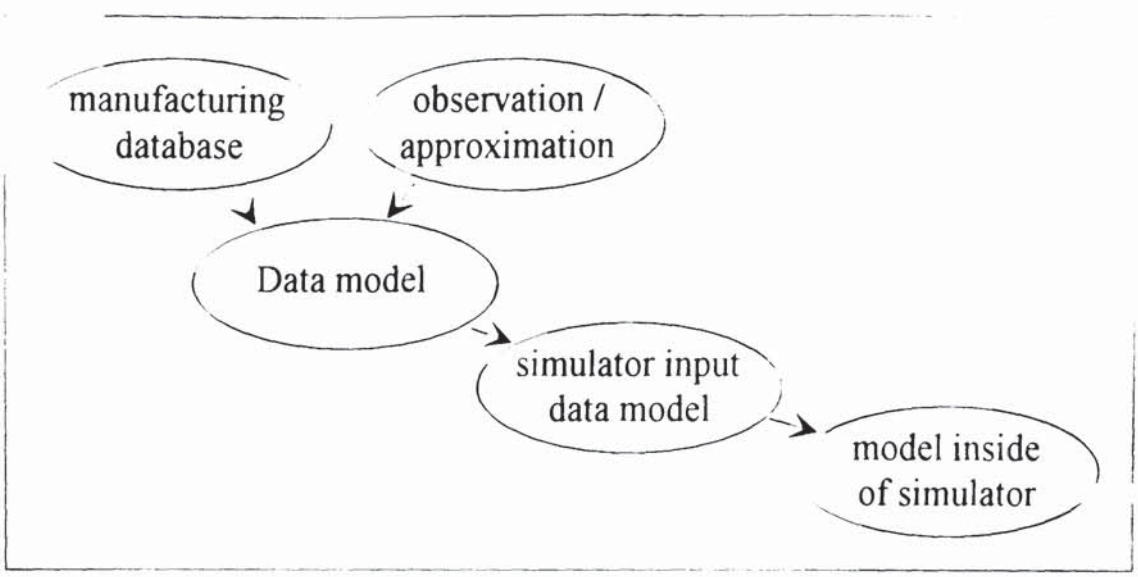


Figure 4.1. Converting raw data to simulator input data.

The conversion process may not be a simple one, for example:

- data may require arrangement or aggregation;
- control rules (for example, the conditions governing the movement of a quantity of materials) might have to be developed;
- data may be lost or extra data may be required.

The less complicated this conversion process is within a simulator the more user friendly it will be; less skill will be required to convert the data, the model creation time will be shorter and the model created is more likely to reflect the original data model. Pidd (1992a) makes an important distinction between generic data-driven systems, such as Witness (Thompson, 1993) which can, in principle, be applied to the analysis of any system, and domain specific systems, such as ATOMS (Bridge, 1990) designed specifically for manufacturing systems applications. The greater the one-to-one correspondence between real world data and simulation model data the easier the software will be to use and the faster models can be created. The correspondence between real and model data may also have implications for the validity of the model.

4.1.2. Manufacturing simulators

There are a number of simulators that are specific to the manufacturing domain. Such simulators include ATOMS (Bridge, 1990), MAST (Lenz, 1989) and SIMFACTORY (CACI, 1992). With each of these simulators a user is able to build manufacturing

models by using the user-interface specifically designed for modelling manufacturing systems. Other simulators may be classed by some as domain specific but such a classification would be debatable. For example, ProModel (Harrell & Leavy, 1993) is promoted as a manufacturing simulator. The user manuals describe the construction of manufacturing models, models of manufacturing systems have been built using it and yet there is little if anything within the software to suggest it is for modelling manufacturing. For example, to create an operator or machine using ProModel then the 'resource' menu option will be used. Within the 'resource' menu the modelling options are equally vague.

Since manufacturing specific simulators employ concepts and terminology that closely match reality they can be considered easy to use. Simulators that employ the appropriate concepts and terminology minimise the abstraction required and thereby reduce the time to build simulation models (Ball & Love, 1992). For example, ATOMS is able to automatically load a routing file taken from a manufacturing database with minimal manipulation.

Tailoring simulators to allow a close match with reality has in practice resulted in specialisation; simulators are dedicated to a particular class of manufacturing problem. For example, ATOMS is well suited to manual discrete part manufacturing but cannot be used to model automated equipment such as Flexible Manufacturing Systems (FMSs), Automated Guided Vehicles (AGVs) or robots. Similarly ProModel is well suited to modelling material handling systems but is weak on modelling operators or labour intensive activities. O'Keefe & Haddock (1991) in their discussion of their FMS simulator, RENSAM, admit potential difficulties in modelling operators. Whilst such specialisation may provide ease of use it may also result in non-use; failure of a simulator to enable all aspects of a manufacturing system to be modelled may preclude its use.

4.1.3. Generic simulators

Generic simulators on the other hand are able to model a wider range of problems, possibly including both FMS and traditional manufacturing system modelling. Generic simulators (for example, Witness (Thompson, 1993) and Hocus (Szymankiewicz et al., 1988)) achieve increased flexibility by utilising more generalised concepts. For example, a simulator may use the term 'resource' that could be utilised to create an operator or a

machine. The creation of a model will use a mixture of data and logic. For example when building a manufacturing system data may be used for a machine name and logic built up from key words to specify how parts are moved to and from the machine.

Since generic simulators can be applied to a range of applications the interface will be a general one and not provide terminology or concepts specific to the problem domain. This will result in a number of drawbacks.

Firstly, the generalised nature of the user-interface means that the one-to-one correspondence of concepts and terminology with the real world will have been eroded. Hence data taken from the real world will need to be converted. This may require both time and skill and will therefore make the task of model building more difficult for the non-simulation expert.

Secondly, not only will the data have to be manipulated but so too will the functionality present within the simulator. For example, Kanban control systems (Monden, 1981) are unique to manufacturing systems. It is unlikely a generic simulator will contain terminology directly relating to Kanban; more probable is the presence of modelling elements that can be built up to provide Kanban properties. Therefore generic simulators will require skill and ingenuity to represent the behaviour of a specific application.

This conversion or abstraction process may be beyond the skill of the user or exceed the time that is available to the user to build a model of the manufacturing system. Therefore generic simulators could be viewed as an extension to traditional simulation languages in which the user must become involved in the logic building process and not a 'black box' in which the user selects, not conceives, the model logic.

4.1.4. Modelling detail

Many simulators, whether specific to the manufacturing domain or otherwise, are unable to model concepts found in the real world with ease. A number of examples that are typical deficiencies are illustrated below.

Firstly, consider the modelling of Kanban material control systems. Some simulators only provide limited modelling of Kanban (for example ATOMS (Bridge, 1990) is only able to model a simple two card system). Even simulation languages that are supposedly more flexible than simulators model Kanban inadequately (Mejabi & Wasserman, 1992).

Secondly, consider the modelling of operators. A majority of manufacturing systems are still under human control, whether at the level of machine operation or cell control (this can be inferred from the analysis presented by the Commission of European Communities (1991) showing the adoption of automated production systems to be very low). This would suggest that humans are a significant source of the dynamics of a manufacturing system. Despite this, current simulators play down the significance of operators, often modelling them as a resource to be acquired and released by batches of parts.

4.2. Limitations of integrated modelling tools

As introduced in the previous chapter, there is a whole range of tools that can be used to build simulation models. So far only the two extremes, namely simulators and simulation languages, have been discussed in detail. This section will examine tools that combine features of both simulation languages and simulators through integration. It will be shown that such modelling tools are limited either in scope or ease of use.

4.2.1. Program generators

The use of program generators is an alternative route to building simulation models. Here the user will specify the modelling elements and the software tool will generate simulation code that represents those elements. This code can then be executed to evaluate the model. Some program generators have user-interfaces that are typical of simulators and the tool would therefore be used in the same way as simulators. Examples of program generators are Arena (Collins & Watson, 1993), SmartSim (Ulgen & Thomasma, 1990) and AUTOSIM (Paul & Chew, 1987).

Two key drawbacks are associated with program generators. Firstly, their use implies the necessity to become involved in programming. Secondly, the user-interfaces of current program generators are limited because they present concepts and terminology

that are inconsistent with the real world. This latter point is discussed in more detail in section 4.4.

The disadvantages of the use of a programming approach to building simulation models have already been outlined. Whilst program generators have user-interfaces to shield users from much of the programming effort, some programming may still be required. In their discussion of SmartSim, Ulgen and Thomasma (1990) state:

> *"In SmartSim we have not yet reached our goal of providing an environment in which a manufacturing engineer can build simulation models without knowing any programming language."*

The amount of programming effort required to complete a simulation model will vary between different tools and different models. In a discussion of program generators Crookes (1987) states:

> *"It is generally accepted that about 70% of the final text of a practical program may be produced by a good program generator. This implies that new programming input is required in most simulation projects using generators."*

Paul & Chew (1987) estimate that the amount of code program generators produce is 95% or more of the total required. It is assumed that these figures refer to the volume of code and not the programming effort. If that is the case then these figures would appear optimistic; since the program generator is unable to generate the remaining code it would suggest that the remaining code is relatively complex and difficult to formulate. Paul & Chew (1987) conjecture that the requirement of the user to become involved in programming activities acts as a barrier to the use of program generators.

It is appreciated that both SmartSim and AUTOSIM are research projects. The commercial software tool Arena may differ. Arena employs a hierarchical or layered interface for generating SIMAN (Pedgen, 1985) simulation code. These layers or templates present different views that can be utilised for model building. Figure 4.2

shows the aggregation of four SIMAN base modules into a 'serve' module. Similarly the serve module can be aggregated with other derived modules to form another layer.

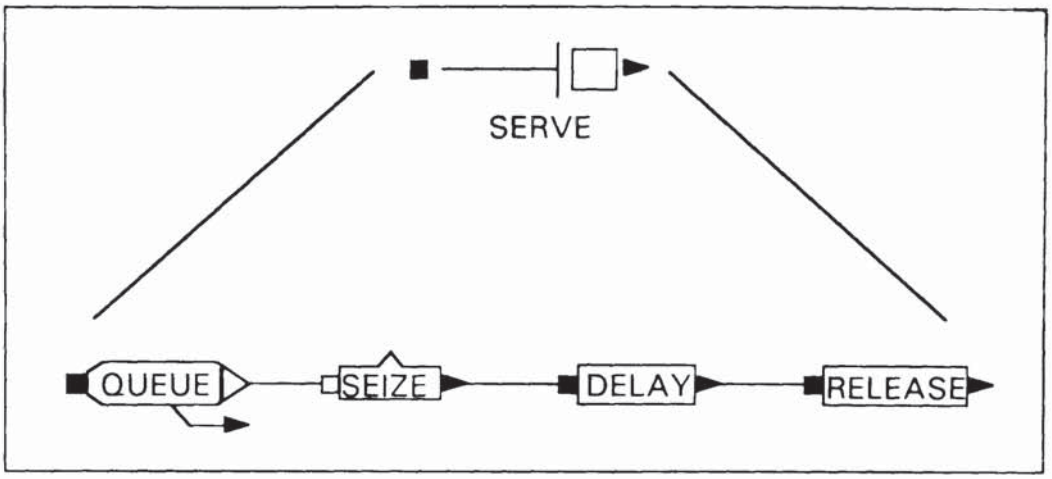


Figure 4.2. The Serve Module (from Collins & Watson, 1993).

The templates range from those presenting a view of manufacturing, to a more abstract view involving resources, servers and queues, to a very abstract view represented by SIMAN blocks. It is possible to create simple models using the least abstract template without the need to program. However, the situation will be different for more complex models. As the model requirements become more complex the user will have to resort to the use of the more abstract templates and possibly even the SIMAN language. Whilst this approach may shield the user from programming in the true sense of the word it is inevitable that the more abstract templates will need to be employed. The use of the abstract templates suffers the same drawbacks as the use of generic simulators. It is interesting that Banks (1993) classifies Arena as a simulation environment rather than as simulation modelling software.

4.2.2. Combined analytical and simulation tools

An alternative approach to building simulation models by programming is to use a combined analytical and simulation modelling system. Here a user would create a model of a manufacturing system using an analytical modelling tool. Such a tool would be primarily data driven. The analytical tool would allow relatively quick model building followed by extremely fast model evaluation, possibly a matter of seconds. Such speed is possible due to the relatively low data requirements and low computational complexity when compared to simulation. The analytical tool would then be used to generate a

simulation model via a program generator. An example of such a system has been developed by Shimizu (1991), see Figure 4.3. Examples of where analytical models and simulation models are used but have not been linked to allow automated data and logic transfer (for example, Dietrich & March, 1985; Haider et al., 1986; Kiran et al., 1989; Huettner & Steudel, 1992) are not relevant to this discussion since they provide little if any assistance to building the simulation models.

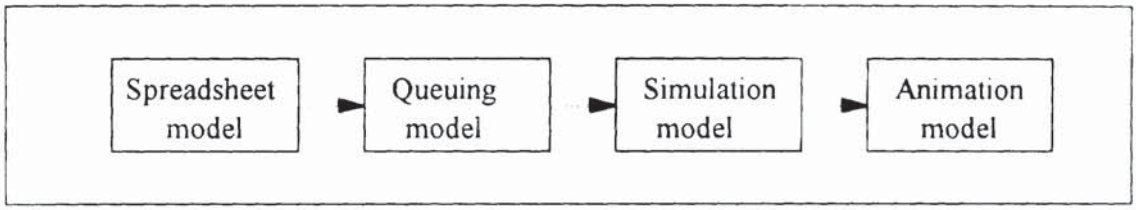


Figure 4.3. Sequential interfacing approach to integrated modelling (from Shimizu, 1991)

The combined analytical and simulation approach to manufacturing system evaluation can be advantageous. The hierarchical nature of the technique allows high level decisions to be taken early and detail is only added later when required. The early stages of evaluation using queuing models such as MANUPLAN (Suri & Diehl, 1985) allows very fast model evaluation.

The combination of analytical and simulation techniques is not without disadvantages. Analytical tools are unable to model certain aspects of manufacturing systems such as assembly and buffer sizes (Jackman & Johnson, 1993). Failure to include some aspects of the manufacturing systems means that early decisions will be made using unrepresentative models.

The combination of analytical and simulation tools involves the creation of two models: one analytical model and one simulation model. Examples of current combinations of the tools involve creating the simulation model with a simulation language. There are no examples of creating the model using a simulator. The simulation model could be created using a program generator. The problems associated with program generators were discussed earlier. Even if the program generator employed to convert the analytical model is able to generate all the simulation code, programming will still be required. Since analytical tools only model at an abstract level and ignore aspects such as assembly

then detail will have to be added manually to the simulation program. Therefore the user must become involved in programming activities to account for the deficiencies in analytical tools.

4.2.3. Combined IDEF and simulation tools

There have been attempts to combine tools able to represent IDEF models with simulation tools. IDEF (ICAM Definition or Integrated Computer Aided Manufacturing Definition) (see Bravoco & Yadav, 1985) comprises three methodologies (functional, information and dynamics) that can be used to build models representing manufacturing systems. By developing software tools to automate the process of developing IDEF models the tools can be linked to simulation software for creating dynamic models. These models can then be evaluated. There are a number of examples of such systems (for example, Wang et al. 1993; Mujtaba, 1994).

Busby & Williams (1993) voice a number of criticisms of IDEF techniques both in terms of the modelling process adopted and the usefulness of the results generated. In addition there are the associated drawbacks seen with the integrated tools discussed earlier; integration of tools with simulation invariably involves a program generator approach. The whole software system will therefore suffer from deficiencies in both the program generator and the language of the generated program. Both Wang et al. (1993) and Mujtaba (1994) discuss the level of detail of the simulation models being created. In both cases the simulation models are high level in nature and do not attempt to address detailed issues such as activities at a work centre or the activities of operators. Wang et al. (1993) discuss the ability to add detail where appropriate but do not discuss how; presumably programming would be required.

Failure to address detailed issues easily means that such tools are limited in their application. The requirement of the user to become involved in programming activities results in a tool that requires skill to use.

4.3. Limited range of modelling detail

Manufacturing personnel view the manufacturing system in different ways. For example, a cell leader may view the manufacturing system as a collection of machines, operators, parts, tooling, etc. whereas an operations manager may not view the system in any greater detail than cells or work centres. Similarly in the process of manufacturing system design advocated by Parnaby (1986) the design process proceeds hierarchically. The hierarchical process first considers the manufacturing system in an abstract way that is gradually decomposed into cells and later individual machines as the design process progresses. See Figure 4.4.

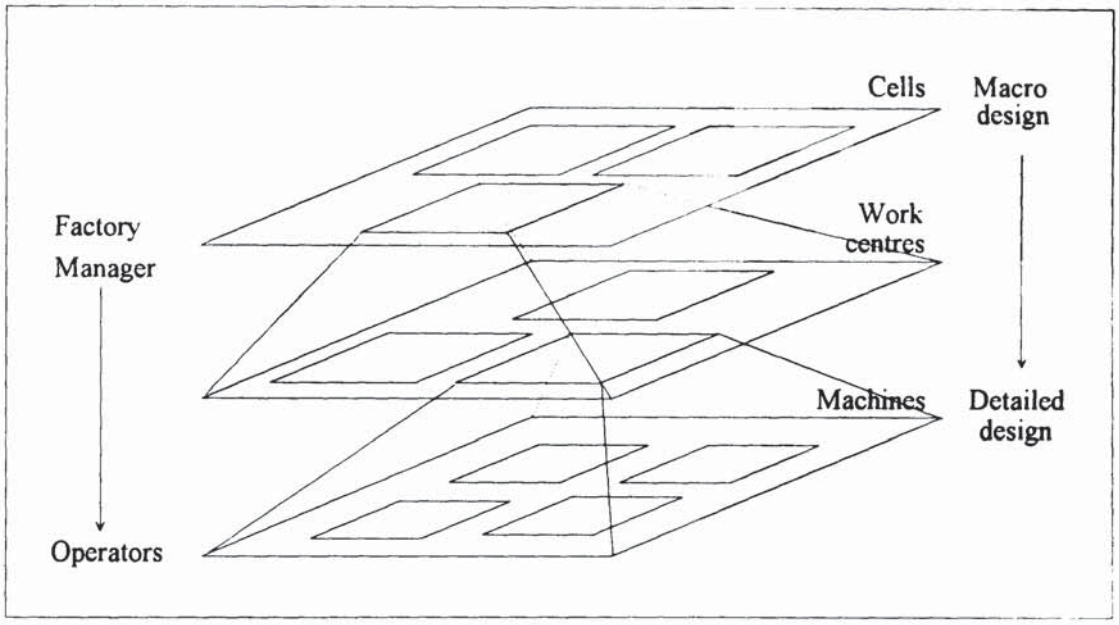


Figure 4.4. Multiple views of a manufacturing system

Support for dynamic evaluation of the various views of the manufacturing system would suggest the need for a simulation tool that is capable of supporting multiple levels of modelling detail. Some simulation tools are able to support multi-level modelling, for example ATOMS (Bridge, 1990) and MAST (Lenz, 1989).

Whilst some simulation tools are able to support multiple levels of modelling these levels are limited. For example, ATOMS only supports three levels of modelling. These levels only include the machine related aspects of the model. Other aspects such as machine repair can only be modelled at one level, hence abstract modelling of repair is not possible. Also the abstraction between levels can be questioned. For example, ATOMS

supports a 'work centre' modelling level but this level is unable to model reliability even in an abstract way.

Generic simulators can implement multiple levels of modelling detail but this relies on the expertise of the user. For example, a resource could be used to represent a machine, a work centre or a cell. It is up to the user to decide how to implement the model. For example, a resource may represent a cell at an abstract level. In this case the user must decide how to set up the parameters of the cell depending on the mixture of parallel and sequential machine working, batching rules and the number of operators.

It can be concluded therefore that simulators provide little help with multi-level modelling. In the case of manufacturing simulators that do support multi-level modelling some of the abstract (less detailed) levels contain deficiencies that may prevent the desired model being created.

4.4. Inadequate representation of the real world

Ideally a simulator should be able:

> *"... to provide the end user of the product with a tool that closely matches the real system being modeled--hence the user is presented with concepts and terminology that are focused to his or her problem."*
>
> Collins & Watson (1993)

There are many simulators that have been developed to achieve this, however, many contain deficiencies. Deficiencies exist in how simulators represent concepts found in the real world. Failure of simulators to represent real world concepts well will result in the following difficulties:

- translation of data from the real world into the simulator;
- creation of the desired model;
- understanding by others of how the simulation model relates to reality;
- relating the results to reality.

Many simulators adopt a material oriented view whereby the passage of entities through the model drives the simulation. For example, ProModel (Harrell & Tumay, 1993), Arena (Collins & Watson, 1993), ATOMS (Bridge, 1990), SimFactory (CACI, 1992) and others model machines, operators and batches by focusing on the batches. Hence the machines and operators are resources to be seized and released by the batches. Whilst such a view may allow internal efficiency it is a distorted view of the real world in which operators are dominant. Adoption of such a view can make the selection of a batch from a queue based on a particular rule difficult to model.

The representation of material control systems is also poor. Many software tools are unable to adequately model Kanban (Mejabi & Wasserman, 1992). In the case of modelling centralised material control systems, such as Material Requirements Planning (MRP), few if any tools are able to provide adequate modelling. Many tools are able to accept files containing lists of orders to complete. This, however, ignores important interactions such as the delays associated with releasing orders and the discrepancies between the physical stock situation and the computerised stock records.

The modelling of quality is also poor. ATOMS models scrap and its removal by assigning machines scrap rates. Other packages (Arena, ProModel, SimFactory) model scrap by the use of defect rates at inspection stations. There are no simulation tools that separately model the production and detection of scrap. In such a situation incapable machines would produce defective batches that could later be examined at inspection stations. Such an approach would enable the effect of positioning inspection stations and the effect of different sampling rates to be modelled implicitly. It would also avoid situations possible with many commercial simulators where even if parts had yet to be machined scrap could be detected!

Other discrepancies include the change over times between parts being constant and time to respond to broken down machines being zero.

4.5. Difficulties of software modification

4.5.1. Improving functionality by modification

This chapter has so far discussed the deficiencies with simulators and integrated software tools. Some of the deficiencies are due to the inability to model the detail or range of activities required. Instead of resorting to a different analysis technique or developing a new simulation tool one option could be to modify existing tools. Modification would involve the addition of new functionality to account for the deficiencies.

Simulation tools, of which simulators are no exception, are used to build models of complex manufacturing systems and evaluate their performance over time. This implies that simulation tools are complex and the complexity is compounded by the interaction between elements of the software over time. Modification would first require an understanding of the existing complex code and then addition of new complex code.

Of course modification can only be attempted if the source code to the simulation tool is available. O'Keefe & Haddock (1991) discuss situations involving commercial simulation packages where 'hooks' are available to add new subroutines. They conclude that the use of such facilities would require considerable knowledge of the package. Such modifications are only desirable if they act on the package itself and not a specific model; modifications that act only on a specific model cannot be carried across to future models.

4.5.2. Design of simulator software

Conway et al. (1959) discuss the requirements for constructing simulators. Whilst they use the term 'simulator' to describe simulation software developed for a specific problem with a number of alternatives their comments are equally applicable to the development of simulators that can be used to address a range of problems in a range of companies. They believe that:

> *"If the simulator must be capable of modification to represent each of the initial set of alternatives and must be able to accommodate additional alternatives whose nature is not known at the time the*

simulator is being prepared, it is imperative that flexibility and provision for rapid, simple modifications be primary considerations in the construction of the simulator ..."

Few, if any, simulators will have been designed with such an approach and thus modification will be difficult.

There is a relatively new approach to software design and implementation currently gaining popularity. This approach is known as object-orientation (see Booch, 1991). The details of this approach are discussed in more detail later. Suffice to say now that the approach has been promoted as a means of developing software that is more amenable to modification than more traditional approaches.

It would appear therefore that simulation tools developed using object-oriented programming techniques would be easier to modify. Ease of modification would therefore ease the task of adding new modelling functionality. However, this does not necessarily follow, for example ProModel is promoted as being developed using object-oriented techniques (Harrell & Tumay, 1991) and yet there have been no claims of new functionality being added. This is surprising since there is a commonly held view that simulators are limited in their application.

4.5.3. Accounting for existing deficiencies

Even if it is possible to extend existing simulation software to allow new functionality to be introduced there is the issue of the adequacy of the current functionality. The new functionality will inevitably interact with the existing functionality. Deficiencies in the existing functionality may have to be carried forward to the new functionality.

Deficiencies in the existing functionality could be rectified by modification. However, some of the deficiencies in the functionality of current simulation tools discussed are so well embedded within the tools that modification of the tool would be unthinkable. For example, modifying logic such that operators could use machines rather that machines using operators would be extremely difficult.

4.6. Compromise of ease of use and range of application

This chapter has discussed the various limitations of data driven simulation tools. Of the range of tools and techniques discussed each suffers from one or more of the following deficiencies:

- limitations in ease of use;
- limitations in range of applications;
- incomplete representation of concepts found in reality;
- inconsistent representation of concepts found in reality;
- limitations in the level of modelling detail.

Some of the above are unique to particular simulation tools and could be overcome by modification. Other deficiencies, however, are more serious. It has been shown that current tools are a compromise between ease of use and potential range of application. It is therefore not possible to provide manufacturing engineers with a tool that can be applied easily to a wide range of applications.

The situation has been summarised by Pidd (1992b):

> *"... simulation tools should be designed in such a way that they have sufficient power to allow proper work and yet should be simple enough for an untrained person to use. Clearly these two aims are in conflict and the challenge is to develop tools which go some way toward meeting both of them."*

Subsequent chapters will examine whether this compromise of ease of use and range of application can be overcome by adopting a different approach to the design and implementation of simulators.

5. Potential of object-oriented techniques

> *"The problems we try to solve in software often involve elements of inescapable complexity, in which we find a myriad of competing, perhaps even contradictory, requirements."*
>
> Booch (1991)

It has been established in preceding chapters that, although simulation has potential to assist manufacturing engineers in the analysis of manufacturing systems, current tools are limited. The limitations are such that current tools are a compromise between ease of use and range of application. This section will examine software construction techniques and argue that some of the deficiencies in current simulation tools can be attributed to the style of software construction. Whilst there is now the potential to overcome the deficiencies using object-oriented techniques attempts to date have failed to exploit it.

5.1. Object-oriented vs. functional approach

This section will briefly introduce and discuss the application of object-oriented techniques to software construction. Whilst the object-oriented paradigm has its roots in the development of the Simula simulation language (see Dahl & Nygaard, 1966) its popularity and application have only recently been established. This section will highlight the root causes of many of the deficiencies that can be found in current object-oriented simulation tools.

5.1.1. Object-oriented software

The effort required to develop a well structured program will depend on its size. Small programs of a few hundred or thousand lines of code perhaps do not need a systematic approach to design or implementation (Ormsby, 1991). Such programs may be written quickly for a specific task such as manipulating files or carrying out calculations. It is likely that small programs are written by a single author and any modification will be carried out by the same author. If the developer is also the sole user of the program then the reliability and interface will not be primary considerations. If there are other users then the situation is different since the way in which the program is used and the limitations may not be apparent.

The situation is different if the program is 'large'. The term large is used to describe programs that are many thousands of lines long, possibly requiring a team of developers to complete in a realistic time frame. Large is also used to indicate the complexity of the program; typically a developer involved in large program development will be unable to understand the detailed operation of the whole program. Also modification and extension could be both difficult and error prone.

The design of the software architecture and the user-interface is therefore important for software that is complex and will have users that were not involved in the original program development. The problems associated with the development and maintenance of complex software led to the development of structure programming techniques (Booch, 1991; Taylor, 1993). Such developments were based on decomposition in which key mechanisms of the program design were broken down into sub-mechanisms, and so on. See Figure 5.1.

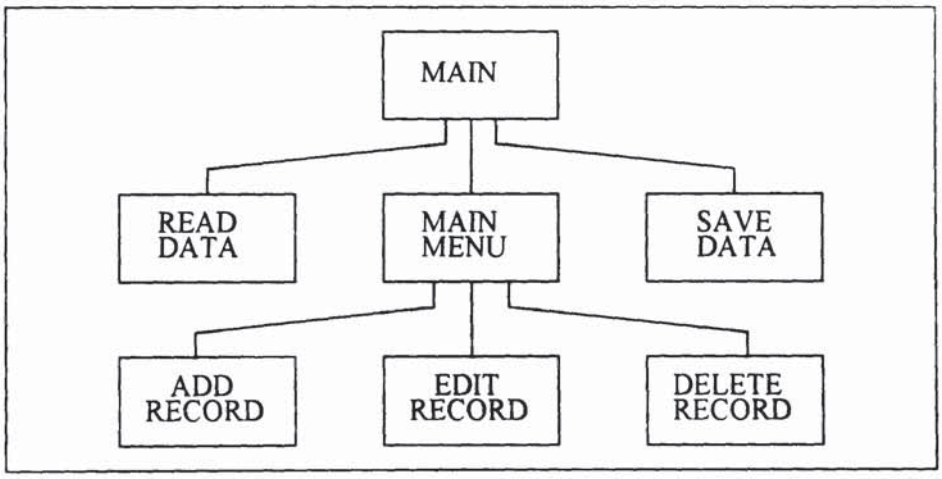


Figure 5.1. Program with 2 levels of nesting (from Taylor, 1993)

The functional decomposition approach, however, failed to account for the other key element of a program, namely data (Taylor, 1993). Decomposition of the mechanisms therefore only contained part of the program complexity. Failure to contain the data meant that although the mechanisms were contained the actions of those mechanisms were not. Complexity therefore remained and often gave rise to errors during program execution (Booch, 1991), the sources of which were difficult to trace. The complexity in turn acted as a barrier to modification and extension.

There are a number of software design techniques, one of which is object-oriented. The object-oriented approach is better able to cope with complexity (Booch, 1991). Software that has been developed using an object-oriented approach will bundle mechanisms and data together. Such bundles can be referred to as classes. A class will have a number of data items that can be accessed and manipulated via the class's 'methods'. Methods are similar to functions or procedures in that they can be used to return data or action code. Methods, however, differ in one key respect: they are 'tied' to a class. The binding of methods and data into a single entity (class) has important consequences for software development.

In some cases the class developed for the program will have an equivalence in reality. For example, in software terms a car class may have data or attributes such as maximum speed and colour and methods such as accelerate and turn. Objects are instances of classes. For example, a particular car with a speed of 100mph and colour red is an instance (or object) of the car class. There may be many other car objects that can be grouped into the car class. Within software, the code can be referred to as classes and instances of those classes created during software execution can be referred to as objects.

The application of the object-oriented approach to software development is eased by the availability of programming languages (for example, Smalltalk, Simula, C++ and Turbo Pascal). Such languages support the key concepts deemed to be prerequisites of object-oriented. Booch (1991) details a number of concepts considered to be essential:

- Abstraction;
- Encapsulation;
- Modularity;
- Hierarchy.

Booch (1991) and a number of other authors (Cox & Novobilski, 1991; Rumbaugh et al., 1991; Taylor, 1993; Graham, 1994) provide more in-depth descriptions of object-

oriented concepts. The benefits of the combination of these concepts when compared to structured techniques are also detailed.

The application of object-oriented concepts enables software developers to contain complexity. The ability to contain complexity enables software to be developed that is not only easier to understand but also easier to modify and extend. The ability to contain complexity within objects enables objects to be reused, both within the same application or by other applications. Cox & Novobilski (1991) describe the concept of Software-ICs analogous to Eli Whitney's interchangeable parts concept; objects developed by one author could be used in turn by others.

In theory, the development of object-oriented software has significant advantages over software developed using more traditional approaches. The development of object-oriented software is relatively recent and therefore evidence to support the claimed advantages are limited. Graham (1994), however, discusses a number of applications where significant benefits of using the object-oriented approach have been claimed. Benefits include reuse of existing software that has been tried and tested, rapid creation of software using object libraries and reduced maintenance requirements once software is in use.

5.1.2. Object-oriented design

The previous section briefly examined two contrasting approaches to software construction: procedural and object-oriented. Software construction requires the use of a programming language. Software implemented using an object-oriented programming language can allow complexity to be contained and allow easy modification and extension.

There are a number of object-oriented programming languages, each of which has advantages and disadvantages of use in different situations. Choosing a 'good' language will not result in 'good' software per se; the programming language is just the medium through which a software design is implemented. Hence a poor design implemented by

using an object-oriented programming language will most likely result in software which performs poorly in use.

Irrespective of whether an object-oriented programming language is used to construct software, the design of the software must be considered. Design is important for two reasons. Firstly a good design is likely to result in a product that matches the objectives or specification. Secondly a good design is likely to save time and effort both in terms of the initial effort of producing the software and the subsequent effort involved in maintenance. The issue of maintenance is important. Graham (1994) notes that some industry commentators estimate that over 50% of programmers' efforts are devoted to corrective maintenance and that corrections can cost ten times as much as new developments.

The process of software design is typically iterative and is unlikely to be right first time (Booch, 1991). Some designs will involve the application of software to areas that are new to the developers. In such cases elements of the design will not be proven until development is underway (Booch, 1991). The design process should therefore be robust to changes and not require extensive effort to make alterations. The design process, whether embedded in the process of 'hacking' or a more 'formal' design methodology, can be a significant part of the overall software development process. A good design methodology is therefore important.

There are a number of object-oriented design methodologies. Some methodologies have been developed specifically for the object-oriented approach whilst others have been adapted from structured design methods. Perhaps one of the earliest methodologies was developed by Booch (see Booch, 1991). Reviews of various design methods are given by Ormsby (1991) and Graham (1994). The use of object-oriented design methods with object-oriented programming languages is not necessary, however from a notation point of view it offers a far more efficient route than mixing object-oriented techniques with structured techniques (Graham, 1994). Furthermore structured design techniques are not robust to change (Taylor, 1993) whereas a combination of object-oriented design and programming offers a smooth and robust transition (Graham, 1994).

Software will need to support concepts and terminology present in reality. Ideally design methods should support, even emphasise, the relationship between real world and software objects. A good design method is one that eases the process of capturing concepts and terminology in the real world and translating them into software objects. This approach enables software to be understood and used more easily in practice. Object-oriented design methodologies attempt to do this. The use of object-oriented programming techniques and object-oriented design methodologies can offer a powerful combination for software development.

5.2. Object-oriented programming techniques

The use of object-oriented programming techniques can result in the development of software that is easy to understand, easy to modify and easy to extend. The concepts of abstraction, encapsulation, modularity and hierarchy enable such development. The use of these concepts can be developed further to change the 'world view' employed.

5.2.1. Matching the manufacturing engineers' and programmers' views

Traditional approaches to software construction focus on the necessary mechanisms and data structures required. The view employed here is a programmer's view not an engineer's view. An undesirable result of this could be that the underlying programmer's view will be apparent in the user-interface of the software. The engineer may therefore find the software difficult to use.

With the object-oriented approach a developer creates a library of classes (known as an object-oriented library). During program execution, instances (or objects) of the classes are be created. The objects will be a combination of data and methods and interact with one another in a predetermined way. The form these objects take and their interaction will depend on the approach taken by the developer.

People can view the world around them as objects (Adiga, 1989). People can also form an understanding of how these objects interact. For example, a manufacturing system could be viewed as a collection of machine objects that are operated by people objects. Similarly, railways could be viewed as collections of track objects along which train

objects move. If programmers are developing software models involving elements of reality then those elements could match the object-oriented view used by people (Adiga, 1989; Najmi & Stein, 1989; Taylor, 1993).

The concept of matching the programmers' view with the engineers' view has a number of advantages:

- The software could be easier to understand from the programmers' point of view: the object-oriented view has the potential to provide a 'standard' view of the software mechanisms;
- The software could be easier to understand from the engineers' point of view: if the underlying software construction concepts appear in the user-interface then the user-interface is likely to be easy to understand and use.

Cox & Novobilski provide a good illustration of the concept of matching the users' and programmers' views, see Figure 5.2.

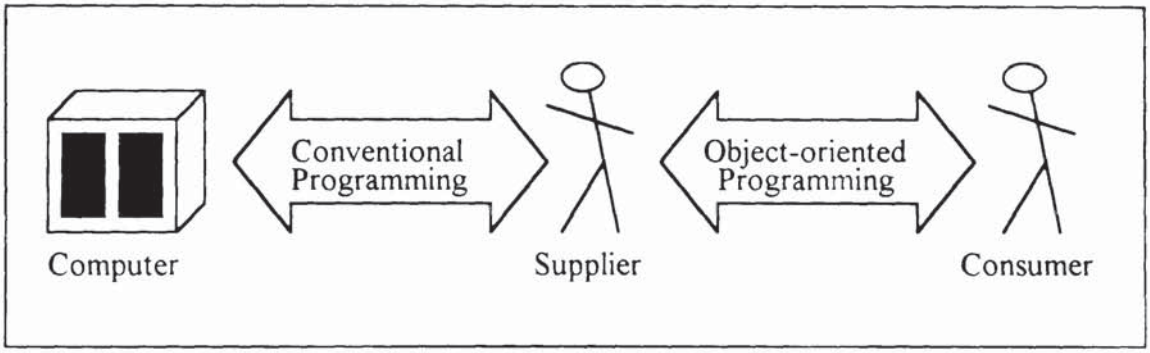


Figure 5.2. Program-building versus system-building. Conventional programming tools emphasize the relationship between a programmer and his code, while object-oriented programming emphasizes the relationship between suppliers and consumers of code. (Adapted from Cox & Novobilski, 1991)

5.2.2. Programming discipline

The benefits of programming using object-oriented software are well documented (Booch, 1991; Cox, 1991; Taylor, 1993; Graham, 1994), object-oriented programming languages can be used to create software that is easy to understand, modify and extend. As discussed earlier, these are *potential* benefits, they are not automatic. The

programmer must design and implement the software in such a way that the benefits are realised.

Similarly the programming languages can be used to develop software in which the programmers' view and the engineers' view are the same. Again providing such correspondence is the responsibility of the developers. The developers must understand the engineers' view and implement it. Failure to provide a match in world views may be due to a lack of understanding or a lack of intent.

5.3. Current object-oriented simulation libraries

5.3.1. Use of object-oriented approach in simulation

The process of carrying out a simulation study involves the creation of a model of the system under analysis. Ideally the model should be built in such a way that elements of the model clearly represent elements found in reality. Taking such an approach will enable both the model and the results from simulation of the model to be easily understood by users. The nature of the object-oriented approach is therefore ideally suited to simulation modelling; objects from the real world can be replicated inside the simulation model (Adiga, 1989; Najmi & Stein, 1989).

Using an object-oriented approach for the development of simulation libraries offers a number of advantages (Najmi & Stein, 1989; Shewchuk & Chang, 1991):

- Reusability of existing code;
- Rapid creation of new software through inheriting the properties of existing software;
- Close correspondence between real world objects and objects in the software.

Whilst using an object-oriented approach for the development of simulation models offers these advantages over existing techniques there are still drawbacks. These drawbacks are associated with understanding the object-oriented software. There are two aspects to understanding how to use an object-oriented simulation library. The first

is the need to be familiar with the language used, whilst the second aspect is the need to understand the way class libraries are structured (Ball & Love, 1993).

5.3.2. Programming skills required with libraries

Object-oriented libraries tend to be implemented in high level programming languages such as Objective C (for example, Glassey & Adiga, 1990) and Smalltalk (for example, King & Fisher, 1986 and Bhuskute et al., 1992), rather than specialist simulation languages. The class libraries provide some of the functionality for a model (for example, the behaviour of machines and operators) but software must be written around these classes to create usable models (for example, the interaction between machines and operators, the loading and saving of the model and the user-interface). Hence, like simulation languages, programming experience is needed to create a model using a library.

The user of the class library must also understand the style of its implementation. Knowledge will be required of which classes exist, what their behaviour is and how they interact with other classes. For example, whether a machine uses an operator or vice versa. A learning curve will be experienced before using a library (Najmi & Stein, 1989) which, according to Glassey & Adiga (1990) can be considerable:

> *"Designing simulations is quite simple, but we have learned that one must also spend considerable time in knowing the objects in the library and what they are expected to do."*

It can be concluded that these libraries have been developed by programmers for programmers. Ideally a route should be sought such that libraries are developed by programmers for users.

5.3.3. Programmers' view

Shewchuk & Chang (1991) believe that more authentic representation will result in easier to use object classes, though such an approach cannot be seen in every class hierarchy reported in the literature. King & Fisher (1986) describe:

> *"When a part arrived for processing, the simulation finds the first operation in its process plan and attempts to acquire that machine as a resource ..."*

whereas a more natural representation would be for an operator to query the part's route card or a material handling system (by virtue of the part's position) to select an available machine. Also King & Fisher's description of the 'simulation' finding the first operation has no equivalence in the real world. This situation can also be found in the simulation libraries described by Bhuskute et al. (1992). In describing their simulation libraries Mize & Pratt (1991) and Adiga (1989) also imply that operators are resources to be used by machines rather than the more natural view that operators run the machines.

Failure to adopt a more natural view of the classes and their interaction could limit the application of the libraries. For example, in the previous case modelling various types of Nagare or 'U' shaped cells (Black, 1991; Lucas Engineering & Systems Ltd., 1989) would prove difficult with the 'machine using operators' approach. Also placing control logic in material objects may result in difficulties in modelling different material control systems. For example, batches can be selected from storage areas based on a number of rules: use a schedule, first in first out, earliest due date, shortest processing time, etc. If the selection has to be made by the material then the use of a schedule or the earliest due date rule will be difficult. Also modelling Kanban will be difficult since the configuration may involve the *absence* of parts as a trigger for production. Since control systems have a key influence on the dynamic performance of manufacturing systems then valid models of the control systems are important.

It would appear therefore that not only does the use of simulation libraries require programming skills but there is the need to adopt a more traditional programming view of the problem domain. Failure to employ an object-oriented view could result in

software that is difficult to understand and modify. As demonstrated above the underlying mechanisms can show through in the user-interface resulting in software that could be difficult to learn and use. It is possible that the authors of such libraries merely used object-oriented programming techniques to ease the task of the developer whilst ignoring their potential to improve the ease of use with respect to the manufacturing engineer.

5.3.4. Limitations of current object-oriented libraries

Official statistics for European Industry (and selected other countries including Japan and USA) show that the level of penetration of Flexible Manufacturing Systems (FMS) and robots is low and that, given the long life of traditional machine tools, the level of adoption of automation will be gradual (Commission of European Communities, 1991). Mansfield (1993), in a survey of firms in Japan, Europe and USA, also shows that the use of FMS represents a very small section of the manufacturing industry. This would suggest that the human element in manufacturing systems will continue to be extremely significant.

Glassey and Adiga (1990) present a library of manufacturing classes called Berkeley Library of Objects for Control and Simulation of Manufacturing (BLOCS/M). Such a name is misleading since the actual library concentrates on classes for automated production. Little attention is paid towards systems in which an operator plays a key part. Despite the significance, in the class hierarchy developed by Shewchuk & Chang (1991) no mention is made of operators even though the classes are designed specifically for manufacturing systems. Bhuskute et al. (1992) describe their modelling environment for modelling discrete part manufacturing systems but no mention is made of operators. This therefore restricts modelling to fully automated machines and will result in problems in the modelling of manually operated lift trucks as proposed in an earlier paper (Basnet et al., 1990).

Whilst minor deficiencies in an object's functionality can be handled by overwriting the offending methods (or adding new ones) omission of a whole class (or classes) is much more difficult to deal with. King & Fisher (1986), Shewchuk & Chang (1991) and Bhuskute et al. (1992) describe the logic governing the manufacturing of parts embedded

in the part or 'system' objects. To introduce a new class and move logic across to that new class would be extremely difficult and error prone. An example could be the introduction of an operator class and moving logic from the batch class to the operator class.

Alternatively the new classes could be manipulated to fit into the existing world view. For example, in the process of adding an operator class the operator objects could become resources to be acquired and released by parts. This is undesirable for many reasons:

- The behaviour of the new class would not match the engineers' view;
- The addition would increase the complexity of the software, possibly preventing the addition of further classes;
- The implementation of the new class may be such that the dynamics of that class cannot be modelled fully or correctly. For example, modelling an operator as a resource fails to recognise that operators make decisions and that operators have other prioritised duties such as setting, changeovers, transportation, etc. This may affect the validity of the model.

The functional limitations of current object-oriented simulation libraries could preclude their application even if the user has the necessary programming and simulation skills to make the necessary changes. Clearly there is a mismatch between the library view and the real world view. It is possible that this mismatch has occurred because the programming languages used do not enforce the real world view. This would suggest that the use of programming techniques alone is insufficient. The use of object-oriented design methods could be used to overcome such limitations; object-oriented design methods have the potential to adopt a real world view and to guide the implementation process.

5.4. Using object-oriented techniques to reflect reality

The previous sections discussed the ability of object-oriented techniques to act as a means by which concepts in reality can be replicated in software. A number of authors

have discussed the ability of object-oriented techniques to support this (Adiga, 1989; Najmi & Stein, 1989; Taylor, 1993).

In the process of manufacturing system design, the manufacturing system can be viewed as a collection of resources that can be selected and configured to best meet the performance required (for example, see Parnaby (1986) and Afzulpurkar et al. (1993)) Also the manufacturing system could be considered as a library of objects that are selected and configured to best meet the performance required. There is therefore a natural correspondence between the view that is commonly adopted for manufacturing systems and a view that could be created using object-oriented simulation libraries (Nof, 1994). Simulation libraries could therefore be used to build simulation models that match the process of manufacturing system design.

The manufacturing system design process can be considered hierarchically, progressing from aggregate to detailed designs (Parnaby, 1986; Love & Bridge, 1988). Manufacturing systems can also be considered at different levels of detail in an operational context and could therefore be viewed as collection of objects of various levels of detail. The object-oriented approach could be used to create objects of various levels of detail (King & Fisher, 1986) and is therefore well suited to providing different levels of modelling detail. The levels could match the levels as seen by manufacturing engineers.

It is worth noting here that the methodologies described by Parnaby (1986) and Afzulpurkar et al. (1993) consider the detailed design of material and personnel control and information systems. Elements of such systems have been shown to be absent from the software described above.

Conceptually therefore object-oriented techniques are well suited to the modelling of manufacturing systems. Object-oriented techniques have the ability to replicate the view held by manufacturing engineers both in terms of how objects interact and the level of detail held by the objects. A simulation tool using objects that match those used by manufacturing engineers would be conceptually easy to use. Such a tool would also be

able to support models that match requirements. It has been shown, however, that current tools and simulation libraries are deficient in one way or another.

6. Importance of object-oriented analysis and design

> *"No matter what the particular application, the problem space is rooted somewhere in the real world, and the solution space is implemented by a combination of software and hardware."*
>
> Booch (1983)

Earlier the application of object-oriented techniques to the construction of simulation software was discussed. It was demonstrated that there were a number of aspects of object-oriented techniques that were well suited to simulation modelling. These aspects included the ability to capture concepts and terminology found in reality and the ability to match the levels of abstraction employed during the manufacturing system design process. The discussion demonstrated that current applications of object-oriented techniques to simulation were limited in a number of respects. This chapter examines the role of object-oriented analysis and design in software development, in particular in the development of manufacturing simulators.

6.1. Use of object-oriented design and analysis

6.1.1. Design

The previous chapter examined the implementation of object-oriented simulation libraries. The examples chosen were typical of simulation libraries described in the current literature on manufacturing simulation. Most, if not all, libraries have deficiencies in their implementation such as missing classes or classes that have been inappropriately implemented. As a result difficulty may arise in the use of particular classes. From the manufacturing engineers' point of view the classes may not be able to provide the desired representation and therefore the desired type of results. From the developers' point of view the classes may not be reusable in terms of inheritance or inclusion in the software. Also the implementation of the classes may prevent the desired implementation of new related classes. For example, many simulation systems use a world view in which batches contain logic for acquiring and using resources such as machines and operators. In situations such as this difficulties could arise if scheduling needs to be carried out where the operator selects a batch from a storage area rather than always using a first in first out rule.

The previous chapter also examined the role of the programming language in the software development process. It was argued that programming languages are just the medium through which good, or bad, designs are implemented. Examination of current object-oriented simulation libraries would suggest that the design process has either been poorly implemented or driven by requirements other than those of engineers.

The development of easy to use simulation software can be assisted by including concepts and terminology used in the real world. The engineer would therefore be able to easily understand the potential behaviour and interaction of the functionality supported by the simulation tool. Object-oriented design (OOD) methodologies have the potential to capture concepts and terminology from the real world. The design methodologies provide a notation by which such objects can be documented. The documentation can then be used to guide the process of implementing the software.

6.1.2. Analysis

In a discussion of object-oriented design methods, Ormsby (1991) states:

> *"Perhaps paradoxically, design methods cannot design for us."*

Ormsby also observes that many design methods are little more than documentation methods. The subsequent growth of object-oriented analysis (OOA) methods has perhaps gone some way to filling this gap. Object-oriented analysis methods are concerned with abstracting user requirements into objects to be used in a design methodology. Analysis methods should create an understanding of the problem but should not become involved with implementation issues (Rumbaugh et al., 1991). The use of an object-oriented analysis method therefore provides an initial step in developing software.

The distinction between object-oriented analysis and design methods is not a clear one (Graham, 1994). Both methods typically carry elements of analysis and synthesis. For example, the design methods described by Ormsby (1991) include elements of analysis. The application of analysis and design methods will not be distinct either and will

typically proceed iteratively (Booch, 1991). Booch illustrates this point well, see Figure 6.1.

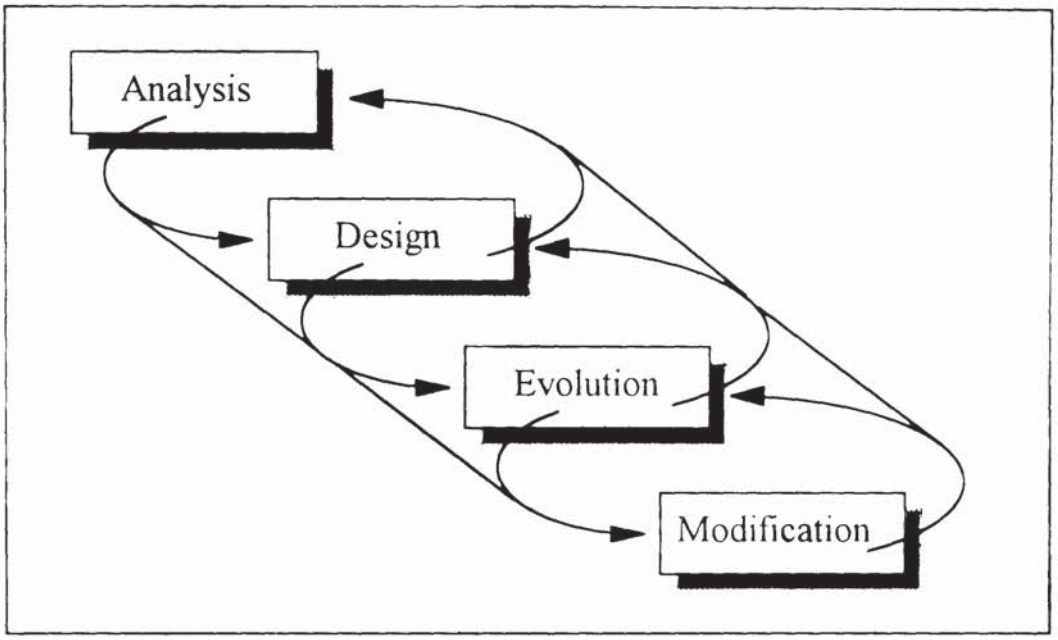


Figure 6.1. Object-Oriented Design in the Software Development Life Cycle (from Booch, 1990)

Object-oriented analysis is important for relating the software design to the application area. Like object-oriented programming and design, object-oriented analysis has the potential to replicate concepts and terminology found in reality:

> *"The purpose of object-oriented analysis is to model the real-world system so that it can be understood."*
>
> Rumbaugh et al. (1991)

6.1.3. Combining analysis and design

Object-oriented analysis (OOA) methods could be used to capture real world concepts and terminology. Through the process of object-oriented design (OOD) such concepts and terminology could be implemented using an object-oriented programming (OOP) language. The combined process therefore has the potential to guide the development of software that users will find easy to use.

The combined process also has the potential to guide the development of the software that is easy to use with respect to the software developer. Careful analysis, design and

implementation could enable the development of software that is robust to changes. Developers could then modify and extend such software straightforwardly.

6.2. Matching the manufacturing engineers' view

6.2.1. Manufacturing engineers' requirements

There are numerous different types of software used in industry. Examples include spreadsheets, databases, word processors, computer-aided drawing packages, statistical analysis packages, material requirements planning systems, etc. Different packages will be used in different ways. For example, a database may be used by numerous individuals to retrieve specific information whereas a material requirements planning system may be used by a few individuals and the output passed on to others. It was earlier argued that in the case of manufacturing simulation the most appropriate users would be manufacturing engineers:

- engineers will be the users of the results;
- engineers will have the detailed knowledge of the manufacturing system;
- engineers will benefit from the process of model building;
- engineers will be able to experiment with different designs more readily.

The developers of any software should account for the requirements of the ultimate users of the software. They should be able to implement designs that match the user expectations and enable users to utilise the software to their benefit. In the case of simulation, therefore, the software should be constructed to match the needs of manufacturing engineers.

Object-oriented analysis, design and programming can be applied during the process of software development. These methods could be applied to detailed aspects of the software such as the way in which data are loaded and saved or the structure of the class hierarchy. Application of analysis, design and programming techniques here could benefit developers; modification and extension of the software could be easier and less error prone. Such benefits are, however, closely related to the developers' activities and have little or no effect on the users.

Analysis and design could however be extended to the benefit of the users, that is manufacturing engineers. The analysis and design could be utilised to ensure that engineers' requirements are met. There are two key areas in which engineers could benefit from the application of these methods. Firstly the functionality offered by the simulator could be developed to match the engineers' view of the real world. For example, a manufacturing system consists of machines, operators, trucks, routings, etc. The natural correspondence between the simulator and reality has potential to reduce the learning curve and the time taken to build and evaluate models. Secondly the appearance of the user-interface could be tailored to either engineers' requirements or follow a style typically adopted for software. For example, the style of user-interface of many Microsoft Windows packages is very similar.

6.2.2. Reconciling conflicts

There is a possible conflict that emerges between development and use. Object-oriented analysis and design methods have to meet the needs of the ultimate users of the software whilst at the same time meet the immediate needs of the developers. Engineers require software that is easy to use and matches requirements whilst developers are concerned with developing software quickly and efficiently. This situation applies equally to simulation software as any other software. It could be argued that user requirements take priority; if software is not used or does not meet user requirements then the developers have failed in their task. The use of methods that promote programming convenience at the expense of the ease of use should be avoided where possible.

If analysis and design methods focus on the needs of the users then this need not be in conflict with the developers. Firstly, adopting this approach will ensure that users' needs are met. Secondly, the focus on real world concepts and terminology can be used to guide and standardise the development process. In this way developers new to a software package will be able to use a view of the real world as a guide to understanding the underlying architecture. For example, a developer would be able to judge the relationship between operators and machines. Similarly, if a material handling mechanism had to be altered or added the developer would use a view of the real world

to guide the process of locating the relevant position in the software, in this case perhaps the operator.

The process of object-oriented analysis, design and programming should be guided by user requirements. Use of this approach has the potential to benefit the users and the developers. It should be possible to develop software that is easy to use for both the users and the developers.

6.3. Object-oriented simulator development

6.3.1. Object-oriented and simulator concepts combined

It should be apparent from the discussion so far that there are a number of potential benefits to be gained from the application of object-oriented techniques. Some of the key properties required for simulation tools can be supported through the use of object-oriented concepts. Simulation tools should provide functionality that reflects concepts found in reality. Such tools should be able to provide a sufficient range of functionality to enable useful simulation models to be built. Object-oriented techniques can be used as a means by which real world concepts and terminology can be abstracted into software objects. The use of object-oriented techniques also provides mechanisms for expansion hence functionality of object-oriented simulation software can be increased as necessary.

The use of current object-oriented libraries alone as a means of building simulation models, however, is undesirable. The use of libraries to build simulation models requires a wide range of skills:

- general programming skills;
- object-oriented skills;
- simulation software construction skills;
- application (that is, manufacturing) specific skills and experience.

Time to apply libraries could be lengthy since it involves

- learning general programming skills,

- understanding object-oriented principles;
- understanding simulation theory;
- model building;
- model testing and debugging;
- model verification;
- model validation.

Ideally the skills and time requirement should be reduced. Reduction can be achieved by simplifying or removing many of the above stages.

The concepts of object-oriented libraries and simulators could be combined into a single simulation tool. The object-oriented techniques can enable concepts of the real world to be captured in software whilst simulators remove the need to develop and apply programming and detailed simulation skills. Such a simulation tool would be inherently easy to use and it would be possible to extend its functionality to cover additional application areas with minimal effort (Ball & Love, 1993).

6.3.2. Object-oriented simulator architecture

Object-oriented techniques can be applied to application classes. The term application classes refers to the classes that are specific to a particular domain or application. For example, for manufacturing system simulation the application classes may include 'machine', 'truck', 'part' and 'routing'. Many such manufacturing system simulation libraries have been developed (see King & Fisher, 1986; Basnet, et al., 1990; Glassey & Adiga, 1990; Shewchuk & Chang, 1991).

However, concentration of the application of object-oriented techniques on the application classes alone will only result in reusable application classes at best. A significant element of simulation software is unrelated to the application area. Simulation software must support other mechanisms such as the simulation clock, event list and executive (again see King & Fisher, 1986, Basnet, et al. 1990; Glassey & Adiga, 1990; Shewchuk & Chang, 1991).

There are other parts to simulation software that are critical to successful operation and use other than the simulation clock, event list and simulation executive. Simulation software must be able to support the creation, editing and removal of objects. The processes of editing and removing objects must also be accounted for in the objects that remain. Simulation software must be capable of generating, storing and retrieving results. The use of animation is now standard within simulation software and hence provision for creating, managing and updating graphical display is necessary. All of these are requirements irrespective of the application; management of model data, results and graphical displays is required irrespective of whether the simulation application is concerned with manufacturing, hospitals or airports.

Literature on object-oriented simulation libraries concentrates mainly on the application classes and classes that are concerned with the simulation mechanism. Little discussion is given to the data management, results, graphical displays and user-interface. King & Fisher (1986) and Basnet, et al. (1990) do discuss some of these aspects. In both cases, however, greater emphasis is placed on the application classes and few details are given on these other areas.

The use of analysis, design and programming techniques on the application specific element of simulation software is no more important than other areas. These other areas include the user-interface, the collection and analysis of results, the simulation mechanisms, the graphical displays and the object management mechanisms. The style of implementation of these support areas will have a significant impact on the application of the software. Poor implementation of these areas may result in a tool that is difficult to use or difficult to modify. Again, object-oriented techniques only have the potential to produce easy to use software, they do not enforce its production.

The ability to reuse the user-interface as well as the simulation classes allows reuse of the entire simulation system without the need for programming. The benefits of such a system are two-fold: the user of the simulation system would be able to create models without the need for programming and an expert could extend the system without the

need to re-create the user-interface. Whilst a few object-oriented simulators exist, none have demonstrated ease of expansion.

Strandhagen (1987) and later Borgen & Strandhagen (1990) describe object-oriented manufacturing simulators with user interfaces that use terminology natural to the user. Neither paper discusses the ability to extend the simulation tools to include new functionality. Whilst using an object-oriented approach may lead to a better user interface and allow functionality to be added easily it is not true for all cases. For example, the manufacturing simulator ATOMS (Bridge, 1990) is not object-oriented in its construction and yet its interface uses terminology closely related to the problem domain. Also, as discussed earlier, the use of object-oriented programming languages and object-oriented styles of programming will not necessarily lead to software that is easy to modify.

Collins & Watson (1993) described the simulation system Arena as object-based and extendible. In this case the term objects refers to circles, lines and text. Modules are used to represent modelling constructs such as queues, servers, resources, etc. New modules can be added by grouping together more abstract modules. For example, a server module is made up of the base modules queue, seize, delay and release. In this approach the ability to describe real world objects is governed by the base modules. This will result in poor abstractions of real world objects and a limited range of functionality.

Harrell & Tumay (1991) imply that the simulator, ProModel, is object-oriented in its construction. There is also a weak implication that it is possible to add new features to ProModel. What these new features are is not described, nor have any additions been documented or demonstrated. Again the use of object-oriented programming languages, in this case C++, does not necessarily result in the production of object-oriented software or software that is easy to modify and extend. Whilst object-oriented languages enable object-oriented designs to be implemented, if software is not designed to be easily extended then it is likely that extension will be difficult.

6.3.3. Potential of an object-oriented simulator

From the preceding discussion the potential benefits of an object-oriented simulator should be apparent. An object-oriented simulator would be able to overcome the compromise between ease of use and range of functionality typically associated with simulators. Such a simulator would be able to combine ease of use with potential range of functionality. The ease of use would be provided by the data driven concept of simulators and the object-oriented concept of authentic representation of the real world. Without conflict, object-oriented techniques would also provide the potential range of functionality by providing mechanisms by which the simulator could be extended.

Currently there are few simulators that employ object-oriented concepts. Those simulators that do have limitations in the functionality offered and/or have not been shown to allow the addition of new functionality. There is therefore an opportunity to employ object-oriented techniques in the development of a simulator that more closely meets the needs of manufacturing engineers.

The development of such a simulator would focus on two key areas. Firstly object-oriented techniques provide a means by which superior modelling functionality could be provided. The functionality would closely match concepts and terminology used by manufacturing engineers. Such functionality has the potential to improve simulator ease of use. Secondly object-oriented techniques provide a means by which the software could be developed. Robust and easy modification and addition could be made inherent properties. As new needs arose the simulator could be modified to meet those needs. See Figure 6.2.

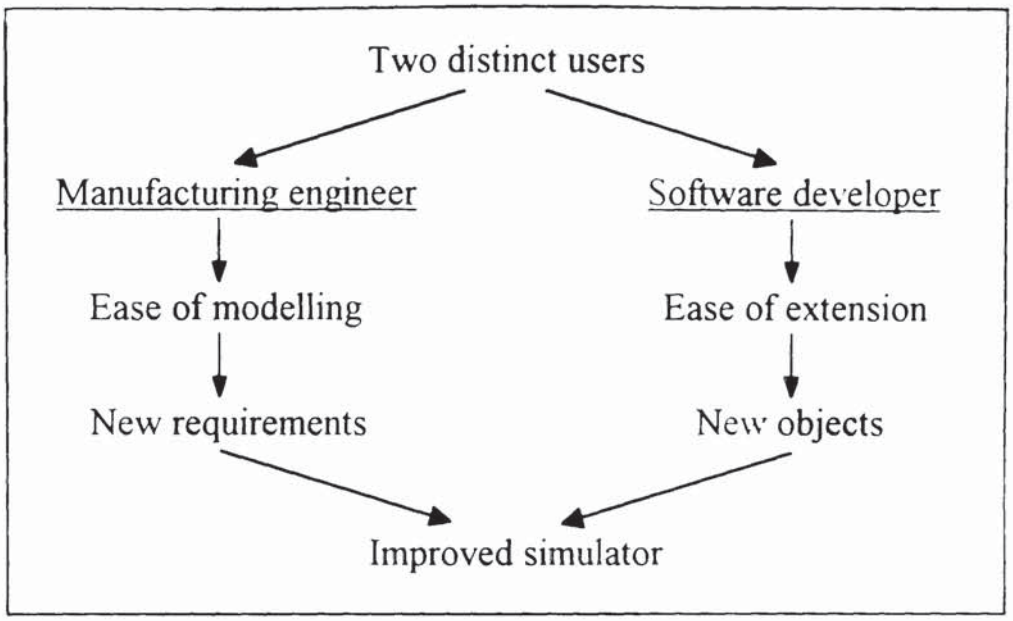


Figure 6.2. Two distinct users of an object-oriented manufacturing simulator

7. Design of the Advanced Factory Simulator

> *"Theory is a wonderful thing, but from the perspective of the practicing engineer, the most elegant theory ever devised is entirely useless if it does not help us build systems for the real world."*
>
> Booch (1991)

The potential contribution of simulation techniques to the design and analysis of manufacturing systems has been established. A review of current simulation tools has revealed deficiencies that act as a barrier to their use by manufacturing engineers. It has been argued that use of object-oriented techniques could be employed to overcome many of the deficiencies. This chapter will detail how object-oriented techniques can be utilised. The benefits that result will be described.

7.1. The approach to the design of the Advanced Factory Simulator

This chapter will discuss the macro aspects of the design of what has been called the Advanced Factory Simulator (AFS) (Ball & Love, 1993). The next chapter will discuss some of the detailed or micro aspects. This and subsequent chapters will contain elements of analysis, design and implementation. Combining these elements was carried out to ensure that the subsequent implementation was both possible and worthwhile.

The methods and notation used follow those of Booch (see Booch, 1991). There are a number of reasons why Booch's methodology has been chosen. The methodology contains elements of analysis, design and implementation. The methodology recognises the pragmatics of design and therefore recognises its iterative nature. There are also elements to the methodology that are applicable to simulation software design, for example state transition diagrams. Finally Booch's methodology is well recognised (Graham, 1994).

This chapter will provide a number of examples to illustrate the concepts employed for the analysis and design, some of which could be seen as implementation aspects of the software development process. It is important that these examples are seen as illustrations of the concepts and not the actual application.

7.2. A manufacturing system class library

7.2.1. Identification of classes

Earlier it was argued that the view of the real world employed by simulation software has an impact on ease of use. The view adopted will affect the ease by which engineers can learn to use the software and also the ease by which data from the real world can be used to create a model. Since object-oriented techniques enable a real world view to be represented in software then such techniques should be used to develop the functionality. The class library that results will contain terminology similar, if not identical to that used in the real world. A selection of classes is shown in Figure 7.1.

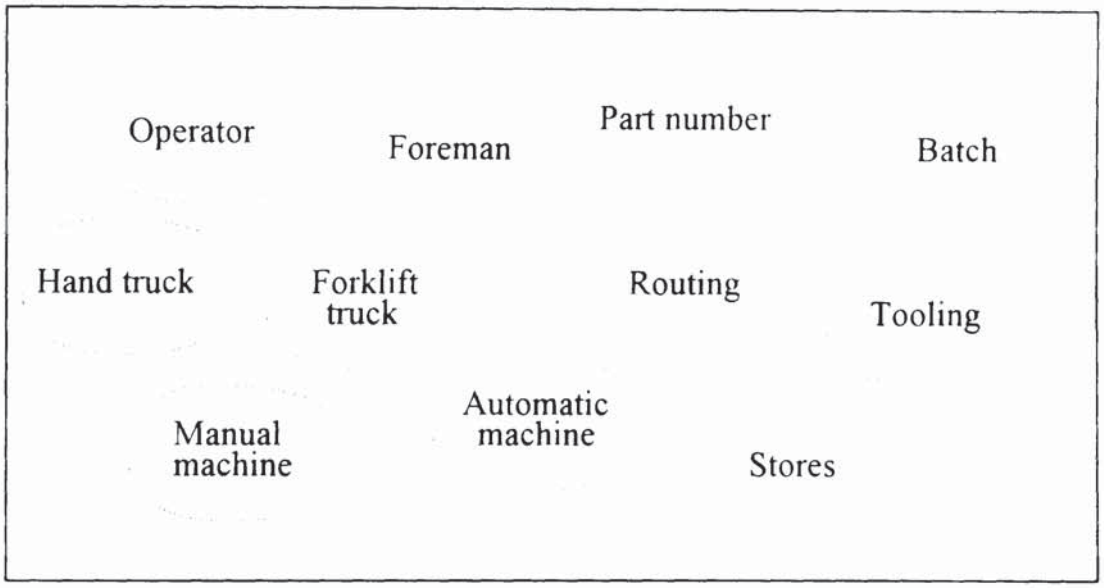


Figure 7.1. Collection of manufacturing system classes

7.2.2. Class behaviour

Once possible classes have been identified their internal behaviour must be considered. For example, a machine must be able to convert parts and a truck must be able to transport items. During this process the list of classes will alter as new ones are added, others merged and some redefined. Some behaviour will be easy to identify whilst some may require more thought and may need to be tested alongside existing classes for consistency. For example, the shift pattern that operators work to may be a mechanism internal to the operators or may be considered a factory policy that dictates their activities.

Where possible internal behaviour of classes should be encapsulated in sub-classes. Using this approach enables inappropriately placed mechanisms to be moved relatively easily. For example, a shift pattern could be a standalone class. An operator could use the shift pattern class to dictate when to work and not work. Since the shift pattern class is independent of the operator another class, perhaps an abstract cell class, could also use it. The shift pattern mechanisms can therefore be reused.

Other aspects of a class are the data items required. For example, an operator could have size, position, efficiency and performance data items. Classes may also have state transition networks. The networks determine the sequence of activities the class can undertaken. For example, an operator may change states from 'off shift' to 'idle' and then, once work has been found, to 'working'. Documentation of these concepts are illustrated in Figure 7.2.

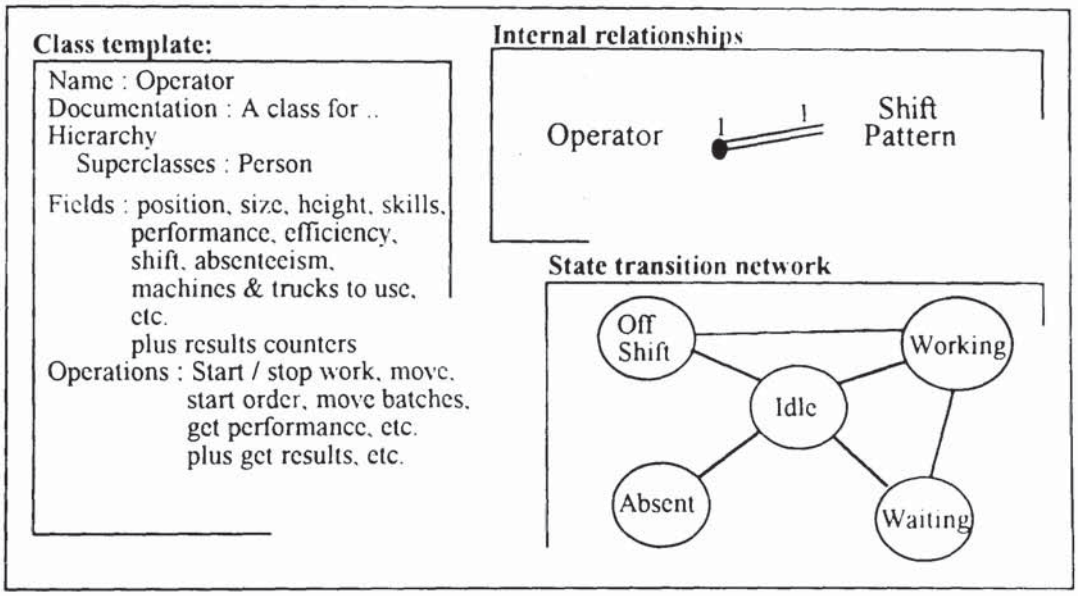


Figure 7.2. Describing internal behaviour

7.2.3. Establishing relationships

Closely related to, and in some ways inseparable from, the internal behaviour of a class is the relationship between classes. Examples of classes relationships are how an operator and machine interact and how a part and routing are linked. Not only must the terminology be taken from the real world but also the concepts. The interaction between the classes described has been ignored up to this point. How classes interact is, however, an important consideration as this will affect how easily the user is able to

understand the model building process. The use of concepts from reality will enable manufacturing engineers to understand the simulation tool more clearly. At this point the use of manufacturing principles to guide the design can be clearly beneficial. Earlier discussion of object-oriented software showed many instances (King & Fisher, 1986; Glassey & Adiga, 1990; Shewchuk & Chang, 1991; Bhuskute et al., 1992) in which parts or other abstract objects used machines and operators as resources. In the approach used in the Advanced Factory Simulator (AFS) operators run machines and use trucks to transport batches, see Figure 7.3. The notation used in the figure is taken from Booch (1991). The figure illustrates 'uses' relationships. For example, one operator will use many hand trucks whilst a part number only uses one routing.

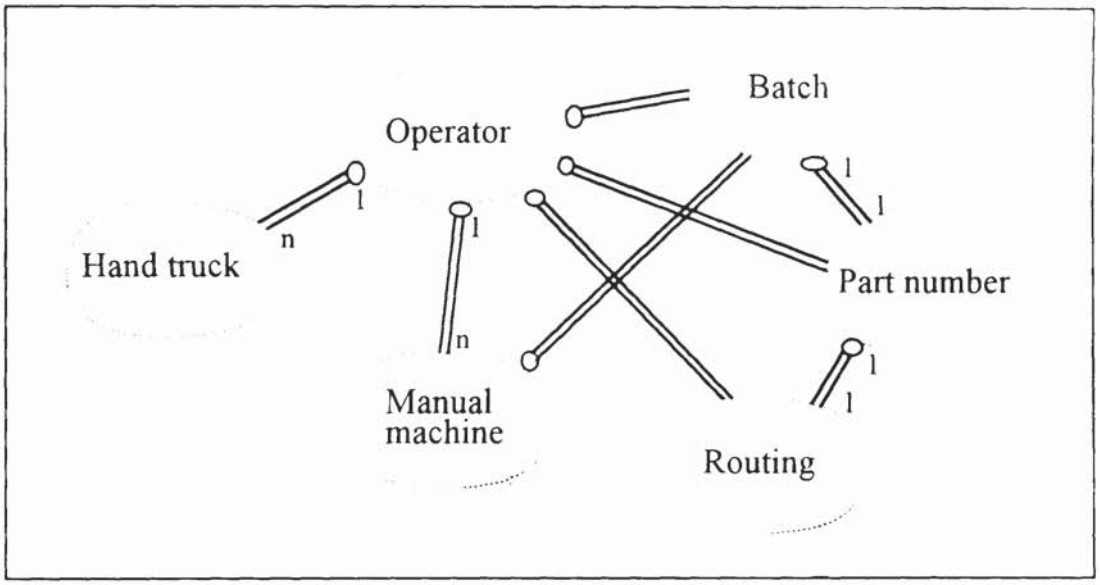


Figure 7.3. Authentic representation of interaction (notation: Booch, 1991)

The use of real world terminology and interaction will enable users of the software to build representative models. Note that in Figure 7.3 there is no relationship between the hand truck and the manual machine and the batch has no knowledge of what a machine is. The user is able to create objects of classes that correspond to objects in the real world. Not all potential users would envisage the manufacturing system at the level of detail considered. A discussion of multiple levels of modelling will be left until later.

7.2.4. Loose coupling and messaging

Whilst the relationship between classes was described in the previous section few details were given. For example, the principle of operators using machines was proposed but

the detail of how this would be achieved was not described. During the simulation operators will load machines, material handlers will deliver goods to stores, supervisors will issue instructions, etc. A mechanism must exist to provide these interactions.

One of the concepts of object-oriented techniques is loose coupling. As the name suggests, loose coupling refers to the distant nature by which classes are linked. If classes are loosely coupled then they possess minimal knowledge of one another and yet at the same time are able to interact. For example, an operator may use a machine. In a tightly coupled situation the operator would know exactly how to operate a particular machine. In a loosely coupled situation the operator would know it could utilise an object that gives the appearance of a machine and be able to instruct it to load a part, run, unload, etc.

Loose coupling is important because it means that objects can interact with one another without knowing what particular type of object they are interacting with. Hence an operator could be operating a milling machine, a semi-automate lathe or a huge transfer machine. No modification would have to be made to the operator to achieve this. Similarly a cell leader could send a works order instruction to an operator or a scheduling board. The object-oriented approach supports this interaction through messages, see Figure 7.4.

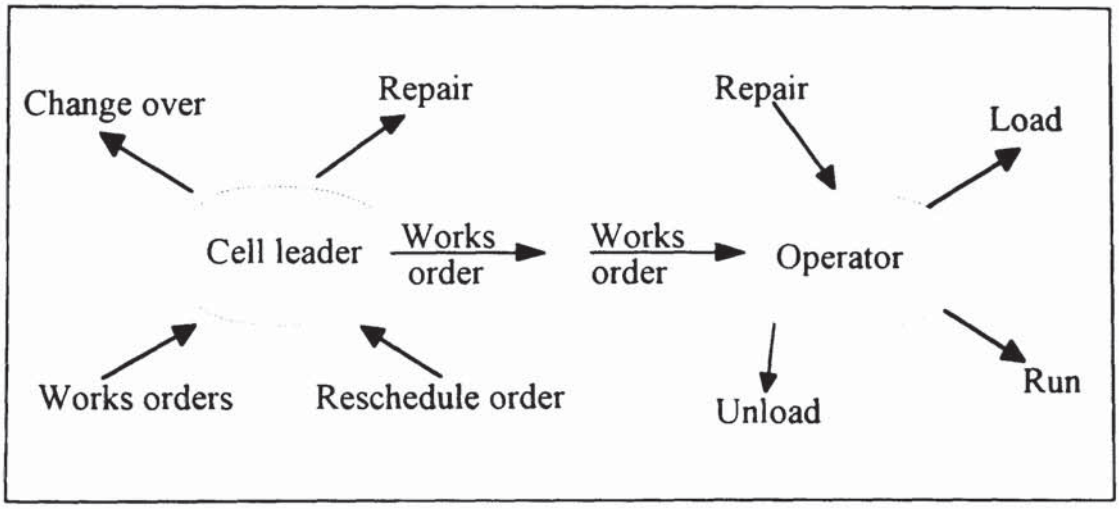


Figure 7.4. Possible messages between cell leader and operator

Using messaging will allow a certain amount of independence between objects, new classes can be added and modified with little or no effect on existing classes. The same

will not be true for the messaging mechanism itself unless it is designed using object-oriented principles. A class hierarchy should therefore be developed for the messages. Each message will be an object. New message types can be added and existing ones modified without adversely affecting the functionality of objects using them. Hence the concept of loose coupling relies on the communicating functionality to be object-oriented as well as on the communication mechanism itself. For example, the functionality could be a machine object and an operator object and the operator will communicate with the machine using message objects. The messaging mechanism should be the same for all functionality and so any object can communicate with any other object using messages.

Messages allow objects to interact without prior knowledge of one another. Messaging is therefore a key concept to use in the design of the architecture. The use of messaging has the potential to ease the process of modifying and extending the functionality. Messaging mechanisms will enable changes to functionality to be carried out with minimal modification to the remaining software.

7.2.5. Simulation results

The use of object-oriented techniques can be extended to the collection of results. Results can be generated and interpreted by classes. For example, the results from the machine class may include down time and changeover time whilst those from the batch class may include number of defects and lead time. Generic mechanisms for results collection and display must be used. Using generic mechanisms removes the requirement for their modification each time a new class capable of generating results is developed.

7.3. Potential of class libraries

The development of functionality using the above approach will not just benefit the user. The abstraction of real world objects into software means that the real world can be used as a reference. Hence developers can use real world objects as a basis for understanding software objects. The use of class libraries can therefore enable new developers to more easily understand the existing software.

The use of the concept of encapsulation (see Booch, 1991) will enable the implementation of the classes to be hidden. Mechanisms and data structures can therefore be modified with little effect on other classes. For example, the loading mechanism of a machine or the shift patterns of operators could be changed without change to other classes. Well encapsulated classes will therefore bring benefits.

The use of inheritance (see Booch, 1991) will bring both immediate and subsequent benefits. Inheritance will enable software to be reused and reduce modifications necessary as new classes are added. For example, both the manual machine class and the automatic machine class are types of machine. Hence an abstract machine class could be developed and then inherited by the manual and automatic machine classes. The operator class could use methods in the abstract machine class to operate both the automatic and manual machines. If a new class, perhaps an indexing machine, is inherited from the abstract machine class the operator class software would not need modification. See Figure 7.5.

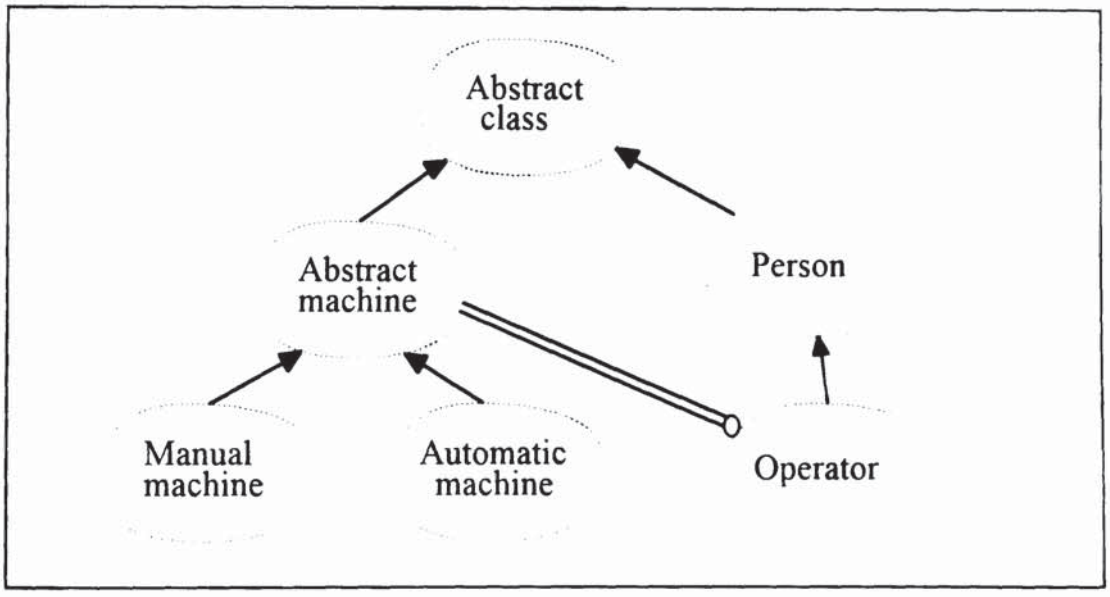


Figure 7.5. Inheritance relationships in a manufacturing class hierarchy

The nature of the interaction of the classes described follows closely that of the real world. Approaching the design of the simulator in this way will bring benefits to the engineer and developer. The engineer will be able to build representative models and in turn be able to obtain meaningful results. For the developer the opportunity to extend the functionality offered has been preserved. For example new machines can be added,

operators can be modified to undertake additional duties, supervisor and management classes can be added. These additions can take place within the existing framework and modification of existing software would either be minimal or non-existent. This approach is in contrast to the approaches used to develop libraries in which Automatic Guided Vehicles (AGVs) can load and unload themselves (for example, Bhuskute, et al., 1992), parts have intelligence (Shewchuk & Chang, 1991) and material control rules are embedded in the parts, machines or other abstract classes (for example, King & Fisher, 1986; Adiga, 1989).

7.4. Simulation, animation, results and user-interface

Object-oriented techniques combine to provide developers with a number of powerful software building mechanisms. Use of the concept of inheritance was described above, new classes can be added by inheriting abstract classes and adding software to account for any differences. Hence object-oriented techniques enable programmers to concentrate on the differences and avoid any duplication. When adding new functionality the programmer should ideally only need to concentrate on adding new software directly related to that new functionality.

When the manufacturing system class library is considered in isolation then the ease by which new classes can be added is clear; a new class inherits a more basic class. When the whole of the simulation software is considered the situation is different. A simulation tool does not just consist of application classes alone. Supporting software is also required to manage the simulation, the animation, the results and the user-interface. Ideally, when a new class is added the developer should not need to understand or modify the supporting software. If modification of the supporting software is required then a number of serious drawbacks are encountered:

- the developer needs to understand the architecture of the supporting software;
- the developer will need to spend time modifying the supporting software;
- additional software testing and debugging will be required;
- the development time and skill required will be much increased;
- modification of the support area implies increase in software complexity

Hence a supporting architecture that is well designed will not require extensive knowledge and modification when new functionality is added (the use of compiled, rather than interpreted, software means that for any additions modifications of existing code is essential).

The supporting architecture comprises the graphical displays, user-interface, results mechanisms, simulation mechanisms and the object management mechanisms. Since these are independent of the application area, in this case manufacturing, then they should not require modification when new manufacturing classes are added. Also the supporting architecture should be robust to changes. For example, it should be possible to change parts of the displays or manufacturing classes without changing other parts of the software. This can be achieved via abstract classes, see Figure 7.6.

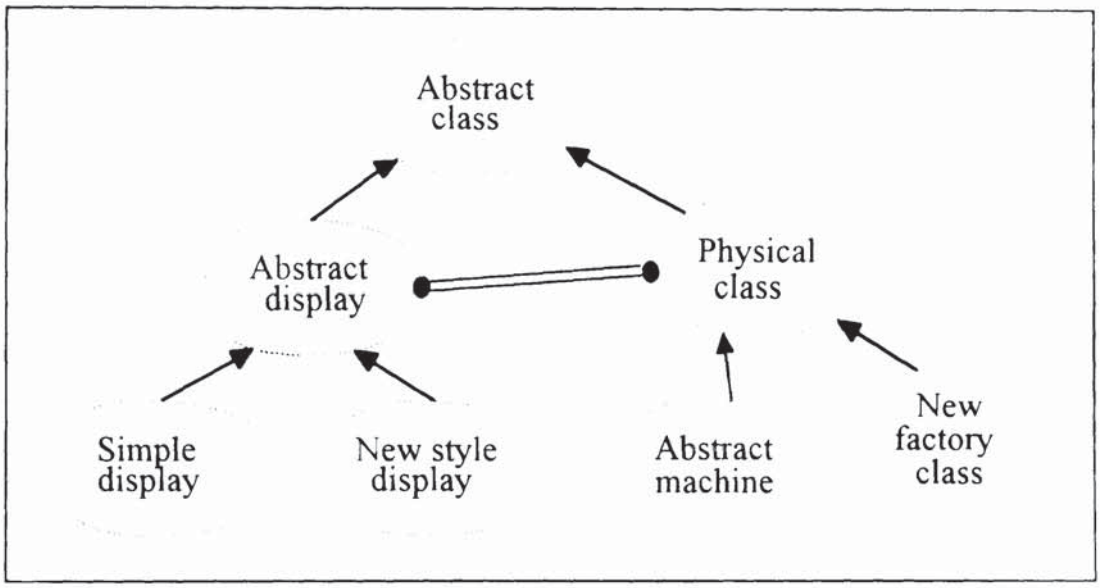


Figure 7.6. Possible display and physical class architecture

Figure 7.6 shows that new displays and new application classes can be added. The relationships between the application classes and display classes have been implemented at an abstract level. Since there is no direct relationship between the actual displays and the application classes, additions to either side of the hierarchy will not require modification of the other.

7.5. The role of class managers

The concept of a class has been described. A particular class will contain all the code associated with that class, for example a machine class will contain all the data structures and mechanisms associated with the operation of a machine. To be of use it must be possible to create objects of that class. Since objects cannot create themselves a separate mechanism must exist to create the objects. The danger is that such mechanisms may become dispersed throughout the simulator and give rise to in an increase in complexity of the software. This will result in software that is difficult to understand and maintain.

It is necessary to contain the mechanisms used for the creation, editing and destruction of the objects. Encapsulating these mechanisms in classes has the advantage of minimising the spread of complexity and therefore minimising the effect of any changes made. A separate hierarchy of management classes should therefore be developed to mirror the hierarchy of the simulation classes. Each simulation class will have a corresponding 'class manager' to manage the creation, editing, etc. of the objects. This concept is illustrated in Figure 7.7.

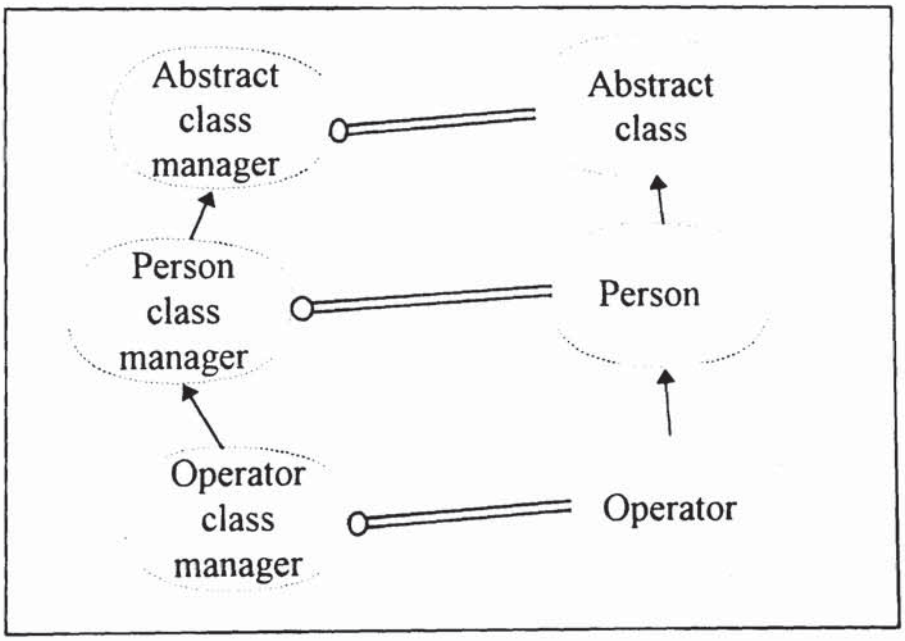


Figure 7.7. The class manager and class hierarchy.

Each class manager can cover a number of mechanisms that are not specific to particular objects of the class it is managing. As well as for creating and editing, class managers

would be responsible for loading data from text files, managing the load and save process and preparing the results data for presentation by the display mechanisms.

Introducing class managers into the architecture design allows a clean split between the simulation functionality and the data management routines. For example having machine objects for manufacturing system modelling and the associated machine class manager object for managing the load, save, editing, etc. An alternative mechanism would be a series of functions placed inside the same file as the simulation class. This approach fails to take advantage of object-oriented techniques and fails to contain complexity.

During simulation, objects would communicate directly using the messaging mechanism. For the purposes of editing, loading, saving, displaying results, etc. the class managers would be used. All the code specific to a type of functionality (e.g. a machine) should be held between the actual class and the corresponding class manager.

An object of each class manager class would be initialised and placed on a class manager list. All non-simulation related access to the objects would be via the appropriate class manager object. This allows an indirect relationship between the actual functionality and the supporting architecture described earlier. Hence to add a new class would require the development of the class, development of the class manager and the insertion of the class manager object onto the class manager list. The addition of new classes would therefore require minimal changes to the supporting architecture.

7.6. Multiple levels of simulation modelling

7.6.1. Matching the users' view

One of the recurrent themes in this thesis is the concept of matching the capabilities of simulation tools to potential users' needs. However, as discussed, personnel in different parts of a business will view it in different ways. In particular the manufacturing system can be viewed in different ways for either operational or design purposes. A simulation tool must therefore have provision for presenting different views.

In the case of operational analysis of the manufacturing system the view presented by the simulation tool could be real or illusory. For example, if a high level, abstract view was required then the simulation tool could either use functionality that modelled the manufacturing system at that level or could give the appearance of modelling at that level by aggregating the results from a more detailed model. The problem with this latter approach is that the model evaluation time would be high. In the case of design it is unlikely that data for detailed simulation would be available for all stages. A simulation tool should therefore present multiple levels of modelling detail by providing different functionality for different levels.

Current simulation tools tackle the problem of multiple levels of modelling inadequately. Generic simulators and simulation languages require the user to develop the functionality for the particular level. This requires time and skill to achieve. Domain specific simulators such as ATOMS (Bridge, 1990) and MAST (Lenz, 1989) enable a number of predetermined levels of modelling to be selected. Such simulators have a limited range of levels.

7.6.2. Object-oriented approach

The object-oriented approach has been proposed as a way of achieving multiple levels of modelling (see King & Fisher, 1986). Objects at a detailed level could be merged and represented by one aggregate object, see Figure 7.8. The aggregate object could be used to model the essential behaviour of the collection of objects. The aggregate object would require less data and would shorten the overall simulation run time.

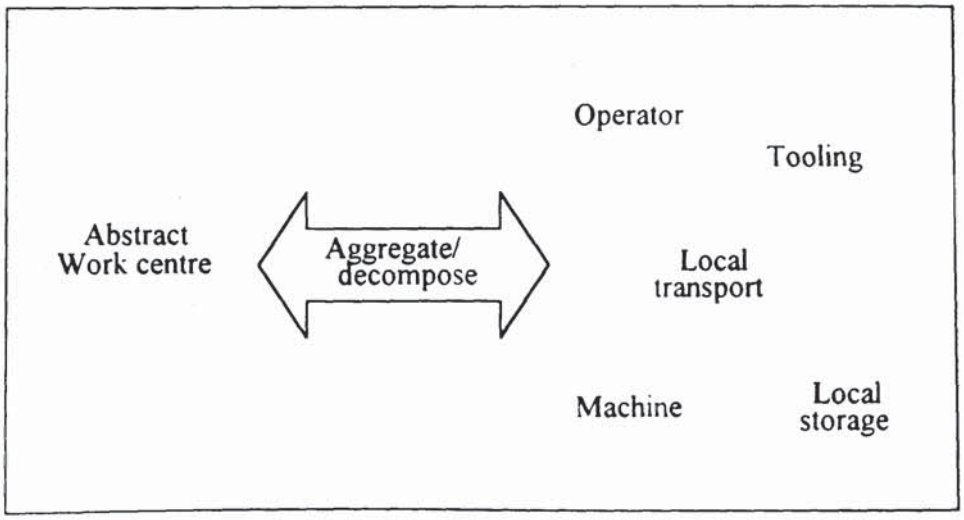


Figure 7.8. Aggregating classes

In an object-oriented approach to multi-level modelling it is likely that the levels will be constructed by aggregating collections of detailed classes. This is in contrast to the decomposition approach used in techniques such as IDEF. With the object-oriented approach the levels are constrained by the detailed classes whereas in decomposition approaches the levels are constrained by the top level abstraction. It has been shown that detailed classes can be abstracted by employing concepts and terminology from reality. Mistakes or inappropriate abstractions will be dampened as classes are aggregated. With the decomposition approach it is likely to be harder to develop abstract representations of the real world. Furthermore inappropriate abstractions of the real world will be amplified as decomposition takes place. See Figure 7.9.

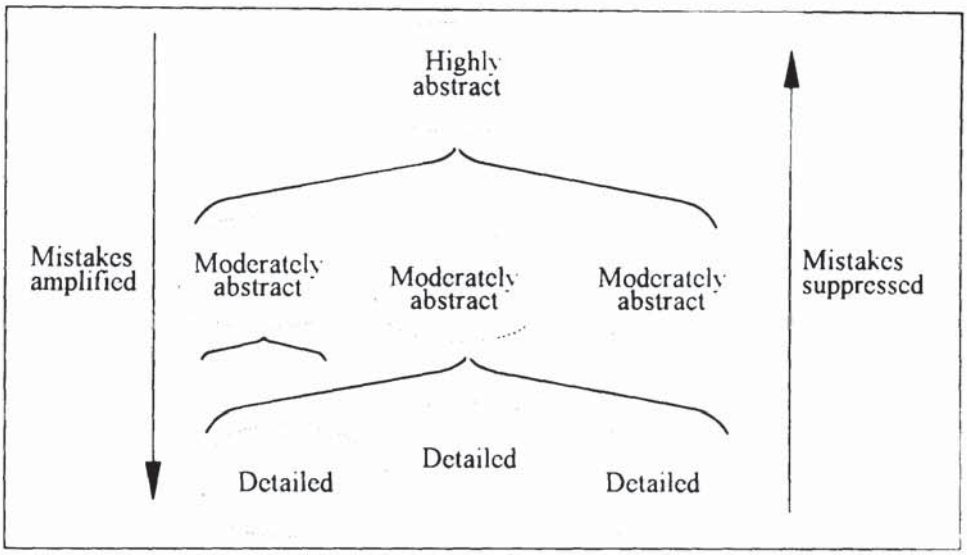


Figure 7.9. Approaches to creating multiple levels of modelling detail

Another issue concerns the number of levels of modelling detail available. Use of object-oriented techniques has potential to provide numerous levels of modelling through the aggregation of classes. For example, a machine and an operator class could be abstracted into a 'supermachine' class; collections of operators, machines, etc. could be abstracted into work centres or cells; collections of cells could be abstracted into manufacturing systems. The limit to the number of levels depends largely on the number of abstract classes a developer is prepared to create. It must be noted, however, that any abstraction must be understandable to the user and generate meaningful results.

7.6.3. Messages and multi-levelling

The creation of more abstract modelling levels could be guided by the message passing described earlier. If a group of classes were to be represented by a single abstract class then the abstract class would possess the same external message links as the group of classes together, see Figure 7.10. This concept is similar, if not identical, to that used in input / output charting or functional decomposition, for example the IDEF methodology.

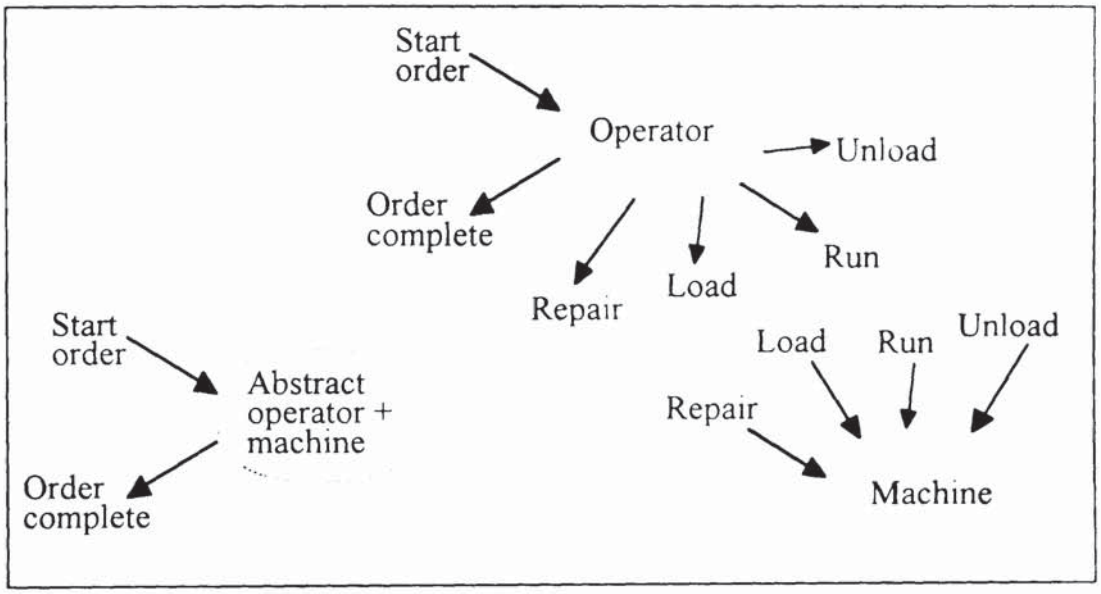


Figure 7.10. Using messaging as a guide to abstracting

7.7. Ease of use of simulation tools

Criticisms have been made against simulation tools that require programming skills to apply. Clearly the simulator thus far described requires software development to extend the range of functionality and could therefore be criticised on the grounds of ease of use. The development of new functionality is a separate activity, however, to the application of the functionality to model a particular manufacturing system. This therefore gives rise to two different types of user: the model builder who would use the simulator to build models easily and the software developer who would use the architecture to add new functionality easily. This process was illustrated previously in Figure 6.2.

The process of development of object-oriented software is complex and requires time and skill. Such time and skill requirements are beyond those of the average model builder hence facility to add new code would not be included. Additionally it would be extremely undesirable for a novice to become involved in software development as

poorly designed and incomplete classes could lead to problems for other developers later on. To ensure that this potential problem is avoided the processes of model building and software development can be separated. The software developer would use programming tools to develop new software and pass new versions onto the model builder. The model builder would not have access to source code or programming tools. Using this approach preserves ease of use whilst providing flexibility.

Data driven simulation tools, or simulators, are potentially much quicker and much easier to use. In developing such a simulation tool there are two key issues. Firstly, using engineering concepts and terminology in the user-interface. Secondly there is the issue of providing a user-interface that is easy to modify with respect to the developer.

In the most pure form a user-interface merely provides a means by which data can be entered, edited and retrieved and does not form part of the data storage or manipulation. Separation of simulation mechanisms from the user-interface can be advantageous. Separation will prevent the spread of simulation logic and therefore contain the complexity. Also separation may allow the user-interface development language to be different from that used for the core simulation logic. If the core simulation architecture has been well designed then provision of the user-interface should be relatively easy.

The user-interface should be developed using object-oriented principles. Complexity should be contained and mechanisms should be put in place to allow ease of modification and addition. Provided that the core simulation architecture has been well designed the user-interface should not be required to make any conversion of data or concepts.

Some generic simulators require both data and logic to be provided by the user. The necessity to develop logic statements will require much more time and skill on the behalf of the user. Ideally models should be created using data only:

> *"The essence of good design method is to clarify and sharply delineate the distinction between what must be program and what could be data, and with careful thought it is surprising what can be data."*
>
> Crookes (1987).

A simulator architecture that has been well designed will enable provision of a user-interface that only requires the user to specify data to build models. The format and type of data required should be such that it can be found or generated easily by the engineer.

7.8. Matching engineering concepts

It has been shown above that it is possible to abstract real world concepts and terminology into software. It is possible to guide the process of abstraction such that the end product will present concepts and terminology familiar to manufacturing engineers. Construction of a simulator using object-oriented techniques could therefore provide a user-interface that is relatively easy to understand and use.

The use of objects as a basis for multiple levels of modelling detail has also been discussed. Objects can be aggregated to form more abstract objects. These abstract objects could present abstract views of a business that engineers and others would find familiar. The use of object-oriented techniques could therefore enable the development of a simulator that could be used by a range of personnel for operational purposes or used for analysis throughout a manufacturing system design process.

The design of an object-oriented simulator could be such that the application specific software is distinctly separate from the supporting simulation software. Hence developers could add new functionality to the simulator with minimal effort. Manufacturing engineers would therefore be able to use an object-oriented simulator and, with the support of software developers, would not be constrained to modelling predetermined types of manufacturing systems. As manufacturing engineers have new requirements software engineers can meet or anticipate those requirements.

8. Construction of the Advanced Factory Simulator

This chapter details the mechanisms that have been used to build the Advanced Factory Simulator (AFS). The construction of AFS is such that it overcomes the compromise between ease of use and potential range of application typically associated with simulation software.

8.1. Overview

The Advanced Factory Simulator (AFS) has been designed (Ball & Love, 1992) and developed (Ball & Love, 1993) using object-oriented techniques. Object-oriented analysis and design processes have led to the implementation of an object-oriented software system. Much of the development has been assisted by the use of an object-oriented programming language, Borland Pascal (the suppliers of Borland Pascal and other software used are detailed in Appendix 1).

The functionality is provided by a library of application classes. These application classes are managed by number of support classes. The architecture is such that new application classes can be added with little or no change to the support classes or other application classes. It is believed that the functionality in AFS matches concepts and terminology found in reality and that the match is such that some functionality is superior to that found in other manufacturing simulators. Whilst the range of functionality may be limited at present there is potential for new additions.

AFS has been developed into a working system and has been used by a number of people. Those people involved in software development have introduced new functionality by making use of the existing source and the commercial development environments (Borland Pascal and Microsoft Visual Basic). The introduction of new functionality was achieved relatively easily. Those people involved in model building have created models using the functionality developed. These users have been able to build representative models using data only, there was no access to the software development environments. In the case of the "U"-shape (Nagare) cell modelling, the author developed and added functionality to the system following a request from a potential model builder. The following sections deal with the developers' view of AFS

The mechanisms employed will be described as well as the current and potential benefits offered.

This chapter provides an overview of the AFS software. The commercial software used is listed in Appendix 1. The requirements, in terms of both hardware and software, for using AFS are detailed in Appendix 2. Due to the size and complexity of the software, most documentation is covered by the Developers' Manual (Ball, 1994). Appendix 3 does list summary documentation and detailed documentation can be found in each software file. It is believed that by using this approach documentation is more likely to be correct and avoids the situation of having large, out of date documents in use.

8.2. Macro design

8.2.1. Separate applications

On a macro level there are a number of requirements that a simulator must fulfil. These requirements include creation and modification of a model, simulation, animation, presentation of results and provision for help. Within AFS, each of these requirements is fulfilled by a separate program or application. These applications communicate with one another through a DDE (Dynamic Data Exchange) messaging mechanism, DLL (Dynamic Link Library) calls or Microsoft Windows messages. See Figure 8.1. DDE is a Windows standard inter-application communications protocol that allows separate applications to communicate using predefined standards. A DLL is a software library that can be used by one or more applications, allowing data and mechanisms to be shared rather than duplicated.

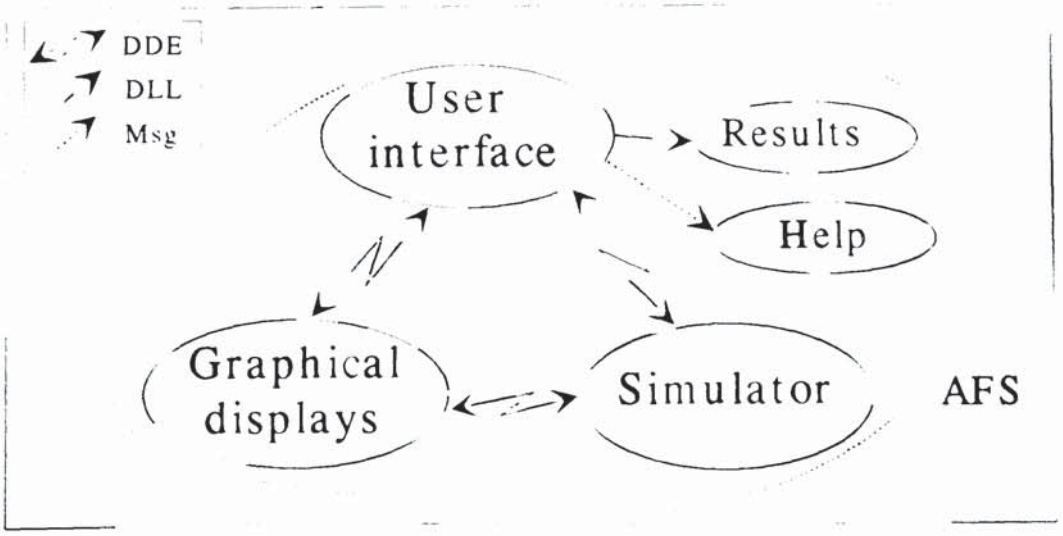
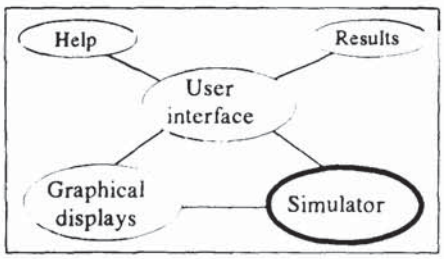


Figure 8.1. A macro view of the Advanced Factory Simulator

Whilst a number of applications make up the AFS system, the software appears to the user as a single application. Except for the help system, the user-interface is the only application that appears to be used directly.

8.2.2. The simulator



The simulator is responsible for the model representation and simulation. The simulator can be split into two distinct parts: the application classes and the support classes. The application classes refer to those available for model building, for example machines, operators and trucks. The support classes enable the application classes to exist in a simulation environment, for example event list and clock. The style of construction of the simulator is key to simultaneously providing ease of use with a potentially wide range of functionality.

The simulator has been designed and implemented using the full range of object-oriented principles. The result has been the creation of an architecture to which application classes can be added. The architecture is independent of the functionality and therefore new functionality can be added without modification, see Figure 8.2. Objects of classes that provide new functionality are able to participate in the simulation, to have their results collected and presented, to be used by and use other objects without modification of the architecture or existing classes. Such a structure is extremely powerful and has the potential to support modelling of any application and not just those areas that are typically modelled using simulation software.

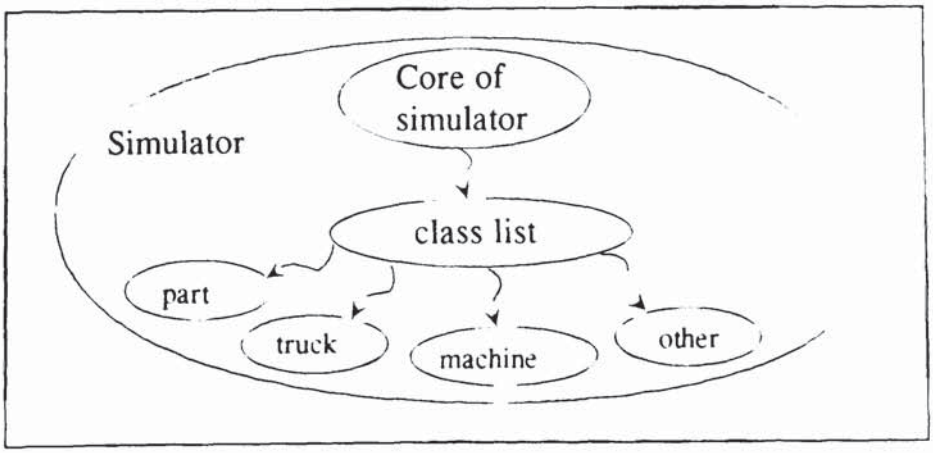


Figure 8.2. Addition of new functionality to the simulator

The mechanism illustrated in Figure 8.2 is achieved using class managers. Each class has a corresponding class manager. All management routines for objects of a class are handled by an associated class manager object. Such routines include object creation and editing. An instance (object) of each of the class managers is created via a registration unit when the simulator program is started. This registration process places the class manager objects onto a class manager list. This list can then be searched and manipulated to allow objects of each class to be created and edited via the class managers. The class manager is the only link between the simulator architecture and the functionality hence for new additions to the simulator the only change required to the architecture is two lines of code to create an object of the class manager and place it on the list.

Communication is based on message passing. The message passing mechanism allows objects to communicate with minimal knowledge of one another. For example, a supervisor could instruct an operator to start an order using a message. The actual message passed is an object. The detail of the message is encapsulated within the object and therefore the object must be interrogated to ascertain the message content. Using this approach the message can be passed between objects (e.g. from a supervisor to an operator) using a generic mechanism. This generic mechanism is a method called 'ReceiveMessage' which takes the following form:

```
PROCEDURE tClass.ReceiveMessage(pMsg : tMessagePtr);
     BEGIN
          { message interpretation code goes here! }
     END;
```

The parameter "pMsg" is a reference (pointer) to a message object. Since all objects capable of receiving messages possess this method the way in which messages are passed is identical for all objects. If the mechanism of message passing is the same for all objects then objects can send messages without prior detailed knowledge of one another.

The use of message passing (instead of method or function calls) allows classes to be very loosely coupled. Although a supervisor can instruct an operator to start an order the supervisor has no knowledge of the operator object, only that it can respond to the instruction. Hence the supervisor could be configured to send that same instruction via the same mechanism to a work centre modelled at an abstract level, an operator with

production control duties or a scheduling board which operators subsequently read, see Figure 8.3. Changing between these different recipients of the message does not require modification of the supervisor class. Also if a new recipient is developed, perhaps an automated production cell, then the supervisor could send messages to it without modification. The messaging mechanism is key to the ability to continuously extend the software.

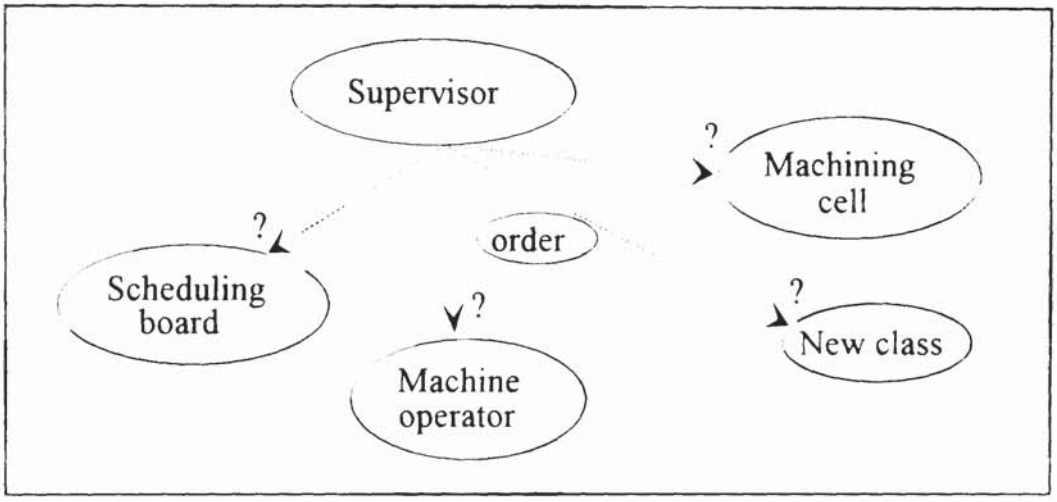


Figure 8.3. The release of a works order by a supervisor

The simulator architecture enables functionality to be extended and therefore benefits the software developer. The style of implementation of the functionality is largely the concern of the user. Careful object-oriented analysis of the real world permits the development of functionality that will ease the task of the user. The functionality will consist of objects that are identifiable in the real world such as machines, operators and trucks. It is important that the functionality is properly represented and the interaction natural since this will be reflected in the user interface. Matching functionality with objects found in reality will allow the creation of a user-interface that is natural to the user and therefore easy to use. For example, an operator will possess a number of skills, one of which could be for machine operation.

In examining the capabilities offered by any object-oriented software consideration should be given to both current and future functionality. The current functionality can be employed by the user immediately. Examination of the object hierarchy will show what could be potentially added by the developer. By viewing the object hierarchy of AFS the current functionality can be shown as well as the potential. A selected hierarchy of the

classes present within the simulator is shown in Figure 8.4. A full hierarchy of the AFS classes is shown in Appendix 4

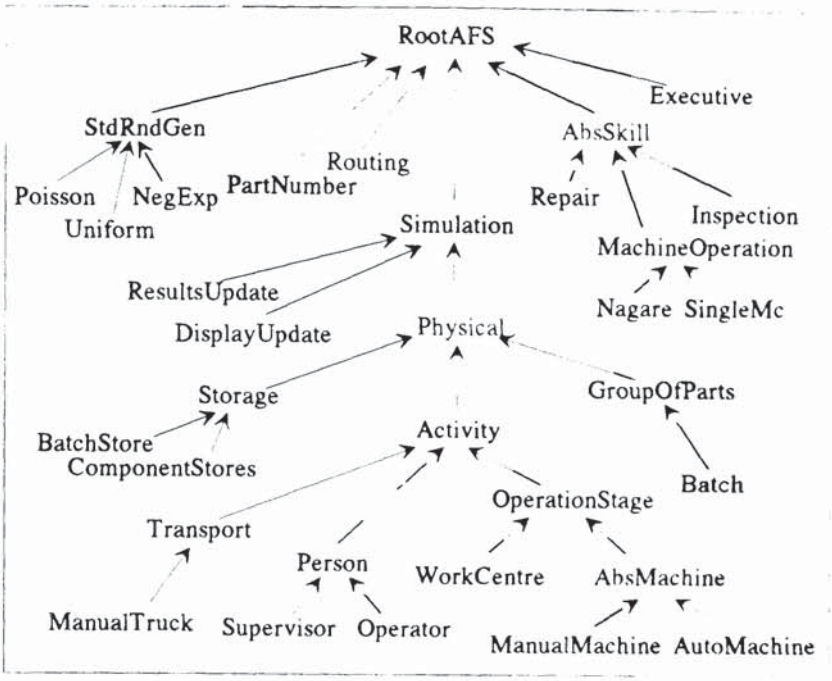
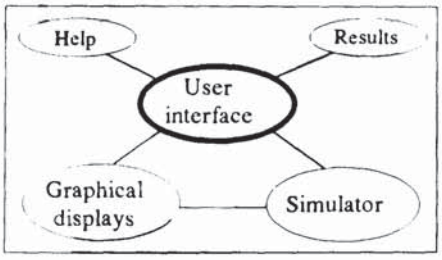


Figure 8.4. Selected AFS class hierarchy

8.2.3. The user-interface



The user-interface must provide a means by which a user is able to build, edit, run and evaluate simulation models. In order to reduce the time and skill required the interface must employ terminology and concepts used in the real world.

Graphical User Interfaces (GUI) are becoming a standard form of interface to software. Microsoft Windows and Windows applications are typical examples of this. Microsoft Windows was chosen as the development platform for two reasons. Firstly, an environment was required in which two or more programs could be executed concurrently. This would allow the simulator to be interfaced with other software such as statistical analysis packages or packages that would form part of the model, for example an MRP system. Secondly tools were required that would assist in the rapid development of a standardised user-interface. Microsoft Visual Basic is one such tool and was used for the user-interface development.

The construction of the user-interface can be divided into elements that are specific to the functionality offered and elements that would be common to any functionality. Where possible elements of the user-interface are made common. The parts that are specific to the functionality are the dialogs for viewing and editing data such as the shift pattern, efficiency and skills of an operator. Common dialogs include the display management and the simulation set up and run. The structure described is illustrated in Figure 8.5. By maintaining commonality, when new functionality is added the number of changes required to the user-interface is minimised.

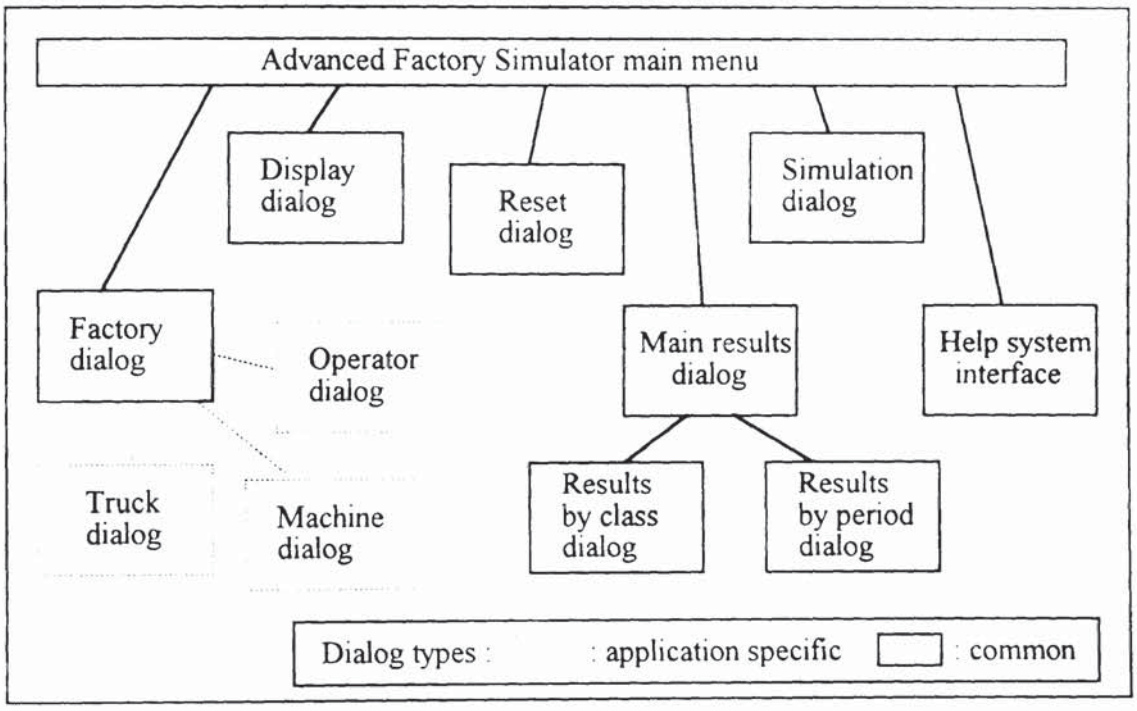


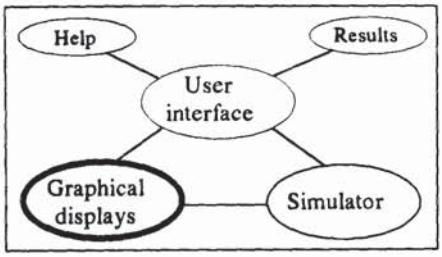
Figure 8.5. Common and application specific dialogs

Visual Basic v3 has limited provision for object-oriented programming. Where possible object-oriented principles were applied even though the syntax did not provide full support. An example of the use of object-oriented principles is the mechanism by which application dialogs are loaded. Here a routine is supplied with the name and class of the object to edit. The routine finds the latest dialog and loads it. Since the mechanism is encapsulated, the method by which dialogs are loaded can be modified without affecting other parts of the software. The dialogs can also be modified without knock-on effects, see Figure 8.6.

```
SUB LoadDialog(className as string, objectName as string)
  'set the name of the object globally, cannot load dialog with parameters
    gObjectName = objectName
  'search for the appropriate dialog
    CASE SELECT className
         CASE "machine"
                  'load the machine dialog for the object
                    MachineDialog.Show 1
         CASE "operator"
                  'load the operator dialog for the object
                    OperatorDialog.Show 1
         CASE etc.
         CASE ELSE
                  'failed to find the appropriate dialog
                    beep
    END SELECT
END SUB
```

Figure 8.6. Encapsulating the application specific dialog loading

8.2.4. The graphical displays



Graphical displays are used in simulation tools to provide the user with a means by which the simulation models can be viewed. Since displays can act as a meaningful way of presenting some of the data contained within a simulation model then they can be used as a means by which the user interacts with the tool.

Within AFS the graphical displays are managed by a separate application, or program. A number of displays can be supported. Each display can show different selections of the objects in the model using icons imported from Computer-Aided Drawing (CAD) software. The use of multiple displays can be used to good effect since different users will be interested in different parts of a model. For example, users may be interested in a particular part of a manufacturing system, the flow of materials or the behaviour of operators. The displays are responsible for displaying the physical position and size of objects. Each display can be independently scaled and can be updated either as a specific object changes state or at specific intervals of simulated time.

The simulator application is responsible for managing which objects are on which display and when the displays are updated. The simulator application then uses this information to drive the graphics application. The simulator has a display manager that maps its own logical displays to physical displays in the graphics application, see Figure 8.7. The object-oriented style of the graphics and the simulator construction is such that the logical displays are independent of the simulation objects displayed and the type of physical display used. Such independence provides the simulator application with the potential to manage different designs of display on different terminals of a network.

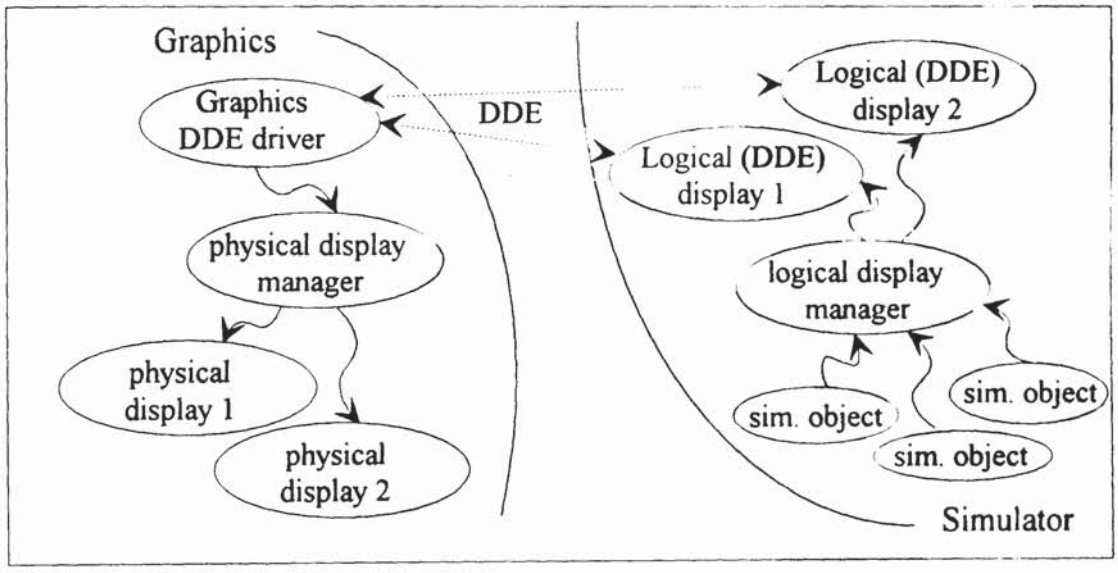
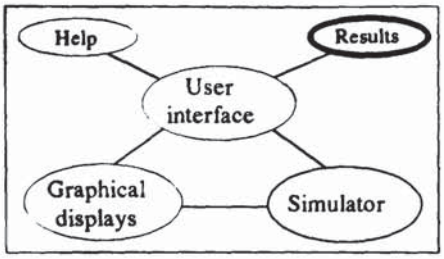


Figure 8.7. The display architecture

The user can interact with a simulation model either via the menus and dialogs in the user-interface or via the graphics application. By dragging an icon the object's co-ordinates in the simulator are updated. By double-clicking on an icon the dialog for the given object is shown. When certain dialogs are loaded in the user-interface the dialog will be updated according to clicks on any of the displays. For example, if the routing dialog is loaded and the user selects a work centre on a display the routing will be updated for the currently selected operation.

8.2.5. The results mechanisms



During the simulation individual objects will collect and store results on their activities. The results mechanism is managed by a results manager. The results manager is able to instruct objects to start and stop recording results according to user defined recording patterns. Results can then be retrieved from each object for each recorded period.

The results mechanism in the user-interface is a generalised one. Results of objects can be displayed without prior knowledge of the objects or their class. Results of objects of newly added classes therefore can be displayed without modification of the user-interface. The mechanism works by providing the user with a list of classes present in the simulator. When the user selects the class the user-interface retrieves the results from the simulator as well as a header to indicate what the results are, see Figure 8.8.

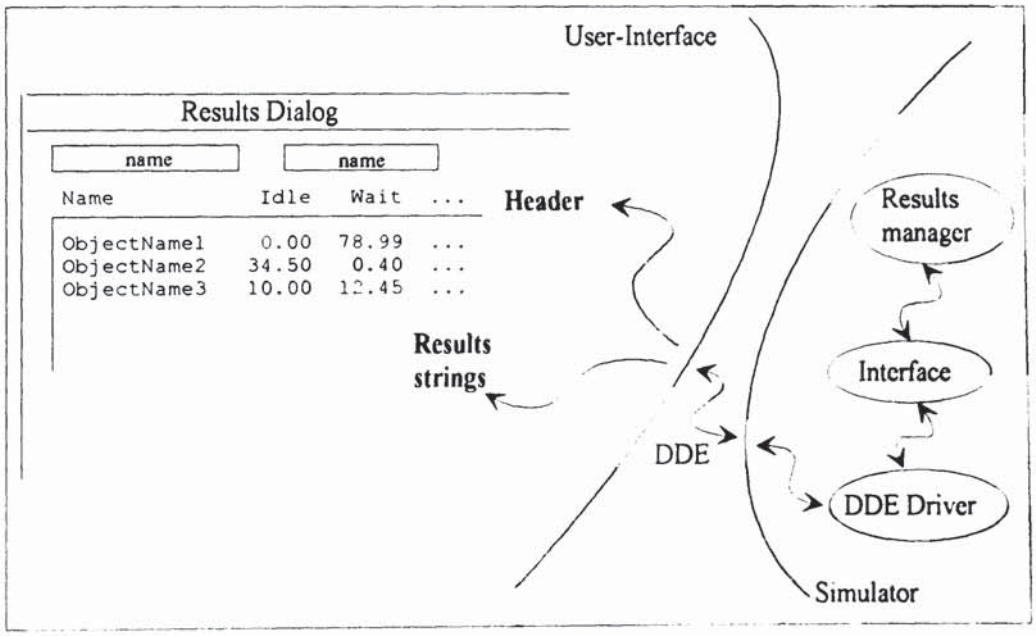
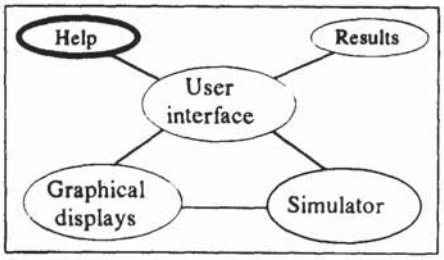


Figure 8.8. The results display mechanism

At any point the results can be displayed graphically by selecting one or more of the rows of results. A graph is displayed using a commercially available package called Graphics Server. Mechanisms in the user-interface initialise the graph and supply the data to display. It is envisaged that statistical analysis will be provided by another commercially

available package, possibly SPSS. Features in both Visual Basic and SPSS enable programmed links to be set up similar to those controlling the Graphics Server application.

8.2.6. The help system



The help system was authored and compiled into a Microsoft Windows help application. This approach had the benefit of presenting a standard approach to software help. The help system was developed with support for context sensitive help. For example, if the 'F1' function key is pressed whilst editing a machine the machine help topic will be shown. It is felt that the content of the help system is such that supporting user manuals are not required.

8.2.7. Benefits of the approach

There are a number of benefits resulting from the macro approach described. These benefits include:

- Ease of location of mechanisms: separation of the editing, display and simulation mechanisms eases the task of locating elements of software;
- Reduction of knowledge of software required: independence of the various mechanisms means that during modification of a mechanism less knowledge of other software is required;
- Ability to use different development environments: separation of the applications enables the most appropriate development environment to be used, for example Visual Basic greatly eases the task of developing a user-interface;
- Ease of replacement: independence of software mechanisms permits their replacement without affecting other parts;
- Provision for portability: the use of different development environments results in the potential to move the system to a different platform, for example the simulator application has been compiled for DOS or Windows.

There are also a number of benefits associated with the design of the simulator application:

- Complexity is contained: independence of the application and support classes prevents the spread of complexity;
- Ease of modification: independence of all classes minimises the amount of modification required when new classes are introduced.

8.3. Configuration

One of the major problems of creating an extensible architecture is that of configuration. Configuration is defined here to be the linking of objects or classes together, for example assigning a machine object to an operator object. The link is based on message passing, for example an operator sending a load message to a machine. Extending the functionality of the simulator whilst minimising the changes to existing software is difficult to achieve simultaneously; it is difficult to add a new class and make other classes aware that it exists without modifying them.

AFS employs a dual level configuration mechanism (or 'configurator') whereby it is possible to link classes together and objects together. The configuration mechanism is used at run time to authorise links between classes and between objects. By linking at run time, classes do not need to know of the existence of each other when they are added to the simulator. System developers are able to link classes through the configuration dialog (see Figure 8.9) whilst engineers would use the specific dialogs for objects, for example the machine and operator dialogs. An example of class configuration would be linking the 'operator' class 'load' message to the 'grinder machine', 'milling machine' and 'broach' classes. An example of object configuration would be making an employee called 'Operator1' responsible for 'loading' and 'unloading' machines 'Machine3' and 'Machine5'. Only links that have been authorised at class level can be implemented at object level.

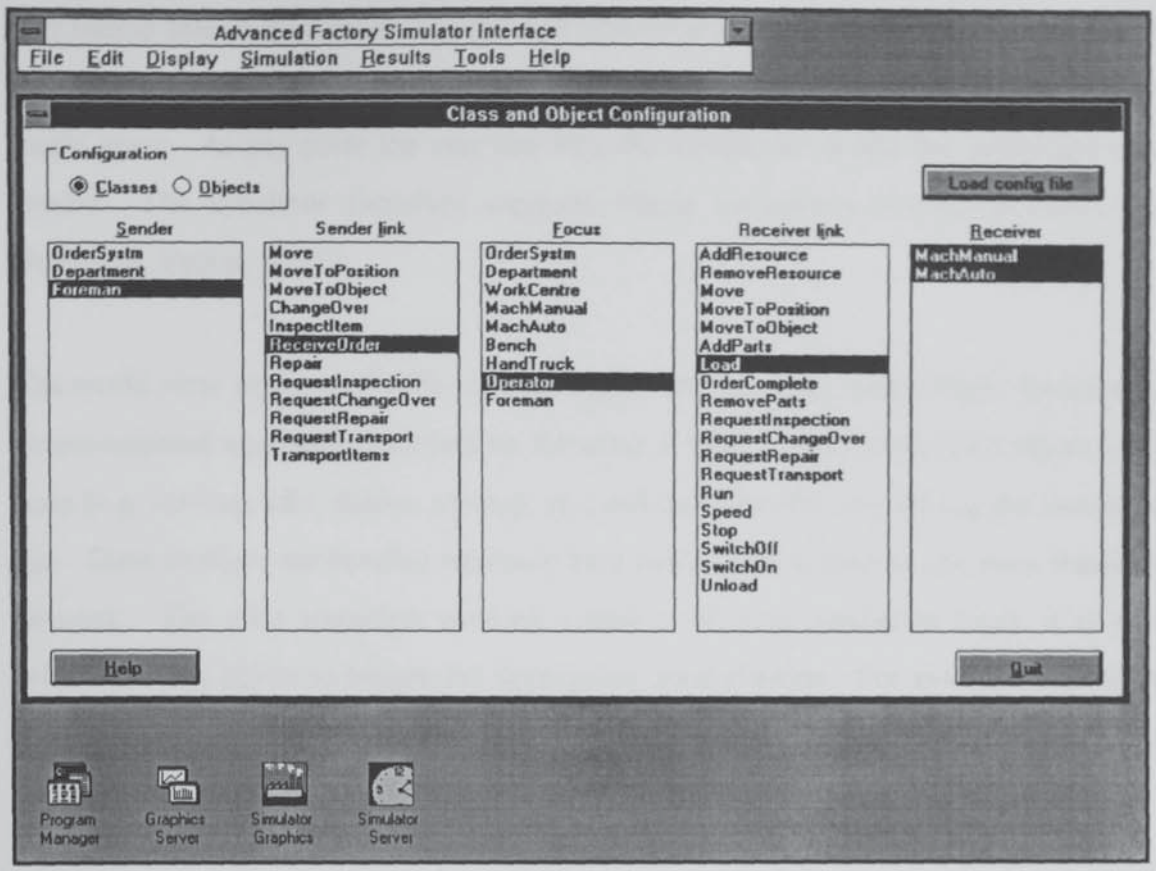


Figure 8.9. The message configuration dialog

Since the configurator is able to query the messages that classes and objects can send and receive, the configuration mechanism can be used to good effect during model building. For example, the user may wish to set a supervisor to release orders within a cell. The user-interface will be able to suggest objects within the cell that the supervisor could instruct to start an order. If no objects exist then the interface will be able to suggest objects that could be created to receive the order message. The process is achieved behind the scenes using the configurator. The flexibility of the mechanism is such that the user-interface may suggest a range of objects including: a machine operator; an operator with production control duties; a work centre being modelled at an abstract level. When new functionality is added it appears automatically in the user-interface. The user-interface does not require modification.

8.4. Simulation executive

The executive is responsible for controlling simulation runs. When a user selects to simulate a model the executive ensures that the correct sequence of events are actioned and that the model is stopped at the appropriate point.

The user is able to simulate to any point in time by selecting a run length in terms of date and time, number of simulation periods (simulation periods are one or more days) or by a single event. At any point the user can stop the simulation to edit the model and view results. The simulator therefore supports Visual Interaction Simulation (VIS) (see Hurrion & Secker, 1978).

The world view employed in the construction of the simulator most closely matches the object-oriented approach described by Kreutzer (1986). Each simulation object has a state (e.g. running, idle, absent, moving, etc) and the state will vary during the simulation run. State changes are handled internally by a mechanism known as the state transition network. The state transition network contains the core simulation logic: it is used privately by an object to trigger the appropriate state changes. For example, a machine has the states running, idle and broken down and the state transition network is used to decide under what conditions the state changes should occur.

The state transition network is a method that is private to the particular class and classes that inherit it. It is possible to inherit a class and overwrite the state transition network method. Using this approach allows inheriting classes to alter the behaviour of the class by overwriting the method and adding new mechanisms (an object-oriented concept known as polymorphism). Making the simulation logic private to each class enables the simulation logic to be contained within a small number of easily identifiable places in the software. For example, all the event logic associated with a machine breakdown is held within the machine class state transition network method.

Each object can access the world clock. Objects use date and time information supplied by the clock and their own time and state records to decide what state to change to, if at all. Because of the access to the clock, each object is able to ascertain the time of the next event for themselves. Each object will calculate the time of the next event and request the simulation executive to schedule the event.

The simulation executive has no knowledge or access to an object's state transition network. The simulation executive is solely responsible for instructing an object to update itself at the appropriate time. The executive therefore does not contain any

simulation logic and exists to schedule events for each object in the correct order. Essentially the executive exists to synchronise objects.

Encapsulated within the simulation executive is the event list, a chronological list of events. During simulation the executive will take the first pending event off the event list and instruct the object to update itself. The event list contains only the date, time and object reference for an event and has no knowledge of what event is about to occur. Objects possess a common method for updating. This process is illustrated in Figure 8.10.

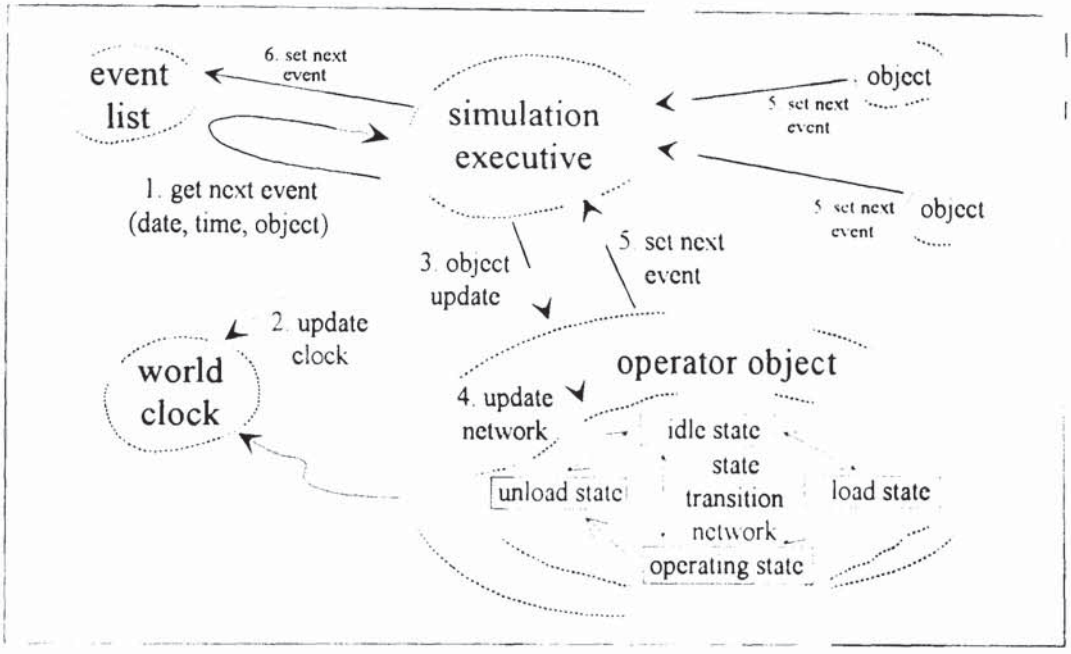


Figure 8.10. The simulation mechanism

The object performs its own time advance and state changes via its state transition network. Hence any object that conforms to a given standard can be placed on the event list and be instructed to update itself. The architecture can therefore support modelling of objects ranging from simple machine tools to an object representing a business function such as the material control system as well as allowing displays and results to be updated at user-defined times.

8.5. Expansion: mechanism and potential

The mechanism for expanding AFS will vary depending on whether the new requirements are a new style of dialog, a variant class or the introduction of a new class. The required changes will therefore affect the user-interface and / or the simulator.

To add a new dialog to the user-interface the new dialog is created and registered inside a dialog loading routine (refer back to Figure 8.6). The new dialog is loaded by supplying the dialog loading routine with the name of the class that the dialog relates to. Since the dialogs are private to the dialog loading routine they can be replaced without changes to the rest of the interface. The dialogs retrieve data from the simulator by calling on what appears to be a simple function. The function makes use of a DDE (dynamic data exchange) mechanism to retrieve the data from the simulator.

Addition of a new class to the simulator requires the creation of the class itself and a class manager. The class manager handles tasks common to all objects of the new class such as creating, editing and saving objects. To add the new functionality to the simulator the class manager must be registered. The registration simply involves the creation of a class manager object and its insertion onto the class list (refer back to Figure 8.2 and see Appendix 5). Once the class manager is on the list, objects of the new class can be created, edited and participate in the simulation. Depending on the implementation, objects of the new class can use and be used by objects of other classes without modifying them. The ability to add new classes in this way is made possible because of the architecture design and the message passing mechanism. Invariably the new class will inherit the properties of an existing class.

Since new classes can be added easily and with no increase in the software complexity the scope for expansion of the simulator functionality has no predefined limits. The limits of the simulator functionality rely very much on the range of classes that software developers are prepared to add. The simulator architecture was first proven with the addition of shop floor functionality. Currently production planning and control system functionality is being developed and added (Boughton & Love, 1994). In the longer

term the development of support department functionality to include areas such as the finance department is planned (Jackson & Love, 1994).

8.6. Mechanisms to assist developer

Two mechanisms in the simulator application assist the developer in adding new classes: the class (manager) list and the configurator.

The class manager list was introduced earlier. It holds the manager object associated with a class, for example the routing class manager object manages the functions associated with the routing class. The user-interface, using DDE, will query this list to ascertain which classes are present. These classes are then displayed in the user-interface. Hence when a developer adds a new class to the simulator the class name automatically appears in the user-interface.

The configurator is able to ascertain which messages each class can send and receive. The developer will declare which messages a new class can send and receive. At start up the configurator checks the message links and alerts the developer of missing links. For example, a developer may add a 'crane' class that can respond to the messages 'lift' and 'lower'. If there are no classes, hence objects, that can send those messages the developer's attention will be drawn to this.

8.7. Contributions by other developers

Evidence to support the claims that AFS can be extended can only come from actual demonstrations of the process. During development new mechanisms and new application classes were added. This demonstrated the ability to extend the functionality as well as enabled refinement of the mechanisms that supported it. More objective assessment can be derived from individuals not involved in the original software development.

A number of individuals from the Department of Mechanical and Electrical Engineering at Aston University are in the process of developing and adding new software to AFS. Introduction to AFS and its internal mechanisms came from the author and an AFS

software manual (Ball, 1994). Actual implementation and testing is being conducted by the individuals alone. The additions are such that the modelling capabilities of the software will be extended and that models will be developed containing functionality developed by a various individuals. Typically the additions will involve new applications classes in the simulator and corresponding dialogs in the user-interface. The additions to both the simulator and the user-interface simply involved a registration process. The design, size and complexity of the classes developed are the concern of the individuals.

The functionality developed by the author covered the physical production processes, for example machines, operators, trucks, parts, storage areas, etc. The biggest contribution by others to date is the material planning and control functionality (see Boughton & Love, 1994). This includes centralised and localised mechanisms (for example, Materials Requirements Planning and Kanban respectively). This functionality has enabled combined models to be developed. Contributions are currently being made are for support department modelling, job shop modelling and results analysis.

There are a number of important observations that have been made since other authors have started adding new functionality:

- There were no additional architectural requirements: the displays, event list, executive, results mechanism, etc. did not require modification;
- Software complexity was contained: whilst new classes were added the complexity of the supporting software remained unchanged;
- Old and new applications classes were able to interact and permit combined simulation models to be created, in some cases the old classes could interact with the new classes without modification.

8.8. Practicalities of design and implementation.

This and the previous chapter have described the design and construction of the Advanced Factory Simulator. For clarity this was presented as a linear process, free of limitations. This section will introduce some of the problems encountered during the development process and detail how they were overcome.

8.8.1. Commercial development environments

A number of software development environments were used. A decision was made to use Microsoft Visual Basic for the development of the user-interface. Visual Basic allowed quick and easy development which, at the time, contrasted with other approaches such as using the Microsoft Windows API directly or using program generators such as Protogen. Whilst Protogen enabled visual development of user-interface, programming was still required for its operation. The way in which Protogen regenerated code resulted in the loss of any underlying code previously added for its operation. The code for operation of the user-interface would be potentially very large and therefore this approach was not viable. The drawback of using Visual Basic was the limited support for object-oriented development. As a result containing the complexity of the code was not enforced and therefore the responsibility of the developer. There was also no provision for inheritance.

The simulator application was developed using Borland Pascal. This enabled the development of an object-oriented application using a strongly typed language which was relatively easy to learn and use. Towards the end of the project, as the application grew to in excess of 60,000 lines of code, limitations were uncovered in the compiler and actions had to be taken, such as incorporating text in resource files on disk rather than incorporating it in the program.

Other more detailed limitations of Visual Basic and Borland Pascal are given in Appendix 6. It should be noted that these restrictions result from the use of commercial languages and libraries and during construction efforts were made to avoid imposing restrictions both in terms of programming and use.

8.8.2. Understanding object-oriented techniques

To embark on any programming project will necessitate an understanding of the syntax of a programming language as well as how to develop a program. Developing object-oriented software presents additional difficulties: a knowledge is required of object-oriented programming syntax, object-oriented program development and object-oriented analysis. There is a considerable learning curve to this, compounded in this case by the

need to understand simulation mechanisms. Object-oriented analysis and subsequent programming was perhaps the most difficult area to grasp and this can be illustrated by the difference in the 'quality' of early software and that developed later. This can be attributed partly to the initial inexperience of the author in programming and partly to the development of a new type of software system. Hence the quality of the classes depend partly on the maturity of the developer and partly on the maturity of the application.

Development of applications in the Microsoft Windows environment necessitated some knowledge and use of what is termed the Windows Application Programming Interface (API). Use of the API is essential for applications to run in the Windows environment (rather than in DOS) and for construction and operation of the graphical displays. Work on AFS also made use of other libraries for date and time functions and list storage mechanisms. Using these libraries removes the need for additional development and subsequent testing and removal of errors. The disadvantage of this approach was the requirement to understand how these libraries should be used and the development (and removal of errors!) of software to use them. Despite this, it is felt that using the libraries was quicker and more robust than would have been the case had everything been developed "from scratch".

8.8.3. Using object-oriented techniques

The development of AFS has been described as a process of analysis, design and implementation. Whilst the development did follow these stages, invariably the process was an iterative one in which problems encountered in the implementation stage necessitated further design work, and in some cases required further analysis. Problems arose partly from limitations in the programming language (for example, Borland Pascal does not support multiple inheritance) and partly from problems in the detailed implementation of classes. Booch (1991) and others accept (and advise) iterative development and do not consider detailed implementation issues to be within the scope of the earlier analysis and design stages.

Three or four iterations in the design and implementation of some classes was not uncommon. Iterations were usually triggered by inability of a collection of classes to support a new mechanism or by increasing complexity. As the complexity of software

rises it becomes more difficult to understand and maintain. The iterative development tended to be an act of investment rather than an advance in functionality; typically a collection of classes took many days to redesign and at the end only offered *potential*, not actual, advantage over the old design.

It is also interesting to note that iterations were not consecutive but spread throughout the overall development cycle, often many months apart. The reason for this was that once a collection of classes was redesigned their design and operation could be ignored whilst other classes were developed. Eventually the development and redesign of other classes would result in the need to redesign the original collection. For example, after redesigning the display classes, development might concentrate on other classes such as shop floor functionality or results collection or the simulation executive. Eventually these would develop to such a point that the display classes would have to be modified.

It was rare to pass through four or more iterations for a collection of classes. Those classes which have been redesigned several times are generally stable. In retrospect, such collections of classes fulfil their function well and there have been few errors associated with them since.

9. Use and application of the Advanced Factory Simulator

This chapter discusses the Advanced Factory Simulator (AFS) from the perspective of the user and describes how users are able to build, simulate and view the results of models. The final section provides examples of how the simulator has been used, individuals from Aston University and Lucas Engineering and Systems Ltd. were asked to build models.

9.1. Building models using AFS

The user-interface of AFS has been developed using Visual Basic and therefore it has been possible to provide an appearance similar to any other Microsoft Windows application. Whilst the dialogs have a format that is unique to AFS their operation is the same as any other Windows application. AFS dialogs have the same list box, combo box, menu, button and check box controls as other applications. The keystrokes and mouse movements required to utilise them are therefore standard and so the user will not have to learn a new style of operation. Consequently the use of a Microsoft Windows standard interface greatly eases the process of learning and using AFS.

Construction of the user-interface is such that no logical statements can be entered and there is no access to any programming modules. The user-interface can therefore only accept data items. The model logic is selected rather than conceived and developed hence the user does not require great time or skill to build models. If logic or functionality is not available then, with time, a developer can add it. Like ATOMS (Bridge, 1990), models in AFS can be built using text files. Text files can be created manually or by down-loading data from the manufacturing database. Since logic is not required the files will need little if any modification prior to loading into AFS.

The use of a data only approach to building models enables models to be built relatively quickly. Examples are given later in this chapter. Mechanisms available within AFS can further reduce the time to build and test models. During the simulation if inconsistencies are detected then the user is given the option of stopping the simulation to carry out the appropriate edits. For example, if a cell leader is passed a works order but no one has been assigned to operate the required machines the user is given the option of stopping

the simulation. The required edit could be to set certain operators to carry out the task. Similarly if an attempt is made to place too many parts into a truck the user is given the opportunity to increase the size of the truck. Also it would be possible to set up the configuration mechanism such that prior to simulation AFS would check the message links between all objects. Currently it is possible to check for missing class and object message links. Hence there is the potential to warn the user of missing links. For example, there may be a cell leader who can request personnel to repair a broken down machine but there may not be any personnel able to respond to the call.

When objects are created they are given default values. For example, operators are given an 8am to 4pm shift pattern whilst machines are given efficiency and reliability values of 100%. The data entry required is therefore reduced to a minimum. Basic models can be built quickly and values adjusted once the model has been fully built. Most editing is carried out using dialogs only. The graphical display can, however, be used to adjust the position of objects. The graphical display can also be used in conjunction with dialogs. For example, if the work centre dialog is open and the user clicks on a machine on the display that machine will be added to the work centre shown.

9.2. User-interface vs. real world

This section will describe how concepts described earlier that assist the user have been implemented. Much of the detail is contained in the appendices indicated.

9.2.1. Application class library

The application class library is key to providing the ability to create representative simulation models. If the application class library contains functionality that closely relates to the problem domain then the task of creating a model should be straightforward. A sample of the AFS library is shown below in Figure 9.1.

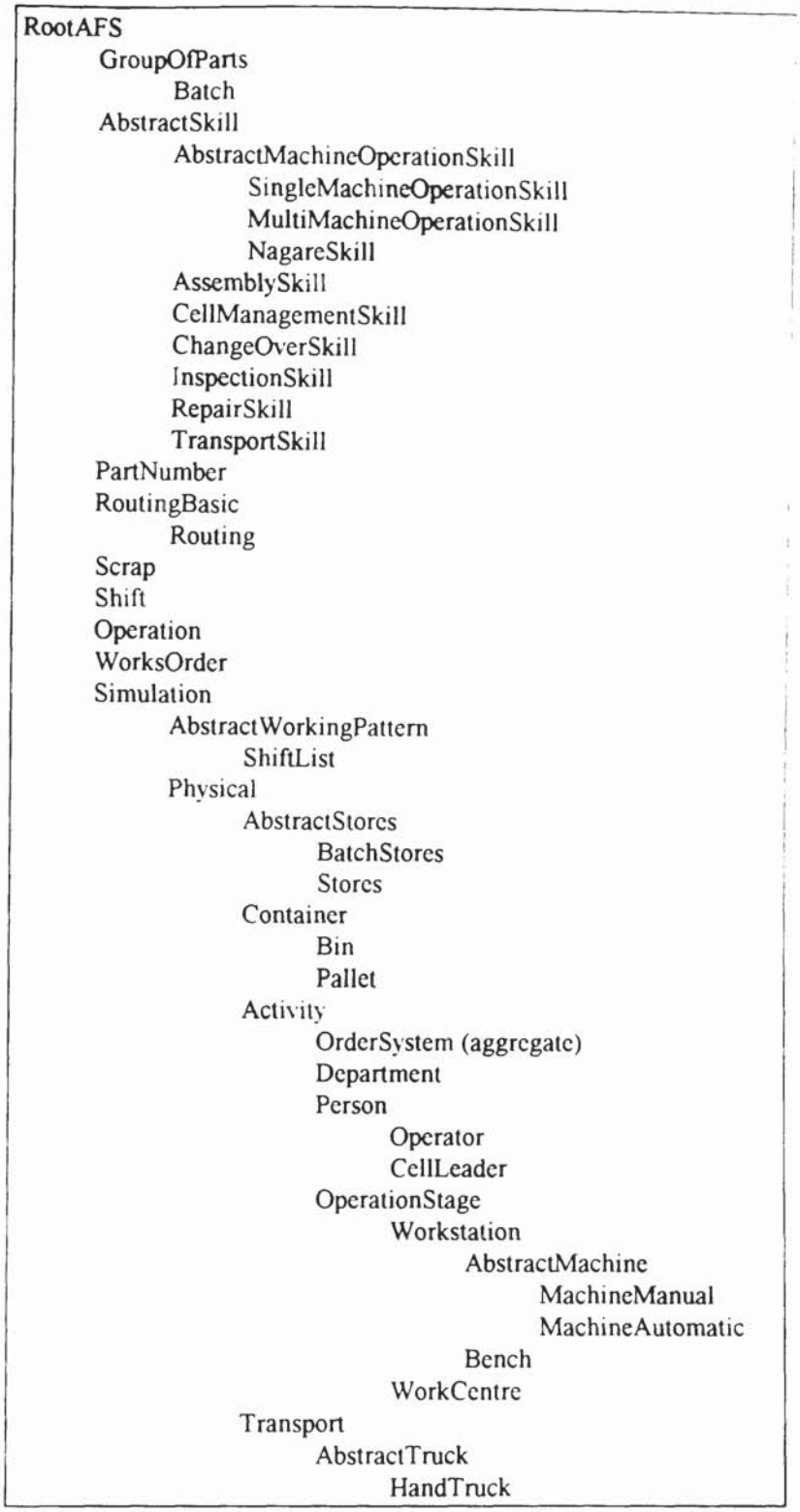


Figure 9.1. Sample of the AFS application class library

The figure shows which classes exist and their inheritance relationships. For example, the HandTruck class inherits the properties of the AbstractTruck class. A full class hierarchy is shown in Appendix 4. The interrelationships between classes, such as the operator class being responsible for loading machines, are difficult to show clearly on a

single diagram. Summary descriptions of each of the classes are given in Appendix 3. The descriptions have been extracted from the header files of each of the classes.

The range of classes represents a coherent set of elements for modelling batch and line manufacturing systems in which operators plays a key part. It will be shown later that this functionality has been used to create a number of distinctly different models. There are two reasons for the concentration on 'low-tech' manufacturing systems modelling:

- it was demonstrated earlier that a majority of manufacturing systems are not fully automated and that this situation is unlikely to change in the near future;
- modelling operators is extremely complicated and therefore the ability of AFS to model their detailed activities shows that complex functionality can be supported.

The functionality developed by the author consists of a mix of direct, indirect and supervisory modelling. Machine and operator classes are examples of direct modelling. Setting and transportation are examples of indirect modelling. Cell leaders and order system modelling are examples of supervisory modelling. This range of modelling demonstrates the scope as well as the detail of functionality in AFS.

9.2.2. User-interface dialogs

Dialogs have been developed for the user-interface to enter and edit data in the simulator application. The dialogs do not translate or alter the format of the data in the simulator, they just provide a means of accessing it. The user-interface therefore faithfully reproduces the internal, developers' view of the functionality. Use of this approach forces developers to use a real world view when designing and constructing the software. This approach also simplifies the user-interface and enables complexity to be contained.

An assessment of the match between functionality supported by AFS and elements of the real world could be made. This should be based on the appearance of the user-interface not the software. Such an approach should be taken since it is the engineer, not the

developer, who is the ultimate user. It is believed that the functionality available in AFS provides a good abstraction of elements of reality.

A typical example of a user-interface dialog can be seen in Figure 9.2. This shows that an operator can be included in the model by specifying department, shift pattern, walking speed and efficiency. Each operator has a physical position, size and icon that can be accessed via the co-ordinates dialog. Each operator can be assigned a range of skills such as assembly, transport and inspection. Other dialogs are shown in Appendix 7. These dialogs demonstrate the functionality available to the user.

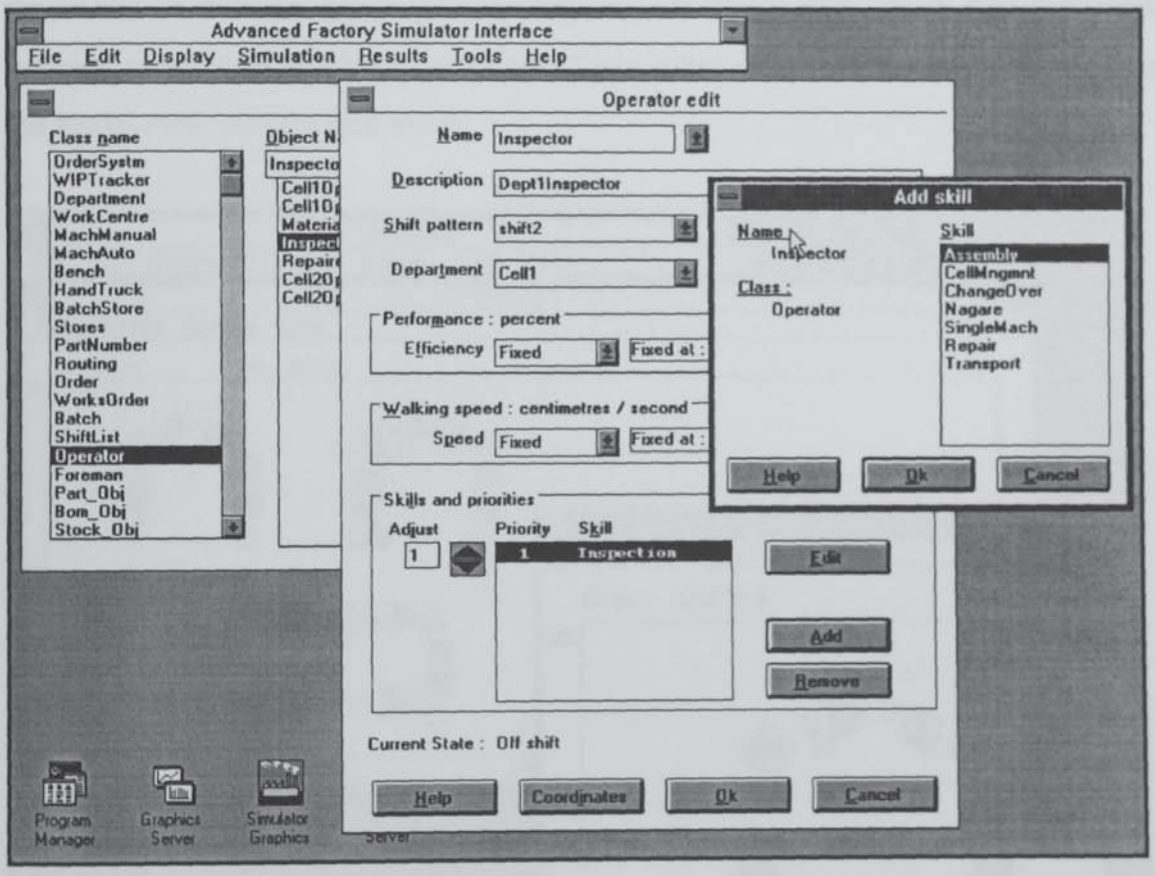


Figure 9.2. Example of AFS model building dialogs

AFS is able to support the addition of classes for which dialogs can then be added to edit objects of the new class. New dialogs can, however, be added without first adding a new class. It is possible to develop new dialogs specific to user requirements. For example, tailored dialogs can be developed to support different ways of viewing or editing the data. This approach provides a means by which ease of use of AFS can be increased. The use of systems dialogs is a good example of this and will be described later.

9.2.3. Graphical displays

Simulation software can be used to assess aspects of a system. For example, users may be interested in the levels of work-in-progress, the movement of operators or the utilisation of machines. Since graphical displays (animation) can be used to provide insight into the behaviour of a system they should be designed with features that can assist this activity.

The use and capabilities of the graphical displays where described earlier. AFS permits multiple displays of a model to be created. The displays can show different elements of the model at different levels of magnification, see Figure 9.3. Such features enable tailoring to the individual needs of a user. This further assists in the process of matching the model view with the real world.

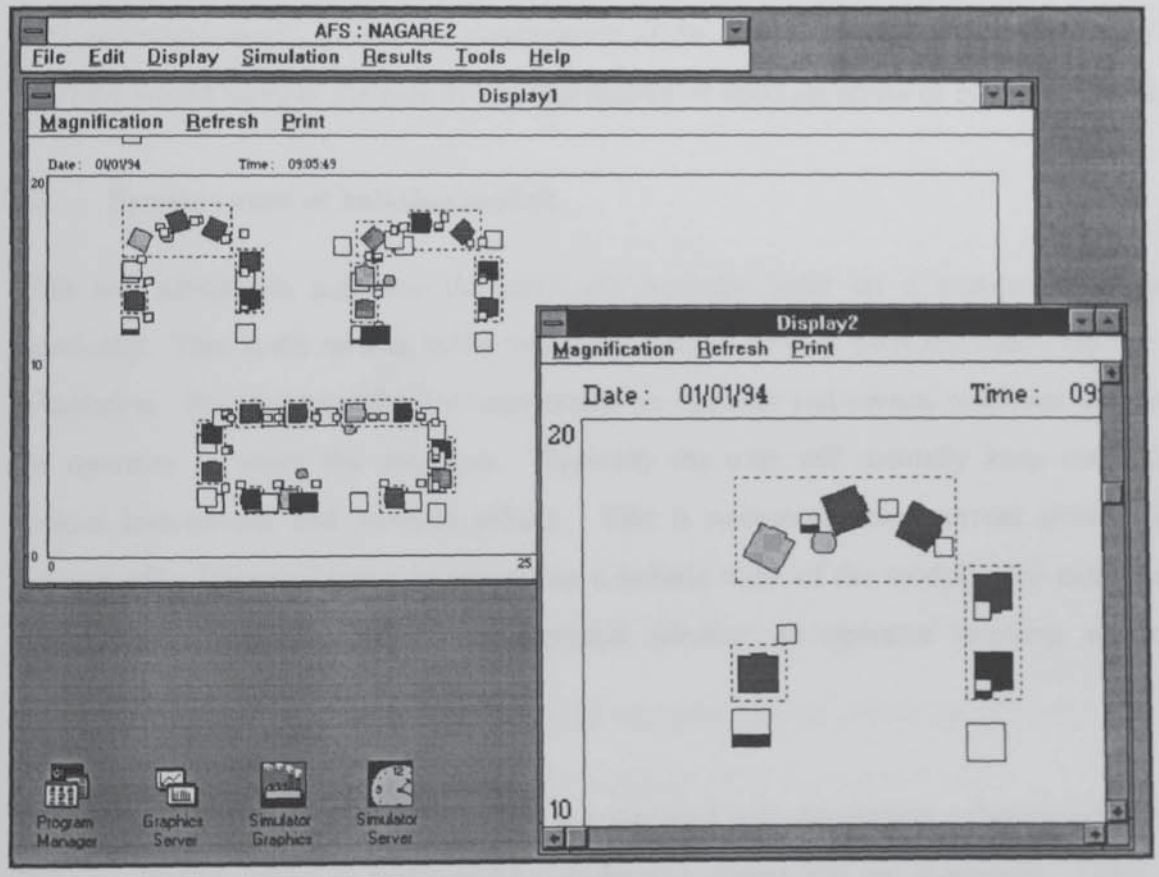


Figure 9.3. Multiple displays of a single model

9.2.4. Potential for multi-level modelling

The use of multi-level modelling has been discussed as a means by which the users' view can be matched within the software. Currently AFS functionality only supports a single, detailed level of modelling. The software does, however, have a number of features that can be used to provide multiple levels of modelling.

Firstly AFS can support the addition of new functionality, hence aggregate modelling elements can be included. Secondly it has been proposed that multiple levels of modelling could be supported by aggregating or decomposing objects. For example a cell could be either modelled in an abstract way or modelled using a collection of operators, machines and trucks. The interface to each object (i.e. the message links that need to be supported) would act as a guide to the aggregation or decomposition. It was shown before that the user-interface is able to query and set up message links at both class and object level. Hence the combination of the message mechanism and the user-interface would support the user in building models of different levels of modelling detail.

9.3. Systems view of building models

With any simulation software the user will typically build up a simulation model atomically. That is the entities in the model are created on their own and linked together in isolation. For example, the user may create an operator and several machines and set the operator to work the machines. Typically the user will mentally keep track of various interactions and possible effects. This is necessary since current simulation systems offer little assistance in presenting a holistic view of the model. For example, before simulation it is difficult to ascertain whether an operator working several machines is an efficient use of resources.

The process of model building, however, is concerned with developing coherent systems. The user should, where possible, build a model as a system and not atomically. Using a systems approach would enable the user to have a clearer view of the model. Such an approach would have the potential to reduce the model build-test-modify cycle.

The object-oriented view permits objects to be grouped to form coherent systems. For example operators, machines, parts and routings can be grouped to form a cell system. The construction of the AFS user-interface allows modifications to the software to be made relatively easy. It is therefore possible to modify AFS to provide a systems view of model building. The user-interface therefore provides a second type of dialog known as system dialogs. These dialogs group objects present in the simulator into a coherent system. System dialogs show how different parts of a model interact and can reduce the number of test simulation runs required to ensure that the model built approximately meets the requirements. One such dialog is the Nagare system dialog that enables users to ensure that the Nagare lines built have first been statically balanced, see Figure 9.4.

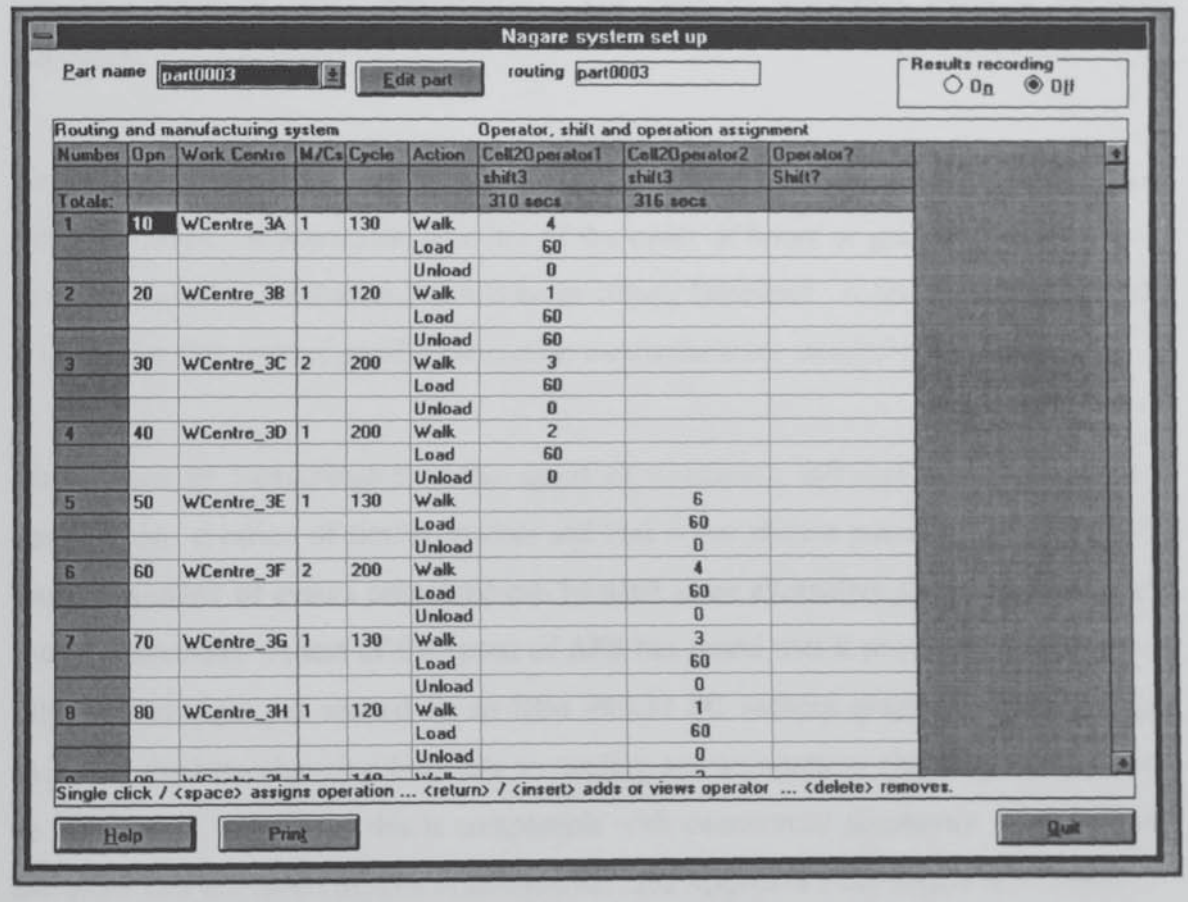


Figure 9.4. System dialog: Nagare manufacturing system set up

The above figure shows a routing down the left hand side of the grid. On the right hand side of the grid the user has selected which operators will carry out which operations. Where an operator carries out an operation the walk, load and unload times are shown as appropriate. At the top of the operator columns the *operator* cycle times are shown. Here they have been roughly balanced to 310 and 316 seconds for operators 1 and 2

respectively. The grid therefore enables a good 'first pass' model and thereby reduces the number of test runs that will be needed.

9.4. Simulation speed

Simulation speed is an important issue in terms of usability. If a simulator or other simulation software is too slow to carry out realistic simulation runs then its use in practice is brought into question. There are few restrictions on the size of models that can be created using AFS and there is also a potentially wide range of functionality. The result is that AFS models could be very large. For example, a model may include the whole of a manufacturing system or may include elements of the production planning and control systems or other departments. The issue of simulation speed is therefore important.

Simulating elements of a simulation model in parallel has potential to reduce the length of simulation runs. If simulation runs are of the order of hours or greater then the use of parallel simulation mechanisms would be extremely beneficial. A discussion in Appendix 8 concludes that current parallel simulation mechanisms are unsuitable for AFS.

Comparison or measurement of the speed of simulation software is difficult since it requires the creation of similar models and that is not always possible. Testing on the basis of number of events processed can be used as an alternative since this will be less model dependent. A test of the speed of AFS has found that it processes an average of 200 state changes per second on an IBM 486/33 PC without graphics. State changes include a machine changing from idle to loading and an operator changing from on shift to off shift. It is believed this is comparable with commercial simulators. It is believed therefore that the speed of AFS is acceptable. See Appendix 9 for details of this test.

9.5. Simulation results

The architecture of the results mechanism was described earlier, however, the description was only relevant to the developer. A manufacturing engineer will use the results facilities without knowledge of the underlying architecture.

By default results are automatically recorded and can be viewed via the appropriate dialogs. The user is able to set up the pattern by which results are recorded and has control over the period length, the periods to record and periods to ignore. The dialog that is used for setting up the recording is shown in Figure 9.5.

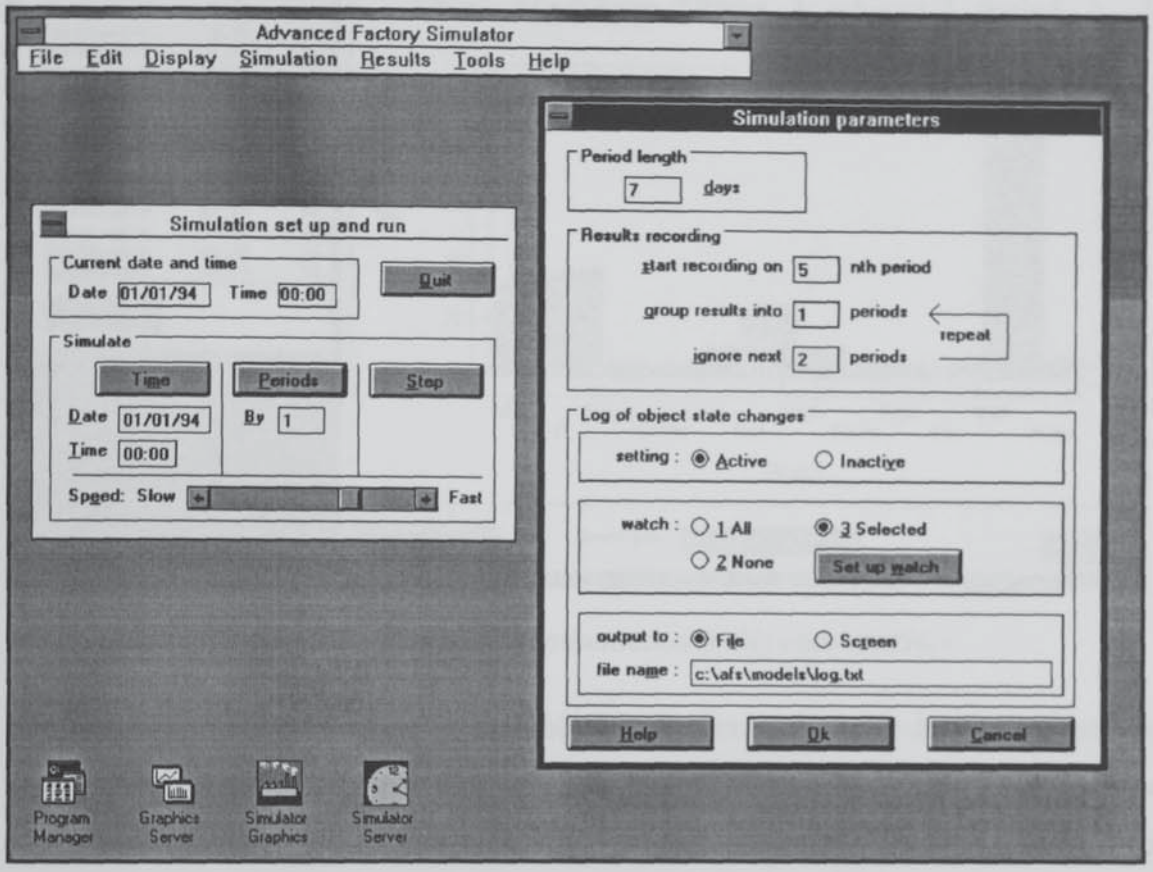


Figure 9.5. Setting up the results recording pattern

The simulation results for objects can be viewed in several ways:

- by class for a single period;
- by object for all periods;
- by a selection of objects and a selection of measures for a selection of periods.

Graphs can be created automatically by selecting one or more rows of results. An example of the results dialogs is shown in Figure 9.6. Automating the preparation of graphs frees the user of the task of preparing the graphs as well as allows results to be viewed more easily. Both the numeric and graphical results can be printed.

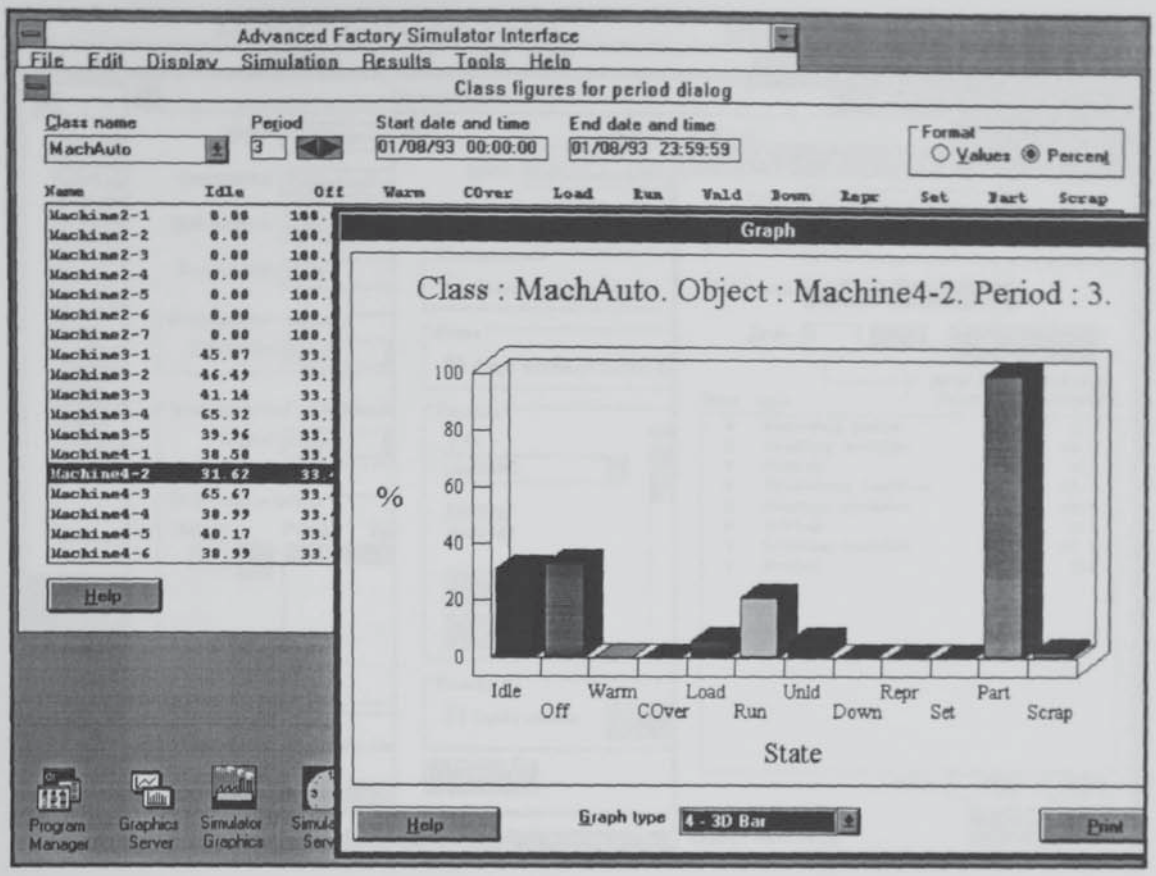


Figure 9.6. The results presentation in the user-interface

The mechanism described here is available for all classes for recording and displaying results that can be aggregated. For example, this mechanism is able to show the total time a machine spent idle or the total number of parts processed. This mechanism is not able to show sequences. For example the sequence of an operator loading, running, walking, transporting, loading, etc. cannot be shown. Due to the specific nature of such results alternative forms must be sought. There are two approaches. The first is to analyse the state change log to ascertain the history of an object's activities. The second is to use a cycle recording mechanism similar to the Nagare results for an operator, see Figure 9.7.

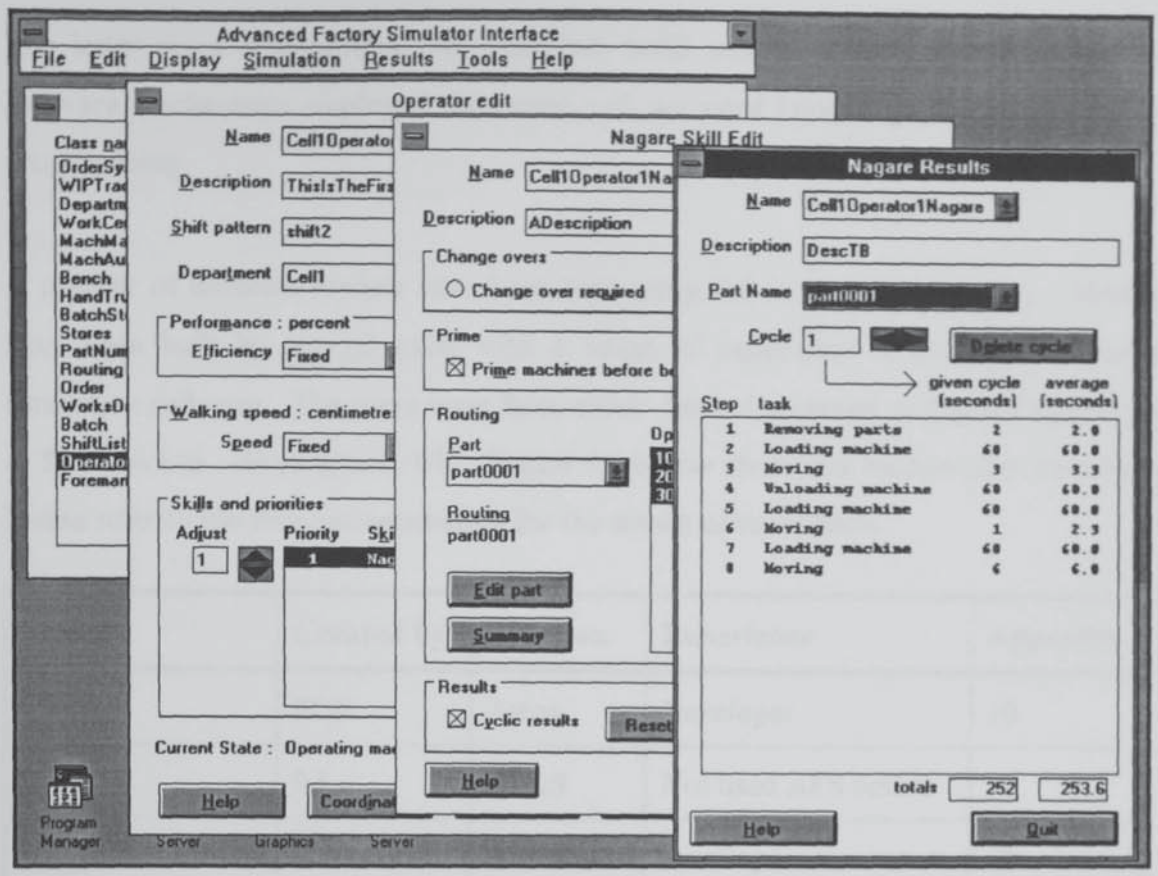


Figure 9.7. Recording the activities of an operator on a Nagare line

9.6. Range of models and users

It was noted in the previous chapter that other individuals have been involved in developing new functionality and adding it to the AFS environment. Developers have added new application classes to the simulator and new dialogs to the user-interface. Such additions have extended the scope of modelling of AFS. This therefore shows that AFS is flexible to meet new modelling challenges. To show that AFS can simultaneously provide ease of use models must be built. If models can be built easily then this demonstrates the ability to develop an easy to use simulation tool that is not constrained by its initial functionality.

To be able to claim that AFS is easy to use it must be possible to build models:

- relatively quickly;
- using data only;
- without knowledge of the underlying software.

The latter point is important. If users can build models without understanding the software mechanisms employed then users will not need knowledge and experience of programming.

A number of different models have been built using the existing functionality. Models have been built by several users with a range of experience of building and using simulation software. The users were from either Aston University or Lucas Engineering & Systems Ltd., Birmingham, UK. Figure 9.8 below shows the models that were built. Please refer to the relevant appendices for the details of the models.

Model	Created by	Location	Experience	Appendix
Simple	PDB	Aston	Developer	10
Lucas	RJ	LE&S	Not used AFS before	11
Nagare	PDB	Aston	Developer	12
Assembly	PDB	Aston	Developer	13
Yoke flange	IEW	Aston	Not used AFS before	14

Figure 9.8. AFS models and model builders

The ability of other users to build simulation models demonstrates the ease of use of AFS. The ability to build models easily shows that AFS can be used just as easily as current commercial simulators. The previous chapter detailed the ability to extend the functionality of AFS. It has therefore been shown that it is possible to construct a simulator that is not constrained to a particular domain.

10. Discussion of the Design and its Potential

This chapter discusses the potential of the Advanced Factory Simulator (AFS), some of which have already been mentioned. This chapter will discuss those aspects in more detail as well as showing that there are more radical opportunities to exploit.

10.1. Potential functionality

The ability to extend the range of functionality has been demonstrated. Extensions are not just confined to variant classes such as a new type of machine or storage area but also have been shown to include radically different functionality. For example, production planning and control functionality has been added. Within the constraints of the development language compiler (Borland Pascal), any functionality can be added. The range of functionality is chiefly constrained by the number and variety of classes developers are prepared to add.

Increasing the range of functionality will increase the potential range of application of AFS. The functionality can be added without adding to the complexity of the core of AFS and without compromising ease of use. In the limit therefore, AFS possesses the ease of use of simulators whilst providing the range of application typically associated with simulation languages.

10.2. Multiple levels of modelling

Multiple levels of modelling were discussed earlier as a means by which available functionality can be matched to user requirements. Provision of multi-level modelling in AFS increases the number of potential users as well as eases the task of building abstract models. For example, more abstract modelling levels may match the requirements of certain people who could not make use of the most detailed level of modelling.

The use of objects as building blocks for more aggregate modelling levels has been described. The message passing requirements of each of the objects can be used to guide the aggregation process. The use of a bottom up approach to achieving multiple levels of modelling could result in poor abstractions of the real world being suppressed. This is

in contrast to the top down approach whereby any deficiencies in abstractions could be amplified.

The nature of the architecture of AFS keeps the application specific classes independent of the supporting classes. Hence the multiple levels of detail are constrained only by the existing functionality and not by the design or implementation of the supporting architecture.

There are two key issues concerning building classes to be used for abstract levels of modelling. The first is the data requirements; the data should be both appropriate to the level of modelling and understandable to the user. The use of abstract factors for adjusting the performance should be avoided wherever possible. The second issue is the dynamic performance; an abstract object should exhibit roughly the same performance as the collection of detailed objects it represents. Assessment of the dynamic performance of the group of objects can be used as a guide to dynamic performance the single, aggregate object should be constructed to give. This would suggest there is potential for an extensive library of aggregate classes. For example, there could be numerous cell classes depending on the configuration of machines, number of operators and type of control system.

Since the levels of modelling are based on the application classes then as new application classes are added the potential for different aggregate classes increases. Even if the new detailed application classes are only subtly different to existing ones they can still be used to guide the construction of less detailed classes. This will further increase the degree by which functionality can be tailored to user requirements.

10.3. Potential of user-interface

The design of the user-interface has been presented. The technique by which new dialogs can be introduced has been shown and achieved in practice. Potentially, within the constraints of the development language (Visual Basic) any new dialog can be added. These dialogs may be added to support a new application class or to provide a tailored interface for a particular user

The concept of system dialogs was discussed earlier. New systems dialogs can be developed and easily added to the user-interface either to suit a particular user's requirements or for supporting general manufacturing scenarios, such as constructing a cell or a flow line. The use of this approach has the potential to increase the speed and ease by which users can build models.

The user-interface is able to access any information present in the simulator application For example, the user-interface can be used to view and edit the set up of a particular machine. In the case of the results, the user-interface retrieves all the results for a particular object for a particular period. This information is then passed to a separate graph presentation package, Graphics Server. The ability of the user-interface to retrieve information and transfer it to other applications is potentially very powerful. In the short term a statistical analysis package could be integrated with AFS. In the longer term, the user-interface could be responsible for passing data between the simulator and other applications such as a real world manufacturing database.

10.4. Potential of object-oriented architecture

Object-oriented analysis and design covers the whole of the AFS system and not just the application classes such as 'machine', 'operator' and 'truck'. Application of object-oriented principles to the whole of AFS has resulted in an architecture that has the potential for powerful extensions.

10.4.1. Distribution

The main functions of AFS (user-interface, graphics, simulation and results) have been implemented as different applications. All applications are executed in the Microsoft Windows environment. Microsoft's Windows for Workgroups (WFW) is an extension to standard Windows. WFW enables peer-to-peer networking in which each computer is able to communicate directly with any other computer on the same network. WFW also supports dynamic data exchange (DDE) across the network. The use of WFW with AFS would enable applications to be placed on different computers, see Figure 10.1.

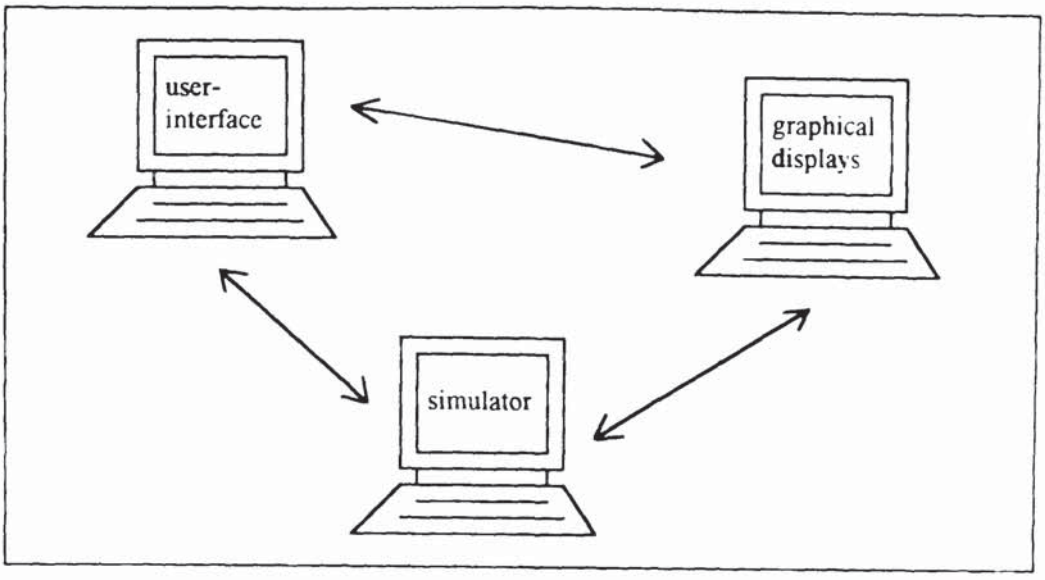


Figure 10.1. Distribution of AFS across network

The distribution of applications could be exploited to provide a reduction in simulation run time by providing simultaneous running of the model. For example, the graphics could be executed on one computer and the simulator on another. Two major hurdles exist to the exploitation of the potential time savings. Firstly mechanisms must be developed to allow the simulator to continually supply the graphics with information without waiting for acknowledgements; waiting for acknowledgements would slow the simulator down. Secondly efficient message passing mechanisms for network communications would have to be developed.

The user-interface is able to request information from the simulator. The simulator returns this information without knowledge of the application that requested the information. Potentially any application can therefore communicate with the simulator. There is potential for multiple applications to link to the simulator to request and edit data. Ignoring the issue of simultaneous access to the data, the AFS system has the potential to support different user-interfaces on different terminals of a network all viewing the same model. Similarly the simulator is able to support multiple displays. The simulator thus has the potential to support different displays on different terminals of a network. Different users could therefore view the same model in different ways, see Figure 10.2. Using this approach one user could view a manufacturing system as the detailed behaviour of a machines, operators and trucks, another user could view just the

movement of materials whilst yet another user could view the system as a collection of cells.

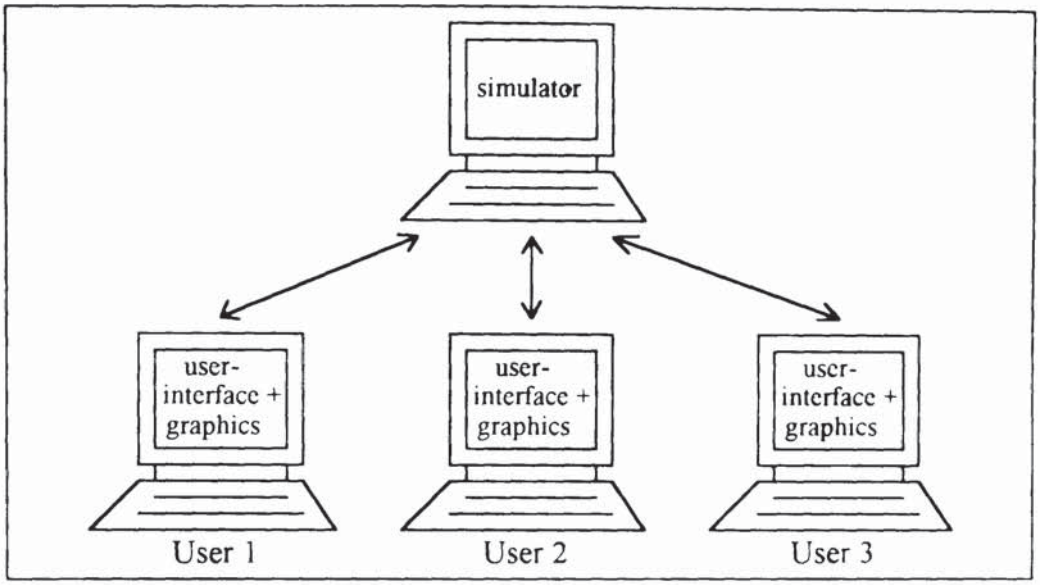


Figure 10.2. Multiple views of the same model

10.4.2. Integration with commercial systems

The ability to integrate other applications with the AFS system has been described. These applications could be described as utility applications; they provide a means by which models can be built and evaluated more easily but they do not contribute to the modelling functionality. A potentially very powerful software system could exist if applications could be integrated with the modelling functionality in AFS (see Ball & Love, 1993). This concept was described in Love et al. (1992) in which commercial MRP and accounting packages can be integrated with a shop floor simulation model. The commercial packages are not just providers of information to the shop floor model but are involved in time phased feed back mechanisms. In this situation therefore the model would be distributed across several applications.

The extension of functionality within AFS can be achieved by adding a new class and routines that manage objects of that class. It is possible to add new classes that can use and be used by existing classes without modifying the existing classes. Within AFS, classes are developed in such a way that the implementation is hidden. Other classes can interact with them but have no knowledge of their internal mechanisms. The internal

mechanisms can therefore be either Pascal software or links to other applications. See Figure 10.3.

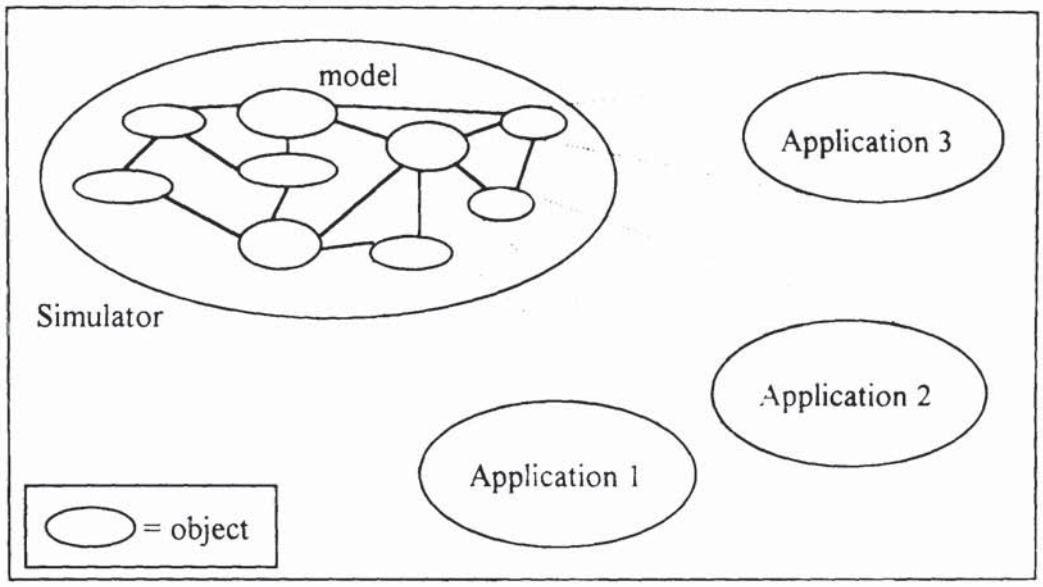


Figure 10.3. Integration of other applications as elements of model

The applications could replace one or more of the existing objects in a similar way to the multi-level modelling mechanism described above. Potentially any applications could be used provided they replace functionality in the simulator adequately.

An example application of this approach is the integration of a commercial Material Requirements Planning (MRP) system with a manufacturing system model (see Love et al., 1992). Using this approach would enable a combined manufacturing and planning and control system model to be created. The effects of interaction between the two areas could be modelled. This could be achieved without the need to be involved in extensive control system software development.

Clarke (1988) discusses the benefits of integrating real systems with simulation models:

- reduced validation problems: it is the real system so it must be valid;
- ease of data management: the data is readily available;
- familiar user-interface: the user could be in the real world;

- direct translation of model findings into new policies is possible: the model and real system parameters are identical.

10.4.3. Whole Business Simulation

Love et al. (1992) discuss the concept of simulating the whole business using a combination of real systems and simulation models. The Whole Business Simulator (WBS) would enable large models to be created. These models would include elements from different parts of the business at appropriate levels of detail. The elements would typically include:

- customer model;
- design function;
- production engineering;
- material requirements planning;
- suppliers;
- manufacturing operations;
- accounting system.

The approach would allow design and policy changes to be evaluated in terms of effect on the whole business. This would reduce the possibility of decisions being made that would optimise parts of the business at the expense of the whole. An overview of the scope of WBS is shown in Figure 10.4.

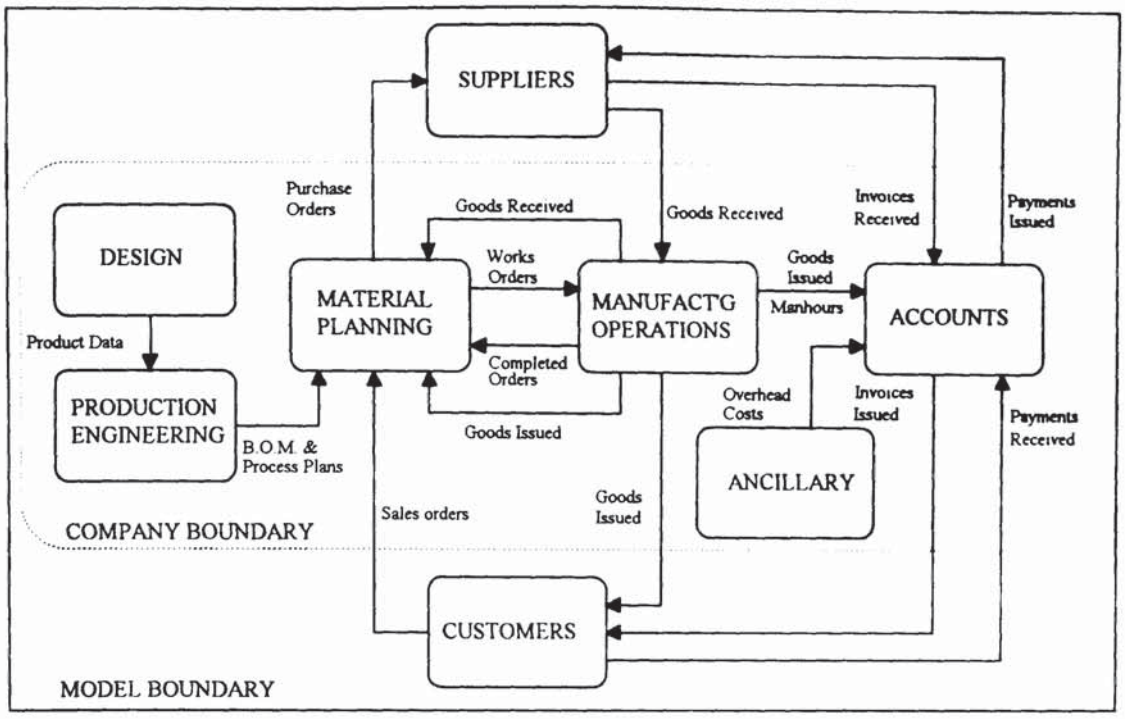


Figure 10.4. Key Transactions between Whole Business Simulation Elements (adapted from Love et al., 1992)

WBS has been implemented using ATOMS (Bridge, 1990) as a key element. Elements such as the accounting system and material planning and control system are implemented using commercial packages. Whilst ATOMS is able to provide support for certain types of manufacturing systems its scope is limited and modification is difficult. Also the use of commercial software packages to represent the various business elements prevents the detailed modelling of human activities within those elements.

AFS has been shown to support the introduction of new functionality and has the potential for integration with real systems as described above. It therefore has the potential to provide an improved modelling element to WBS. Further, AFS could provide the architecture for modelling the whole business. Therefore whilst AFS has been presented as an expandable manufacturing simulator it could be developed to form a whole business simulator.

Currently there are a number of barriers to developing AFS to support whole business simulation:

- current functionality is confined to the shop floor;

- there are problems with communicating with applications not designed for integration;
- distribution and integration of AFS is as yet untested.

Extensions to the current functionality will be provided by members of the Department of Mechanical and Electrical Engineering in the near future (see Boughton & Love, 1994; Jackson & Love, 1994). The issue of integrating applications with simulation software is discussed by Love et al. (1992) and Ball & Love (1992). Whilst the aspects of distribution and integration have been considered during the design and implementation of AFS, mechanisms have not as yet been implemented and tested.

10.5. AFS: An expandable manufacturing simulator

It has been argued that the concept of manufacturing simulators best meets the needs of manufacturing engineers; assessment of detailed dynamics for design or operational purposes requires the use of simulation techniques. Analysis of current simulators and other simulation software has identified a number of deficiencies, the key deficiency being the compromise between ease of use and potential range of application.

It has been argued that object-oriented techniques could be employed to overcome the deficiencies in current simulation software. Use of object-oriented techniques could be used to develop a manufacturing simulator that is easy to use and yet not confined to the application area for which it was initially designed.

A manufacturing simulator (AFS) with such properties has been designed and implemented. It has been shown that individuals (other than the original software developer) can add new functionality and create simulation models. Individuals were drawn from both academia and industry to build models.

The potential for AFS has been described. It has the potential for modelling areas not typically modelled used simulation software. It has been shown that AFS could be used to provide an architecture for modelling the whole business.

10.6. Conclusions

The process of research, design and development of the Advanced Factory Simulator (AFS) enabled a number of general conclusions to be made. Conclusions can be made on the object-oriented development process undertaken, the simulator functionality and the architecture that supports it.

10.6.1. Conclusions on object-oriented development process

The development of AFS took a considerable time due to the need to understand and use object-oriented techniques and software. This work has confirmed reports from current literature of the considerable learning curve associated with using object-oriented techniques. This arises from the need to adopt an object-oriented mind-set as well as understand the programming syntax and techniques.

Limitations in current object-oriented development environments were noted. For example, constraints on the development of large programs in excess of 60,000 lines of code were uncovered in the Borland Pascal compiler. Also in the development of a user-interface a decision was made to use Visual Basic, despite its limited support for object-oriented concepts, since other environments did not offer speed and ease of use.

In the development of the AFS system it was noted that, despite initial analysis and design effort, the initial implementation was often unsatisfactory and iteration was required. This iteration was rarely a continuous effort and was typically triggered after intervals of lengthy development of other parts of the software. Three or four iterations of some code over the whole software development cycle were not uncommon. The requirement for this many iterations is consistant with reports from current literature.

The use of object-oriented techniques has brought considerable benefits to the design and operation of the AFS system, in part helped by the iterative process other developers have been able to add to the system in a robust manner without having to understand the detailed operation of the whole software. Reuse of existing code is commonplace.

10.6.2. Conclusions on the functionality of AFS

In the development of AFS a conscious decision was made to separate the functionality and architecture: the functionality would be used to build models whilst the architecture supports that functionality in a simulation environment.

The range of functionality was sufficient to enable the construction of a range of batch and line manufacture involving discrete parts. Models were built using data only and involved model builders other than the author.

The process of development of the functionality for shop floor modelling demonstrated that object-oriented principles could be used: objects could be created that had a correspondence with the real world and could be grouped to form a coherent model. This is not inconsistent with the findings of other researchers in the field of object-oriented simulation software development.

10.6.3. Conclusions on the architecture of AFS

The architecture forms a substantial part of the software, providing creation and editing of models, time advance, graphical displays, results collection and display and model load and save. Considerable effort and discipline was required to keep the architecture and functionality separate. This additional effort has been offset by the potential power of the architecture, namely the provision for supporting an expanding range of functionality.

Since architecture is independent of the functionality there are no restrictions on the type of functionality that can be added. During the development process the architecture supported the addition of new machines, operator skills and other classes related to the shop floor. Following this other software developers have added new functionality without the need to modify the architecture. This expansion has been possible despite the additions being in application areas other than the shop floor, namely office modelling and production planning and control. It can be concluded that the Advanced Factory Simulator can support the addition of new functionality for modelling.

The ability to combine ease of use with potential flexibility of application within a single simulation system has been demonstrated. Under the approach described, engineers can build models using data only whilst software developers can generate successive versions of the software with an increasingly wider range of functionality.

References

Adiga, S. 1989. "Software modelling of manufacturing systems: a case for an object-oriented programming approach." *Annals of Operations Research*, 17: 363-377.

Afzulpurkar, S., Huq, F. & Kurpad, M. 1993. "An Alternative Framework for the Design and Implementation of Cellular Manufacturing." *International Journal of Operations and Production Management*, 13 (9): 4-17.

Ball, P.D. 1994. *Advanced Factory Simulator Development Manual*. Report No. IDM 0794/1. Department of Mechanical and Electrical Engineering, Aston University, Birmingham, U.K..

Ball, P.D. & Love, D.M. 1992. "Design of an extensible manufacturing simulator." *Simulation and AI in Computer Aided Techniques, Proceedings of the European Simulation Symposium*, Dresden, Germany: Society for Computer Simulation, November: 491-495.

Ball, P.D. & Love, D.M. 1993. "A Prototype Advanced Factory Simulator." *Advances in Manufacturing Technology VII. Proceedings of the Ninth National Conference on Manufacturing Research*, Bath, U.K.: 77-81.

Banks, J. 1993. "Software for simulation." *Proceedings of the Winter Simulation Conference*, Los Angeles, California, U.S.A.: Society for Computer Simulation: 24-33.

Banks, J., Aviles, E., McLaughlin, J.R. & Yuan, R.C. 1991. "The Simulator: New Member of the Simulation Family." *Interfaces*, 21 (2): 76-86.

Basnet, C.B., Farrington, P.A., Pratt, D.B., Kamath, M., Karacal, S.C. & Beaumariage, T.G. 1990. "Experiences in developing an object-oriented modeling environment for manufacturing systems." *Proceedings of the Winter Simulation Conference*, New Orleans, Louisiana, U.S.A.: Society for Computer Simulation, December: 477-481.

Bhuskute, H.C., Duse, M.N., Gharpure, J.T., Pratt, D.B., Kamath, M. & Mize, J.H. 1992. "Design and implementation of a highly reusable modeling and simulation framework for discrete part manufacturing systems." *Proceedings of the Winter Simulation Conference*: Arlington, Virginia, U.S.A.: Society for Computer Simulation, December: 680-688.

Black, J.T. 1991. *The Design of the Factory with a Future*. New York, U.S.A.: McGraw-Hill.

Booch, G. 1983. "Object-oriented design." Chapter 5 in *Software Engineering With Ada*. Menlo Park, California, U.S.A.: Benjamin/Cummings Publishing Co. Inc.

Booch, G. 1991. *Object oriented design with applications*. Redwood City, California, U.S.A.: Benjamin/Cummings Publishing Company, Inc.

Borgen, E. & Strandhagen, J.O. 1990. "An object oriented tool based on discrete event simulation for analysis and design of manufacturing systems." *Optimization of Manufacturing Systems Design : International Conference Proceedings*: North-Holland Publishing Company, August: 195-220.

Boughton, N.J. & Love, D.M. 1994. "An object-oriented approach to modelling manufacturing planning and control systems." *Proceedings of the National Conference on Manufacturing Research*, Loughborough, U.K.: 426-430.

Bravoco, R.R. & Yadav, S.B. 1985. "A methodology to model the dynamic stucture of an organisation." *Information Systems*, 10 (3): 299-317.

Bridge, K. 1990. *The application of computerised modelling techniques in manufacturing system design*. Ph.D. Thesis, Aston University, Birmingham, U.K.

Busby, J.S. & Williams, G.M. 1993. "The value and limitations of process models to describe the manufacturing organisation." *International Journal of Production Research*, 31 (9): 2179-2194.

CACI. 1992. *Reference Manual for SIMFACTORY II.5 and SIMPROCESS*. La Jolla, California, U.S.A.: CACI Products Company.

Carrie, A. 1988. *Simulation of Manufacturing Systems*. Chichester, U.K.: John Wiley & Sons Ltd.

Chaharbaghi, K. 1991. "Using Simulation to Solve Design and Operational Problems." *International Journal of Operations and Production Management*, 10 (9): 89-105.

Christy, D.P. & Watson, H.J. 1983. "The Application of Simulation: A Survey of Industry Practice." *Interfaces*, 13 (5): 47-52.

Clarke, S.R. 1988. *Aspects of the design and control of manufacturing systems subject to demand uncertainty*. Ph.D. Thesis, Aston University, Birmingham, U.K.

Coccari, R.L. 1989. "How Quantitative Business Techniques Are Being Used." *Business Horizons*, 32 (4): 70-74.

Collins, N. & Watson, C.M. 1993. "Introduction to Arena." *Proceedings of the Winter Simulation Conference*, Los Angeles, California, U.S.A.: Society for Computer Simulation, December: 205-212.

Commission of European Communities. 1991. *Production Automation*. pp. 10-33 to 10-40 in *Panorama of EC Industry - 1991-1992*. Office for Official Publications of the European Communities, Brussels, Luxembourg.

Conway, R.W., Johnson, B.M. & Maxwell, W.L. 1959. "Some problems of digital systems simulation." *Management Science*, 6: 92-110.

Conway, R. & Maxwell, W.L. 1986. "XCELL: A cellular, graphical factory modelling system." *Proceedings of the Winter Simulation Conference*, Washington, DC, U.S.A.: Society for Computer Simulation: 160-163.

Cox, B.J. & Novobilski, A. 1991. *Object-oriented programming: an evolutionary approach*. Reading, Massachusetts, U.S.A.: Addison-Wesley Publishing Company.

Coyle, R.G. 1977. Management System Dynamics. London, U.K.: John Wiley & Sons Ltd.

Crookes, J.G. 1987. "Generators, Generic Models and Methodology." *Journal of the Operational Research Society*, 38 (8): 765-768.

Dahl, O.J. & Nygaard, K. 1966. "SIMULA - an ALGOL - Based Simulation Language." *Communications of the ACM*, 9 (9): 671-678.

Dietrich, B.L. & March, B.M. 1985. "An application of a hybrid approach to modeling a flexible manufacturing system." *Annals of Operations Research*, 3: 263-276.

Forrester, J.W. 1958. "Industrial Dynamics: a major breakthrough for decision makers." *Harvard Business Review*, 36 (4): 37-66.

Glassey, C.R. & Adiga, S. 1990. "Berkeley Library of Objects for Control and Simulation of Manufacturing (BLOCS/M)." pp. 1-27 in *Applications of Object-Oriented Programming*, Ed. Pinson, L.J. & Wiener, R.S. New York, U.S.A.: Addison-Wesley Publishing Company.

Gooden, D.I. 1988. *The development of a total system simulation modelling approach for management decision support*. M.Phil. Thesis, Aston University, Birmingham, U.K.

Graham, I. 1994. *Object-Oriented Methods*. Wokingham, U.K.: Addison-Wesley Publishing Company.

Haider, S.W., Noller, D.G. & Robey, T.B. 1986. "Experiences with Analytic and Simulation Modeling for a Factory of the Future Project at IBM." *Proceedings of the Winter Simulation Conference*, Washington, DC, U.S.A.: Society for Computer Simulation: 641-648.

Harrell, C.R. & Leavy, J.J. 1993. "ProModel Tutorial." *Proceedings of the Winter Simulation Conference*, Los Angeles, California, U.S.A.: Society for Computer Simulation, December: 184-189.

Harrell, C.R. & Tumay, K. 1991. "ProModel Tutorial." *Proceedings of the Winter Simulation Conference*, Phoenix, Arizona, U.S.A.: Society for Computer Simulation, December: 101-105.

Hood, L., Laughery, K.R. & Dahl, S. 1993. "Fundamentals of Simulation Using Mirco Saint." *Proceedings of the Winter Simulation Conference*, Los Angeles, California, U.S.A.: Society for Computer Simulation, December: 218-222.

Huettner, C.M. & Steudel, H.J. 1992. "Analysis of a manufacturing system via spreadsheet analysis, rapid modelling, and manufacturing simulation." *International Journal of Production Research*, 30 (7): 1699-1714.

Hurrion, R.D. & Secker, R.J.R. 1978. "Visual Interactive Simulation. An Aid to Decision Making." *OMEGA. The International Journal of Management Science*, 6 (5): 419-426.

Jackman, J. & Johnson, E. 1993. "The Role of Queueing Network Models in Performance Evaluation of Manufacturing Systems." *Journal of the Operational Research Society*, 44 (8): 797-807.

Jackson, A. T. & Love, D.M. 1994. "Evaluating the impact of engineering decisions on the support departments of a manufacturing business". *Proceedings 11th Conference of the Irish Manufacturing Committee (IMC)*. Belfast, U.K.: 729-736.

King, C.U. & Fisher, E.L. 1986. "Object-oriented shop-floor design, simulation, and evaluation." *Fall Industrial Engineering Conference Proceedings*, Boston, Massachusetts, U.S.A.: Institute of Industrial Engineers, December: 131-137.

Kiran, A.S., Schloffer, A. & Hawkins, D. 1989. "An integrated simulation approach to design of flexible manufacturing systems." *Simulation*, 52 (2): 47-52.

Kreutzer, W. 1986. *System simulation: programming styles and languages* Wokingham, U.K.: Addison-Wesley Publishing Company.

Law, A.M. & Kelton, W.D. 1991. *Simulation Modeling and Analysis* New York, U.S.A.: McGraw-Hill Book Company.

Lenz, J.E. 1989. "The MAST Simulation Environment." *Proceedings of the Winter Simulation Conference*, Washington, D.C., California, U.S.A. Society for Computer Simulation, December: 243-248.

Love, D.M. 1980. *Aspects of design for a spare parts provisioning system*. Ph.D. Thesis, Aston University, Birmingham, U.K.

Love, D.M. & Bridge, K. 1988. "Specification of a computer simulator to support the manufacturing system design process." *3rd International Conference on Computer-Aided Production Engineering*, Ann Arbor, Michigan, U.S.A., June: 317-323.

Love, D.M., Barton, J.A. & Cope, N. 1992. "Whole Business Simulation and Engineering Applications." *Proceedings of the 8th International Conference on Computer-Aided Production Engineering*, Edinburgh, U.K.: 166-173.

Lucas Engineering & Systems Ltd. 1989. *The Lucas Manufacturing Systems Engineering Handbook*. Birmingham, U.K.: Lucas Engineering & Systems Ltd.

Mansfield, E. 1993. "The Diffusion of Flexible Manufacturing Systems in Japan, Europe and the United States." *Management Science*, 39(2): 149-159.

Mejabi, O. & Wasserman, G.S. 1992. "Simulation constructs for JIT modelling." *International Journal of Production Research*, 30 (5): 1119-1135.

Mize, J.H. & Pratt, D.B. 1991. "A comprehensive object oriented modeling environment for manufacturing systems." *Joint International Conference on Factory Automation and Information Management*: Society for Manufacturing Engineers, March: 250-257.

Monden, Y. 1981. "A Special Report: Adaptable Kanban System Helps Toyota Maintain Just-In-Time Production." *Industrial Engineering*, May: 29-46.

Mujtaba, M.S. 1994. "Simulation modelling of a manufacturing enterprise with complex material, information and control flows." *International Journal of Computer Integrated Manufacturing*, 7 (1): 29-46.

Najmi, A. & Stein, S.J. 1989. "Comparison of conventional and object-oriented approaches for simulation of manufacturing systems." *IIE Integrated Systems Conference and Society for Integrated Manufacturing Conference Proceedings*, Atlanta, Georgia, U.S.A.: Institute of Industrial Engineers, November: 471-476.

Nof, S.Y. 1994. "Critiquing the potential of object orientation in manufacturing." *International Journal of Computer Integrated Manufacturing*, 7 (1): 3-16.

O'Keefe, R.M. & Haddock, J. 1991. "Data-driven generic simulators for flexible manufacturing systems." *International Journal of Production Research*, (29) 9: 1795-1810.

Ormsby, A. 1991. "Object-Oriented Design Methods." Chapter 8 in *Object-Oriented Languages, Systems and Applications*. London, U.K.: Pitman.

Parnaby, J. 1986. "The design of competitive manufacturing systems." *International Journal of Technology Management*, 1 (3/4): 385-396.

Paul, R. & Chew, S.T. 1987. "Simulation Modelling Using an Interactive Simulation Program Generator." *Journal of the Operational Research Society*, 38 (8): 735-752.

Pegden, C.D. 1985. *Introduction to SIMAN*. State College, Pennsylvania, U.S.A. Systems Modeling Corporation.

Pidd, M. 1992a. "Guidelines for the design of data driven generic simulators for specific domains." *Simulation*, 59 (4): 237-243.

Pidd, M. 1992b. "SKIM: a Simulation Sketchpad for Plant Design." *Journal of the Operational Research Society*, 43 (12): 1121-1133.

Pidd, M. 1992c. *Computer simulation in management science*. John Wiley & Sons Ltd, Chichester, U.K.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. & Lorensen, W. 1991. *Object-oriented modeling and design*. Englewood Cliffs, New Jersey, U.S.A.: Prentice-Hall, Inc.

Shewchuk, J.P. & Chang, T.C. 1991. "An approach to object-oriented discrete-event simulation of manufacturing systems." *Proceedings of the Winter Simulation Conference*, Phoenix, Arizona, U.S.A.: Society for Computer Simulation, December 302-311.

Shimizu, M. 1991. "Application of an integrated modelling approach to design and analysis of manufacturing systems." *Advanced Manufacturing Engineering*, 3, January 1991: 3-17.

Simulation Study Group, 1991. *Simulation in U.K. Manufacturing Industry*. The Management Consulting Group, University of Warwick Science Park, Coventry, CV4 7EZ, U.K.

Solberg J.J. & Nof, S.Y. 1980. "Analysis of Flow Control in Alternative Manufacturing Configurations." *Journal of Dynamic Systems, Measurement, and Control*, 102: 141-147.

Strandhagen, J.O. 1987. "Expert knowledge in object oriented simulation of manufacturing systems." *Proceedings of 3rd International Conference on Simulation in Manufacturing*, Turin, Italy: IFS (Conferences) Ltd: 23-30.

Suri, R. & Diehl, G.W. 1985. "MANUPLAN: A Precursor to Simulation for Complex Manufacturing Systems." *Proceedings of the Winter Simulation Conference*, San Francisco, California, U.S.A.: Society for Computer Simulation, December: 411-420.

Szymankiewicz, J., McDonald, J. & Turner, K. 1988. *Solving business problems by simulation*. London, U.K.: McGraw-Hill Book Company.

Taylor, D.A. 1993. *Object-Oriented Technology: A Manager's Guide*. Reading, Massachusetts, U.S.A.: Addison-Wesley Publishing Company, Inc.

Thompson, W.B. 1993. "A Tutorial for Modeling with the WITNESS Visual Interactive Simulator." *Proceedings of the Winter Simulation Conference*, Los Angeles, California, U.S.A.: Society for Computer Simulation, December: 228-232.

Townsend, M.A. & Lamb, T.W. 1991. "Plant Characterization and Problem Identification Using Plant Historical Data in a Simulation Environment." *Journal of Manufacturing Systems*, 10 (1): 41-53.

Ulgen, O.M. & Thomasma, T. 1990. "SmartSim: An object oriented simulation program generator for manufacturing systems." *International Journal of Production Research*, 28 (9): 1713-1730.

Wang, W., Popplewell, K. & Bell, R. 1993. "An integrated multi-view system description approach to approximate factory modelling." *International Journal of Computer Integrated Manufacturing*, 6 (3): 165-174.

Wight, O. 1983. *The Executive Guide to Successful MRPII*. Englewood Cliffs, New Jersey, U.S.A.: Prentice-Hall, Inc.

Wu, B. 1992. *Manufacturing systems design and analysis*. London, U.K.: Chapman & Hall.

Appendix 1: Software suppliers

The following table details the software used for the development of the Advanced Factory Simulator.

Application	Software	Version	Manufacturer
Operating Environment	Windows & Windows for Workgroups	3.1	Microsoft Corporation. UK Office: Winash, Wokingham, Berks, RG11 1TP
Simulator & Graphics	Borland Pascal with Objects	7.0	Borland International, Inc 1800 Green Hills Road, P.O. Box 66001, Scotts Valley, CA, USA.
User-interface	Visual Basic	3.0	Microsoft Corporation. UK Office: Winash, Wokingham, Berks, RG11 1TP.
Results Presentation	Graphics Server	2.2	Bits Per Second Ltd, 14 Regent Hill, Brighton, BN1 3ED, UK.
Date & Ttime Utilities	Object Professional	1.12	Turbo Power Software, P.O. Box 66747, Scotts Valley, CA, USA.
Results Analysis (proposed)	SPSS for Windows	6.0	SPSS Inc, 444 North Michigan Avenue, Suite 3000, Chicago, IL, USA.

Appendix 2: Hardware and software operating requirements

1. Hardware

This section details the hardware requirements for developing and using the Advanced Factory Simulator (AFS) system. The requirements listed here represent the minimum. Exceeding these requirements will result in easier, possibly faster, use.

Hardware type	IBM compatible personal computer
Processor specification	486, 25MHz
Video	Super VGA
Memory	4MB
Hard disk space - operation	2MB
Hard disk space - development*	10MB

* The hard disk space required for development is many times greater than for operation since the process of development requires the use of back up files and temporary compilation files. The figure quoted here is AFS software related; it excludes requirements for the installation and use of development environments such as Borland Pascal with Objects and Microsoft Visual Basic.

2. Software

2.1. Development requirements: commercial software

The following commercial software is required:

Application	Software
User-interface	Microsoft Visual Basic v3 Bits-Per-Second Graphics Server v2.2
Simulator	Borland Pascal with Objects v7
Graphics	Borland Pascal with Objects v7
Help system	Microsoft Word Microsoft help compiler

There are numerous AFS source files that need to be installed. The files required depend on which part of the AFS system is being modified. These include the Pascal and Visual Basic development files as well as other files such as the bitmaps.

2.2. Operating requirements: commercial software

The following commercial software is required for operation:

Software	Files
Graphics Server	gsw.exe*, gswdll.dll*, graph.vbx (* v2.2)
Visual Basic	vbrun300.dll, cmdialog.vbx, msmasked.vbx, spin.vbx, grid.vbx, gauge.vbx
Borland Pascal	ddeml.dll

These files should be placed in the "?:\windows\system" directory.

Installation of these files is not required if the development system has been installed.

2.3. Operating requirements: AFS software

The following AFS files are required for operation:

File name	Description	Directory
AfsCore.exe	Simulator application	?:\afs\system
AfsIntf.exe	User-interface application	?:\afs\system
AfsGraph.exe	Graphical display application	?:\afs\system
IconDLL.dll	Icon dynamic link library for graphics	?:\afs\system
Config.txt	Configuration file to link classes	?:\afs\system
AfsHelp.hlp	AFS help file used by Windows help	?:\afs\help
Res*.bmp	User-interface bitmaps	?:\afs\bitmaps

Appendix 3: Summary source code documentation

This appendix lists the summary documentation for the Advanced Factory Simulator Pascal source code. The documentation has been extracted from the source code files using a specially developed program. This approach has the advantage of presenting an up to date compilation as well as being quick.

The documentation is listed alphabetically according to the following categories:

- Core
- DDE (Dynamic Data Exchange)
- Display
- Factory
- Graphics
- Managers
- Random

The categories are based on the directory structure of the AFS development system.

The format used for documenting the software is as follows:

```
Unit TestUnit;
{ ------------------------------------------------- - - -
TYPE:
  CLASS     TestClass1  TestClass2  TestClass3
    or
  LIBRARY   TestLibrary
    or
  PROGRAM   TestProgram

DATES:
  CREATED    PDB    01/01/94
  UPDATED    PDB    01/01/94
  TESTED     PDB    01/01/94 TestProg.exe

SUMMARY:
  This is a summary of the various classes held within the
  unit.  Many lines may be used.  Indentation may be used but
  is not necessary.  The text must be continuous - there must
  be no blank lines.  Avoid using commas - currently commas
  disrupt the layout of the text extracted by the Visual
  Basic extraction program.  Avoid text too far to the left hand
  side as this will also disrupt the layout - use the '-----'
  line below as a guide.  This section must be followed by a
  blank line.

DETAIL:
  The format is the same as the summary section.

-------------------------------------------------------- }
```

All key words are shown in capitals. Key words can be either flush against the left hand side or preceded only by spaces. Key words must be followed by at least one space. Each section must be followed by a blank line.

The documentation that follows only includes the 'summary' section. The 'detail' section is more volatile and so reference must be made to the source files.

Summary of AFS source code : 16th July 1994

Core classes and units

Class : tAbsResult
Abstract results class. This class must be inherited by all results classes. Using the class enables results to be recorded by tSimulation descendent class objects easily. Objects of this class can be placed on results list and can be interrogated for which results period they belong to (either by period number or time and date).

Class : tActivity tActivityResults
tActivity must be inherited by all classes wishing to actively participate in the simulation world. The class provides the ability to send and receive messages and change state and advance through time. Functionality within this class enables objects to be configured with other objects (class configuration is carried out centrally by the configuration class). tActivityResults enables all results of objects of type tActivity to be recorded.

Class : tActivityMngr
The activity manager class manages functions associated with the tActivity class. This includes creation and editing and upload from text file etc.

Program : AfsDos
This program enables the AFS simulator application to run in DOS. The program enables models to be created and executed. Interface with the simulator is by program code only - no user-interface has been developed for DOS mode.

Unit : Global
This unit enables pointers to be stored globally. The tInterface class sets the various pointers which any part of the simulator can subsequently access. For example the clock can be available globally.

Unit : AfsStringTools
This unit is globally available. It provides string manipulation routines. For example strings can be split concatenated converted to numeric etc.

Class : tClassListNode tClassList
These classes combine to provide a run-time list of classes. Classes can be added removed and queried.

Class : tClassMethodNode tClassMethodList
Contains the methods for manipulation of classes. e.g. loading a list of machines from disk. Currently not used with AFS.

Class : tClock
This class is used to provide a clock object for AFS. Clock allows time and date to be set altered advanced and interrogated. The clock object has 'wrapped up' a selection of Object Professional procedures and functions found in the Opdate unit.

Class : tConfiguredClassListNode tConfiguredClassList
These two classes combine to enable a list of class message links to be stored. For example operator class can send load message to class machine.

Class : tConfigtr
The configurator is a key class in the message configuration message. This class is therefore a key class in the AFS expansion mechanism. The configurator is used for linking classes and objects at run-time. The configurator is able to update the message configuration in objects of tActivity.

Class : tConfiguredObjectListNode tConfiguredObjectList
These classes store information about object message links. The classes are used for store the information at run-time. An example of an object message link would be operator1 to send a load message to machine5. Each object of type tActivity will have a configured object list. The list will be private to the class.

Class : tEnvironmentClass
This class stores the physical data on the system being simulated. The class stores the extremes of size of the 'shop floor' (held as a rectangle since this is the only information that can be used when creating the displays. The 'area' data item is used as the shop floor may not be rectangle but the floor area may need to be known.

Class : tEvent tEventCollection tEventList
tEvent and tEventList classes are used to provide an event list for the simulation executive. It is intended that the event list will be private to the executive tEventCollection is a Borland Sorted Collection class that is currently not in use. This class stores time indexed events. Events are indexed automatically and the first event can be queried and removed. The events only hold the date and time and object reference. No event information such as machine loading is stored.

Class : tSimulationExecutive
The simulation executive is the core part of the simulator. It is able to advance objects on the event list through time. The simulation executive is able to simulate from current date and time to some specified point in the future. The simulation executive updates objects on the event list - these objects must be inherited from tSimulation (Simulatn Unit). During updates the executive will call the simulation objects' update method twice. The first time for the advance phase and the second time for the scan phase. The advance phase ensures that the object is at the correct time and state. The scan phase invites the object to interact with others. The dual approach is used to (in theory) be able to switch from next event to global updates without modification of the simulation objects.

Unit : FileManipulationTools
This unit is globally available. It provides file manipulation routines. For example the existence of files can be checked. The unit is independent of the operating environment (Windows / Dos). U_FileIO by DML/JAB has not been used : it contains DOS specific commands.

Unit : GlobalTypeDefinitions
This unit provides developer defined types for global use. For example, an object name will be of type string and x characters long.

Class : tInterface
Interface to the core simulator. An object of this class is initialised by either a DOS or a Windows program. The classes that the tInterface class uses are all DOS/Windows independent. Generally any class or code that uses this class will be dedicated to either

DOS or Windows. This class permits access to any data in the core as well as providing some global editing functions - such as deleting all objects in the world. This class is responsible for initialising and destroying the world. The term world includes the executive and the clock and the logical displays and the master object list and the results manager. In effect this class ties the whole simulator application together.

Class : tLists
This class holds the global lists in the Advanced Factory Simulator core application. The object (only one!) of this class will be able to supply objects with various lists. The class uses a Borland Collection to store objects. It is estimated that the collection can only store 10000 objects hence a list of lists may be needed at a later date to store 10000 x 10000 objects. It is therefore important that the Borland collection remains private to the class.

Unit : MasterClassManagerDeclaration
This is a key unit in the AFS expansion mechanism. This is the one and only place where existing code needs to be modified to insert a new class. Inserting a class manager here forces the compiler to compile the new class into the existing system. This unit ties together different developers' class manager registration units. New developers must create their own unit and register it here. This unit has been created to simplify the merging of code from two or more developers working concurrently. This unit can contain general class managers that are not specific to an application. For example the random class class managers should be registered here.

Class : tMessageClass
This is a key class in the AFS system. This class enables application classes to communicate without prior knowledge of the other classes. Messaging class for carrying messaging types and message sender and receiver and message content. tMessageClass is to be used for messaging between all objects regardless of their location (which will in anycase be unknown). Inter-workstation communication is handled by buffering these messages. The inter-workstation message types and message mechanism are not relevant to the developer and should not be coded explicitly within a class.

Unit : MessageConstants
This unit is used to register new message types. For example mLoad or mReceiveOrder. A message class will use the message types to enable receiving objects to distinguish between messages easily.

Class : tModelInformation
This is a model information class. It holds information about a particular as set by the user. For example it will hold full model and description. A class is used to allow new data to be easily added in the future.

Unit : DateAndTime
This library provides time and date utilities for the Advanced Factory Simulator. The unit is about 99% Object Professional. Additions and ammendments have been made to make it easier to use and to permit use in Microsoft Windows.

Class : tPhysicalMngr
The physical class manager handles the creation and editing of objects descendent from the physical class. For example the class has routines for locating an object and editing its co-ordinates

Class : tPhysical
Physical is an abstract class which must be inherited in order physically exist within the simulation world. Class provides physical position & size and the ability to display its own icon through an abstract display class.

Class : tResultsMngr
This class is central to the results mechanism. This class holds the results recording pattern as set by the user. This class has the mechanisms for instructing each object to start and stop recording. On instructions from the results manager objects will store their results into individual results objects. The results manager is then able to retrieve these results objects for display in the user interface. To be able to initiate and terminate recording at the correct time the results manager will continually place itself on the event list.

Class : tRootMngr
This class manages the functionality associated with the tRootAFS class. The class manager structure mirrors that of the classes. For every class there is an associated class manager including the abstract classes. Just as the tRootAFS is not to be used directly but as an abstract class to inherit the abstract class managers are to be inherited and not used directly. This class provides the ability to edit and retrieve information relating to tRootAFS using text messages (i.e. name and description) and provides some general functionality (list scanning - global notifications of object editing - displaying messages - etc.)

Class : tRootAFS
The root for all AFS classes. All classes wishing to benefit from the functionality present in the core architecture must inherit this class. This class provides the basic functionality:

1. naming : class name and object name and description
2. storage : object of descendent classes can be stored and retrieved from lists.

Class : tRootWBS
The root for all WBS classes. Inherits TObject to enable it to be placed on instances of TCollection (= linked list). Provides ID number for load and save purposes.

Class : tSimulationMngr
This class is similar to other classes in the class manager hierarchy. This class manages functions associated with objects descendent from the tSimulation class.

Class : tSimulation
Simulation is an abstract class which must be inherited in order physically exist and change with within the simulation world. Class provides the ability to change from state to state over time to be updated at a given point in time and to record results. Examples of states are : running - idle - etc.

Unit : StateConstants
This unit is used to register new state types. A state describes the current condition of an object. For example truck is MOVING. States used internally by objects to keep track of 'where they are' and by the graphics to ensure that the current condition is displayed.

Unit : StringResourceLoader
This unit dynamically loads strings from a resource file. This unit will only compile under Protected Mode or Windows (not DOS)/ This unit has been used to avoid using up the data segment - AFS is a very large program - all its global data must share the same 64k segment as the local heap and the stack. Within AFS the stack must be maintained around 25-30k. Global data includes statically declared items and the reason for this unit string literals. For example :

```
WRITELN('Hello');
IF 'Hello' THEN ...
```

both use up 5 + 1 bytes. Whilst such values are small they become significant when the program reaches many ten of thousands of lines of code. At March 1994 AFS was about 50000 lines of code. Some of the units that make up a significant part of that code are the class managers. The class managers use(d) a lot of string literals.

DDE classes and units

Class : tAFSDDE
AFS Simulator specific DDE Class and DDE functions. This unit provides the Dynamic Data Exchange functionality specific to the Simulator application. This includes a class to interface to the WBS DDE manager and an AFS specific call back function.

Class : tAfsGraphicsDDE
AFS Graphics Specific DDE Class and DDE functions. This unit provides the Dynamic Data Exchange functionality specific to the Advanced Factory Simulator's Graphic Display Application. This includes a class to interface to the WBS DDE manager and an AFS specific call back function.

Class : tDDE
DDE client / server class. This class combines the behaviour of a Dynamic Data Exchange (DDE) server with a DDE client. The DDE class is able to maintain multiple conversations as either / or / and client and server.

Class : tDDEClient
DDE client using DDEML (BP's DDE DLL). A DDE client class. Class contains functionality to be able to initiate and conduct a simple client - server conversation.

Class : tDDEServer
DDE server using DDEML (BP's DDE DLL). A DDE server class. Class contains the functionality for conducting a basic DDE conversation.

Display classes and units

Class : tAbstractLogicalDisplay
This is an abstract class - it must be inherited before using. This class acts as a translator between AFS drawing routines and actual drawing routines. This set up thus insulates AFS from the actual drawing libraries - classes - etc. used. Whilst this extra layer is recognised as being inefficient at run-time it offers the benefits of changing the display routines without changing the core.

Class : tAbstractPhysicalDisplay
This abstract class must be inherited before 'using'. This class acts as a translator between AFS drawing routines and actual drawing routines. This set up thus insulates

AFS from the actual drawing libraries - classes - etc. used. Whilst this extra layer is recognised as being inefficient at run-time it offers the benefits of changing the display routines without changing the core.

Class : tAfsApplication
This is the AFS windows application class. This class will be used by the simulator application. This should only be used by a simple program for use in Microsoft Windows environment.

Program : AfsCore (Windows Simulator Application)
This is the AFS simulator program for the Microsoft Windows environment. Compiling this program will result in a simulator dedicated to Windows.

Class : tAfsWindow
This class ties the tInterface class with Windows specific code. The tInterface class is independence of Windows. This class will include functionality such as the provision of an icon - the DDE messaging mechanism - the ability to display error messages and the ability to display a dialog that allows the user to stop the simulation during a run.

Class : tAbsPhyDisplayMngr
Created to fill 'hole' in core - graphics architecture. This class crudely enables the core Display Manager to issue instructions to the user-interface that are not specific to a particular display:

- displaying simulation events / state changes
- displaying messages
- etc.

This class allows the Display Manager to issue commands to the user-interface of the actual format of the user-interface.

Class : tColour tColourClass tColourList
This unit declares a number of colours that can then be matched against object states etc. The colours are held on a list which the user-interface is able to access. The use of a list enables the user-interface to be automatically updated if new colours are introduced.

Class : tDDEPhysicalDisplay
This class converts AFS logical display drawing routine instructions to AFS standard DDE (Dynamic Data Exchange) calls to a given destination. The class architecture is such that the logical displays have no knowledge of what a DDE display driver is and therefore the logical display are independent of Windows and DDE.

Class : tDisplUpdt
This class is designed for updating a given display. The class is placed on the event list and with be told to update itself. This class will then instruct its associated display to update itself. The class will continually place itself on the event list at given intervals. When taken off the event list and told to update it will tell the display to update itself and then place itself back on to the event list using the given interval.

Class : tDisplayManager
This object provides the interface between the core simulator and the simulator graphical display. The tDisplayManager class should be developed in such a way that its display methods do not make any assumptions about the actual display class.

Class : tEventLog
Class for showing events logged during the simulation to the screen. The class makes use of a dialog containing a list box. The events are added to the list box as they occur. Once the number of events in the list box reaches a limit the oldest event is removed to make room for the newest. The dialog has been created using the Resource Workshop and is managed by this class. A object of this class is created by a dedicated windows class.

Class : tLogicalDisplay
This object provides the interface between the core simulator display manager and the simulator graphical display. The tLogicalDisplay class should be developed in such a way that its display methods do not make any assumptions about the actual display class

Class : tPhyDisplayMngr
Created to fill 'hole' in core - graphics architecture. This class crudely enables the core Display Manager to issue instructions to the user-interface that are not specific to a particular display:

- displaying simulation events / state changes
- displaying messages
- etc.

This class allows the Display Manager to issue commands to the user-interface of the actual format of the user-interface.

Class : tSimulationDialog
This class enable the simulation date and time to be displayed during the simulation run. The dialog is placed here rather than the user- interface because the simulation process efficiency is of prime concern. The use of DDE messaging to display the date and time in the user- interface would slow the simulator down considerably. This dialog provides a means by which the user is able to stop the simulation at any time

Class : tStateColour tStateColourList
This unit contains classes that combine to define / store which colours are associated with which object states for display purposes. The use of a list to store the relationship enables the user-interface to query which colours are matched with which states without prior knowledge of what a state and a colour are. This use of a list also enables the user to modify which colour is used to represent which state.

Factory classes and units

Class : tAbsMaterialControl
This is an abstract material control class. This provides the abstract properties of a class used for shop floor material control. Objects of descendent classes can be used by operators etc. to decide which job to work on next. It is intended that descendent classes will include Kanban / MRP / PFC / etc.

Class : tAbsWorkingPattern
The abstract working pattern is a type primarily for use by tPerson class to control the hours that the person works. The class can be inherited to provide a variety of working patterns. The simplest is a shift pattern that provides a variety of shift patterns. Others could be developed to include absenteeism training days and holidays

Class : tAssemblySkill
This class enables operators etc. to assembly parts / batches. This class inherits and adds to the functionality of the abstract skill class. This class is a skill type class and therefore objects of this class can be placed on an operator's skill list. This can be used by an operator along with other skills such as machining and transportation skills.

Class : tAssemblyManual
Manual assembly station class. The class is 'passive' - it can only respond to external instructions (i.e. load / run / etc.) and cannot communicate with other objects or run automatically. Minimum requirements for inclusion in a simulation world:

- general core objects : display / event list / clock
- messaging related : configurator and message classes and types
- factory objects essential for initialisation : work centre
- factory objects essential for running : stores / batch / operator

Class : tAgv
Currently not in use. Modification required! This class has the ability to transport batches / tooling / etc. from one location to another. The transportation occurs under automatic control without intervention of personnel. The loading and unloading must be carried out by personnel or some form of material handling mechanism.

Class : tBatch
A batch consists of a number of identical parts a routing and an associated order reference number. The parts within the batch are all at the same stage of processing. The stage of processing is set by a copy of the routing. Batches can be split but only those at identical stages of processing can be merged.

Class : tBench
This manual bench class is for use for manual operations such as inspection. The class is used in a similar way to machines - items are taken from the input area - placed on the bench - some operation performed - items placed in output area.

Class : tBatchStore
This is a shop floor storage class. It is specifically designed to hold batches. Batches can be added / removed / queried. The batches can be queried on the basis of a number of rules: FIFO / LIFO / Shortest set up / shortest cycle time / same as last / earliest due date / etc.

Class : tCellManagementSkill
This is a skill class that can be used by the tPerson and descendent classes. It can be used with other skills such as transportation or inspection. This class will provide the user will the ability to manage cells

- releasing new and returning completed work orders
- organising of operators for production work
- organising transportation of work
- etc.

This class is primarily intended for use by the foreman class but this use is not exclusive

Class : tChangeOverSkill
This class enables operators etc. to change over machines. This class inherits and adds to the functionality of the abstract skill class. This class is a skill type class and therefore objects of this class can be placed on an operator's skill list.

Class : tContStore
Currently not in use. Testing and possibly modification is required. This class is designed specifically for storing containers. Objects of the class are to be used on the shop floor. Objects descendent from the container class can be added and removed.

Class : tScheduledMatCtrl
The class provides the functionality for an operator to control the flow of materials via schedules. An operator will query objects of this class as if they were tAbsMaterialControl objects. Objects of this class will provide the operator with the next type of batch to work on.

Class : tDepartment
The department class groups a number of resources together. The group will consist of operators / machines / trucks / etc. The group forms a boundary around these resources and enables them to be addressed collectively.

Class : tForeman (name due to change to cell leader)
This is a supervisor type class. Objects of this class are able to manage the activities of a production area (department). The class using skills (e.g. the cell management skill) for deciding what to do next. Over time the distinction between this class and the operator class has been blurred - both are person class with the ability to use a range of skills. In the future it is intended to remove both the operator and this class and have just one personnel class. Distinctions would be made on the basic of number of skill objects assigned by the user at run time.

Class : tGrpParts
Physical group of parts class. A number of identical parts. Has a pointer to the part and a quantity. The number of parts can be increased and decreased (if possible). The number of parts can be set to between 1 and a maximum (group must contain at least one part). A group of parts can be split and / or merged if possible.

Class : tHoliday
Currently not in use. Further development required. The holiday class is initially intended for used by tWorkingPattern for complex working patterns. The holiday class is similar to the shift class and its use is in many respects similar:

- one or more holidays can be held
- due to the complexities of weeks -> years the holidays are held by date and not week number.
- holidays can be added / removed / edited / queried.

Some inputs are check: e.g. holidays in excess of 2 weeks are flagged as possible errors

Class : tInspectionSkill tInspectionResult
This class enables operators etc. to carry out quality inspection. This class inherits and adds to the functionality of the abstract skill class. This class is a skill type class and therefore object of this class can be placed on an operator's skill list. This is a basic inspection skill class. Inspection must be carried out at a work bench. Travelling

inspection / inspection at a storage area is not possible. This either needs to be added or provided in an inheriting class.

Class : tAbsMachine tMachineResults
Basic machine class than allows simple a simple load - run - unload cycle. The class (will) also support breakdown - repair - set. The class is 'passive' - it can only respond to external instructions (i.e. load / run / etc.) and cannot communicate with other objects or run automatically. Minimum requirements for inclusion in a simulation world:

- general core objects : display / event list / clock
- messaging related : configurator and message classes and types
- factory objects essential for initialisation : work centre
- factory objects essential for running : stores / batch / operator

Class : tMachineAutomatic
Automatic machine class. The class can be set to load / run / unload automatically in any combination. Automatic unload will place the completed components in to the machine's output bin. It the bin cannot accept the parts the machine will 'seize' / 'block' until there is sufficient space in the output area. The class is 'passive' - it can only respond to external instructions (i.e. load / run / etc.) and cannot communicate with other objects Minimum requirements for inclusion in a simulation world:

- general core objects : display / event list / clock
- messaging related : configurator and message classes and types
- factory objects essential for initialisation : work centre
- factory objects essential for running : stores / batch / operator

Class : tMachineManual
Manual machine class. The class is 'passive' - it can only respond to external instructions (i.e. load / run / etc.) and cannot communicate with other objects or run automatically. Minimum requirements for inclusion in a simulation world:

- general core objects : display / event list / clock
- messaging related : configurator and message classes and types
- factory objects essential for initialisation : work centre
- factory objects essential for running : stores / batch / operator

Class : tNagareSkill
Nagare operator skill class. An object will use objects of this class to control a number of machines. The operator will operate a sequence of machines according to a given sequence of routing operations. A number of routing sequences for the same part can be specified. A number of parts can be specified. Class is inherited from tAbsMachineSkill and can therefore be used interchangeably with other sub classes of that abstract class.

Class : tNagareSeq
This class holds a sequence of routing operations for use (initially) by the Nagare behaviour class. The order of operations must follow those of the routing but do not have to specify sequential operations.

Unit : NewModel
New model object initialisation unit for Advanced Factory Simulator. This unit loads up new objects when the user creates a new model. Created primarily to allow PDB to create order system and WIP tracker for factory model automatically but at the same time not hardwiring the code in the main body of the core system.

Class : tOpnBasic tOpnNumber tKit
The operation number class details the location / time / etc. of an operation. Operations should only be created and destroyed within tRouting. An operation object should be private to tRouting. The operation number held is use defined. Whilst it defines the insertion point into a routing it does not correspond to the nth operation number in tRouting. The tKit class is used for assembly. Each operation object can have a series of kit objects. A kit object holds a part reference and quantity.

Class : tOperator tOperatorResults
Basic operator class than can perform the following functions :

- work according to predefined shifts
- operate a machine
- load
- run
- unload
- interact with transportation devices.
- etc.

The operator works according to skills assigned to it. The skills may include : transportation / machining / inspection / etc. The distinction between this class and the foreman / cell leader class is gradually disappearing. It is intended that both classes will be removed and replaced by a single personnel class that can be distinguished by the range of skills assigned by the user at run time.

Class : tOpnStage
Operation stage class. This class is an abstract class which all classes capable of carrying out operations (i.e. stage(s) of a routing) must inherit. Operations may include assembly stations / machines / abstract work centres / etc. Each operation stage has an input and an output storage area.

Class : tOrder tOrderList
Order class contains the part number / order reference / quantity required / due date / launch date. An 'information' class.

Class : tOrderSystem
The ordering system class acts as a temporary bridge between the order generation system and the shop floor. The class will hold live orders and direct them to the appropriate part of the shop floor.

Class : tPartNumber
Stores data about a part and makes a connection between the routing and the drawing.

Class : tPerson tPersonResults
Abstract class that is not intended to be used directly. It should be inherited by specific types such as operator / foreman /etc. This abstract class has the following properties:

- The ability to move between locations
- The ability to move other objects with self
- The ability to send and receive movement messages
- The ability to work according to a prescribed working pattern

Class : tRack
Currently not in use - more work required. Rack class is used for storing small (in size) batches. Only one batch can be stored at any one time. The held quantity is dependent on the volume of the rack and the volume of a part.

Class : tRackSystem
Currently not in use - more work required. This class is intended to hold a number of containers / trays / etc. The class inherits the properties of more a abstract storage class.

Class : tRawMatlRequisition
Raw material requisition (RMR). Used in conjunction with an order when it is released onto the shop floor. The RMR is used by the shop floor to get raw materials from stores. If the raw material is nil then it indicates that the raw material should be automatically created.

Class : tRepairSkill
This class enables operators etc. to use repair machines. This class inherits and adds to the functionality of the abstract skill class. This class is a skill type class and therefore object of this class can be placed on an operator's skill list.

Class : tRoutingBasic tRouting tOperationCollection
The routing class stores the list of operations that are carried out to produce a part. The class allows operations to be added (at any point) and deleted. The routing will be able to be modified at will - no special pre-cautions will be made - therefore users of the routing will have to make a copy. Copies can be made on request but will not be placed on the master list to avoid confusion.

Class : tScrap
Plug-in scrap class for material processors - machines / operators at manual operations / etc. The class is used by supplying it with a batch. The class manipulates the batch and returns it. No other information need be supplied.

Class : tShift tShiftList
This shift class allows an series of shifts to be created. The shift series can be set to repeat a number of times. A number of idle days added to the end of the repeated shift series. The combined set of repeated shifts series and idle days is termed the total shift cycle. The total shift cycle starts at the date set by the world clock. The total shift cycle repeats itself according to the date. The total shift cycle can be offset from the start of the simulation run.
For example:

a shift series can be created: shift 1 08:00 to 16:00
shift 2 18:00 to 02:00
the series can be set to repeat 5 times with 2 idle days
in this case the total shift cycle is calculated to be 7 days

Class : tAbsMcSkill
This is an abstract class from which all machine skill classes will be inherited. The class will provide the functionality for operators etc. to control machines. The each operator will be assigned one (or more) skill objects to use to control other activity objects. The control objects can be swapped readily. E g change the operator to operating two machines and not one. The use of skill objects will allow the operator class functionality

to be readily expanded. E.g. a Nagare skill object could be slotted in (hopefully) without modifying the operator code.

Class : tAbsSkill tSkillResult
This is an abstract class from which all operator skill classes will be inherited. The class will provide the functionality for an operator to control machines / trucks / etc. The each operator will be assigned one (or more) skill objects to use to control other activity objects. The skill objects can be swapped readily. E.g. change the operator to operating two machines and not one. The use of skill objects will allow the operator class functionality to be readily expanded. E.g. a Nagare skill object could be slotted in (hopefully) without modifying the operator code.

Class : tMultiMachineSkill
Multi-machine operator skill class. An object will use objects of this class to control a number of machines. It is intended that all those machines will be contained within a single work centre and that all are use for the same operation. This requirement is to critical however and therefore there is potential for using objects of this class in a more flexible way. Class is inherited from an abstract class and can therefore be used interchangeably with other sub classes of that abstract class.

Class : tOneMcSkill
Single machine operator skill class. An object will use objects of this class to control a single machine. Class is inherited from an abstract class and can therefore be used interchangeably with other sub classes of that abstract class.

Class : tStorage tStorageResults
Storage class for storing physical items such as batches and containers. Batches etc. can be added and removed. Limits can be set in terms of size and / or number. Batches can be queried according to a number of rules such as : FIFO / LIFO / earliest due date / etc.

Class : tAbsStore
Abstract class for static (i.e. do not move) classes which store physical objects. Holds a number of batches or number of containers (but not both). Batches and containers can be added / removed / manipulated. A batch or container quantity may not fall below 1 whilst in the buffer (?). The capacity of the buffer can be set in terms of number of containers or number of parts. The items are held in arrival order and can be selected based on a number of queuing rules (FIFO / LIFO / earliest due date / ...). Similar batches 'next to each other in the queue' can be merged or maintained separately.

Class : tStores
Stores class : objects of this class are used for storing large numbers of items : e.g. component stores / central stores / finished goods stores / etc This class differs from storage and batch storage class in that no record of the order of entry of batches / tools / etc. is available and items must be removed by name / reference / etc. This class can be used to store any physical item.

Unit : FactoryStringResource
String resource unit for factory classes. Enables strings to be dynamically loaded into the program. This approach is important since it avoids using string literals: string literals use the global data segment (64k) and therefore seriously reduces the number of classes that can be added.

Class : tTransport
This class serves as an abstract class for all types of transportation classes. The class provides some basic functionality as well as some (empty) virtual methods. It is possible to add or remove items to this class. The class is able to store these items and perform basic searches on them. This class does not require other classes for its existence and use. Operators etc. will use objects of descendent classes for transporting goods etc. This particular class is passive in that it must have things done to it rather than does things to other objects.

Class : tTransportSkill
This class enables operators etc. to use transportation devices. This class inherits and adds to the functionality of the abstract skill class. This class is a skill type class and therefore object of this class can be placed on an operator's skill list.

Class : tTray
Currently not in use - more work required. Tray class is used for storing small (in size) batches. Only one batch can be stored at any one time. The held quantity is dependent on the volume of the tray and the volume of a part.

Class : tAbsTruck tTruckResults
The abstract truck class has been created to be a descendent type for all truck classes. This class inherits the abstract transportation class and adds functionality specific to trucks namely speed and storage capacity.

Class : tHandTruck
This a specific type of transportation class which can be used by personnel to transport parts / tools / etc. This class inherits the properties of the abstract truck class

Class : tBatchInfo tWIPTracker
This class is able to track and record the status of work-in-progress. Batches will inform this class of changed status - next operation / qnty scrapped / etc. This class will record the details on individual batches and use this to create statistics on the batches created todate. The primary purpose of this class is to record information on batches that have been through the system but have since been destroyed. AFS is only able to display results of objects are still in existence. Since this class tracks the movement of batches in order to record historic information then the class is also able to provide current information on the status of batches.

Class : tWorkCentre
This is a detailed work centre class. The class is able to group machines together and provide collective input and output storage areas. The size and position of the work centre is adjusted according to the machines held.

Class : tWorkingPattern
This class can provide classes inherited from tPerson with detailed working patterns including (eventually!):

- absenteeism
- holidays
- breaks (e.g. tea breaks and lunch breaks

This class is very detailed in its modelling ability and therefore use of objects of this class may have a noticeable effect on the simulation time. This objects of this class should only be used when the above characteristics need to be modelled. If not the use of

objects of type tShiftList is advised. This class is inherited from an abstract working pattern type and therefore can be used by a object of type tPerson in place of tShiftList The change from simple shift pattern to use of this working pattern is seamless and the object of type tPerson can undergo this change at any time during run time.

Class : tWorksOrder
Works order class contains the part number / the order reference / launch date / due date and quantity required. An 'information' class.

Class : tWorkStation
Abstract class at which work is carried out on detailed level. All work station types have a work centre.

Graphics application classes and units

Class : tAbstractWindowDisplay
This abstract class must be inherited before 'using'. This abstract class has been created to avoid circular referencing involving the actual display drawing routines and the icons. This abstract class provides the actual screen drawing functionality. The class that inherits this provides the functionality to be used (indirectly) by the core (e.g. draw this icon at this location). The inheriting class will instruct the icon to draw itself. The icon classes use this class for drawing. This class buffers the inheriting class from the Windows specific drawing routines involving device contexts and magnification and the scrollers.

Class : tAfsGraphicsApplication
This is the AFS windows application class. For use with the graphics application. This should only be used by a simple program for use in Microsoft Windows environment.

Program : AfsGraph (Windows Graphics Application)
This is the AFS graphics program for the Microsoft Windows environment. This provides the graphical display and animation capability for the simulator application.

Class : tAFSGraphicsWindow
This class provides the DDE and Windows links for the core part of the graphics application. This class also provides the graphics application with an icon and initialises much of the graphics application data.

Class : tCoordPair tBlock tIcon
This unit contains a number of classes for the creation of icons. Only one of the classes should be used directly. The classes can be used to extract from a single DXF file of one or more blocks. Each blocks must be / is assumed to be (?) made up of a number of lines that form a closed polygon. The ordering of the blocks / the ordering of the lines within a block is unimportant.

Unit : IconDynamicLinkLibrary
This DLL supports the creation and manipulation of one or more icons. Icons can be created - stored on a list - recalled and destroyed. Icons can be loaded from the following file formats

- DXF

This DLL is shared between the simulator and graphics applications

Unit : IconDLLDeclarations
This unit eases the task of using the IconDLL dynamic link library. This unit allows the IconDLL to be used as if it is a unit. No declarations are necessary and the user can use strings rather than pchars.

Unit : WinDsplyMouseMethods
These methods create the appropriate response when the mouse is pressed / moved / etc on any of the graphical displays. These actions are interpreted locally. Local information is updated and the user-interface and the simulator are informed. I.e. if the user clicks on an icon the user-interface is informed. If an icon is dragged the simulator application is informed.

Class : tAbsObjIcon tPolygonIcon tRectIcon tFillRectIcon tBorderIcon
These classes are used to store icon data for drawing to a display. These classes store the data and are able to (un)draw themselves. The classes assume that the display has been prepared for drawing. I.e. a drawing context is available. Coordinates held are logical coordinates and no conversion is necessary.

Class : tPhyWindowMngr
This class allows the manages the physical windows. The class is able to direct display commands to the appropriate physical display. During development (and possibly still) this class was delivered interpreted Dynamic Data Exchange (DDE) commands from logical displays in the core.

Unit : TrigFunctions
Collection of trig functions for use by the graphics application for (among other things) rotating icons.

Class : tWindowDisplay
Actual Windows display class. Inherits the display ability from abstract class. This class is responsible for storing the data that will be displayed. This class instructs each object to draw / highlight / etc. itself and each object will in turn use the abstract display class to carry this out.

Factory class managers

Class : tAbsMatlControlMngr
This class manages the functions associated with all objects of the abstract material control class.

Class : tAssemblySkillMngr
This class manages the functions associated with all objects of the tAssemblySkill class.

Class : tManualAssyStatMngr
This class manages the functions associated with all objects of the tAssemblyManual class.

Class : tBenchMngr
This class manages the functions associated with all objects of the tBench class.

Class : tBatchMngr
This class manages the functions associated with all objects of the tBatch class.

Class : tChangeOverSkillMngr
This class manages the functions associated with all objects of the tChangeOverSkill class.

Class : tCellMngtMngr
This class manages the functions associated with all objects of the tCellManagementSkill class.

Class : tDepartmentMngr
This class manages the functions associated with all objects of the tDepartment class.

Class : tForemanMngr
This class manages the functions associated with all objects of the tForeman class.

Class : tGroupOfPartsMngr
This class manages the functions associated with all objects of the tGrpParts (group of parts) class.

Class : tInspectionSkillMngr
This class manages the functions associated with all objects of the tInspectionSkill class.

Class : tRawMatRequisitionMngr
This class manages the functions associated with all objects of the raw material requisition class.

Class : tAbsMachMngr
This class manages the functions associated with all objects of the abstract tAbsMachine class.

Class : tAutomaticMachMngr
This class manages the functions associated with all objects of the tAutomaticMachine class.

Class : tManualMachMngr
This class manages the functions associated with all objects of the tManualMachine class.

Unit : FactoryManagersRegistration
This is a key unit in the AFS expansion mechanism. This is the one and only place where existing code needs to be modified to insert a new class. Inserting a class manager here forces the compiler to compile the new class into the existing system. This is a developer specific unit (PDB). This unit has been created to simplify the merging of code from two or more developers working concurrently. This unit can only contain factory class managers.

Class : tNagareSkillMngr
This class manages the functions associated with all objects of the tNagareSkill class.

Class : tOperMngr
This class manages the functions associated with all objects of the tOperator class.

Class : tOrderMngr
This class manages the functions associated with all objects of the tOrder class.

Class : tOrderSysMngr
This class manages the functions associated with all objects of the tOrderSystem class

Class : tPartMngr
This class manages the functions associated with all objects of the tPartNumber class.

Class : tPersonMngr
This class manages the functions associated with all objects of the tPerson class

Class : tRepairSkillMngr
This class manages the functions associated with all objects of the tRepairSkill class.

Class : tRouteMngr
This class manages the functions associated with all objects of the tRouting class

Class : tRoutingBasicMngr
This class manages the functions associated with all objects of the tRoutingBasic class

Class : tScheduledMatlControlMngr
This class manages the functions associated with all objects of the scheduled material control class.

Class : tShiftMngr
This class manages the functions associated with all objects of the tShift & tShiftList classes.

Class : tAbsSkillMngr
This class manages the functions associated with all objects of the abstract tAbsSkill class.

Class : tMultiMachineSkillMngr
This class manages the functions associated with all objects of the tMultiMcSkill class.

Class : tSgMcSkillMngr
This class manages the functions associated with all objects of the tOneMcSkill class.

Class : tTransportSkillMngr
This class manages the functions associated with all objects of the tTransportSkill class.

Class : tStoresMngr
This class manages the functions associated with all objects of the tStoresClass class.

Class : tAbsStoreMngr
This class manages the functions associated with all objects of the abstract tAbsStore class.

Class : tBatchStoreMngr
This class manages the functions associated with all objects of the tBatchStore class

Class : tAbsTruckMngr
This class manages the functions associated with all objects of the abstract tAbsTruck class.

Class : tHandTruckMngr
This class manages the functions associated with all objects of the tHandTruck class.

Class : tTransportMngr
This class manages the functions associated with all objects of the abstract tTransport class.

Class : tWorkCMngr
This class manages the functions associated with all objects of the tWorkCentre class.

Class : tWIPTrackMngr
This class manages the functions associated with all objects of the tWIPTracker class.

Class : tWorkStationMngr
This class manages the functions associated with all objects of the abstract tWorkStation class.

Class : tWorksOrderMngr
This class manages the functions associated with all objects of the tWorksOrder class.

Random number generator classes

Class : tExponential
Random number generator : Exponential distribution.

Class : tFixed
Random number generator : Fixed distribution. This class has been created so that non-random and random number generation can be seen as the same. Hence fixed values can be used interchangeably without the need to handle exceptions.

Class : tNegExponential
Random number generator : Negative exponential distribution.

Class : tNormal
Random number generator : Normal distribution.

Class : tUniform
Random number generator : Uniform distribution.

Class : tFixedMngr
This class manages the functions associated with all objects of the tFixed class.

Class : tNegExpMngr
This class manages the functions associated with all objects of the tNegExponential class.

Class : tNormalMngr
This class manages the functions associated with all objects of the tNormal class.

Class : tPoissonMngr
This class manages the functions associated with all objects of the tPoisson class.

Class : tPoisson
Random number generator : Poisson distribution

Class : tRandomManager
Random number generator manager class This class is used when creating new random number objects. The class is a able to provide a new seed for each object.

Class : tRandomMngr
This class manages the functions associated with all objects of the random number generator class.

Class : tStdRndGen
Standard random number generator. Generates random numbers between 0 and 1.

Class : tUniformMngr
This class manages the functionality associated with the uniform random number generator class.

Appendix 4: Full class hierarchies

Overview

This appendix lists the class hierarchies for the Advanced Factory Simulator. Two class hierarchies exist: one for the simulator application and one for the graphics application. The user-interface development language was not object-oriented hence there is no hierarchy for that application.

The hierarchies shown were correct at the time this document was submitted. The nature of object-oriented software is such that with further development of AFS the documentation given below will quickly become out of date. Up to date hierarchies can be obtained by compiling the relevant source code in the Borland Pascal with Object v7 development environment and selecting the 'Search' and 'Objects' menu options. This will provide a graphical view of the application's class hierarchy.

Only the relevant parts of the commercial class libraries are shown below (one of the problems with commercial class libraries is that large parts can be compiled into an application even if they are not used). The Borland Pascal classes are prefixed with a upper case letter (i.e. 'T') whilst the AFS classes are prefixed with a lower case letter (i.e. 't'). Elements of the hierarchies not developed or co-developed by the author are clearly marked.

Graphics application hierarchy

```
tAFSGraphicsDDE
tClock *
tColourList
tDDEClient +
tDDEServer +
tLists
TObject
    TApplication
        tAfsGraphicsApplication
    tBlock
    tColourClass
    tCoordPair
    tDDE +
    tIcon
    tRootWBS +
        tRootAFS
            tAbsObjIcon
                tBorderIcon
                tPolygonIcon
                tRectIcon
                    tFillRectIcon
            tPhyWindowMngr
    TWindowsObject
        TWindow
            tAbstractWindowDisplay
                tWindowDisplay
            tAFSGraphicsWindow
```

+ : Developed with DML & NH. * : Developed with JAB

Simulator application hierarchy

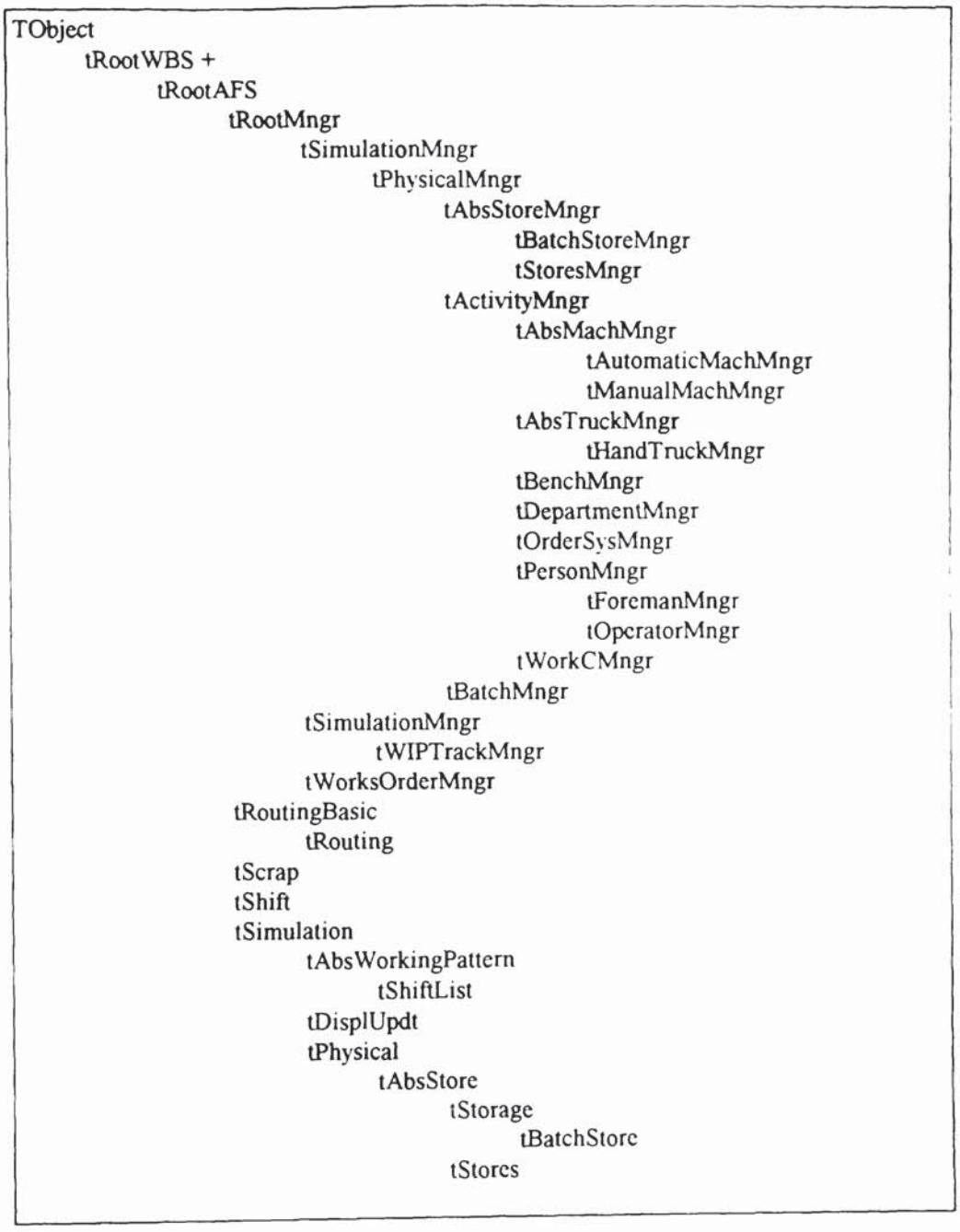
```
tAFSDDE
tClock *
tColourList
tDDEClient +
tDDEServer +
tLists
TObject
        TApplication
                tAfsApplication
        tBlock
        tClassListNode
        TCollection
                tClassList
                tConfiguredClassList
                tConfiguredObjectList
                TSortedCollection
                        tEventCollection
                        tOperationCollection
                        tOrderList
        tColourClass
        tConfiguredClassListNode
        tConfiguredObjectListNode
        tCoordPair
        tDDE +
        tEvent
        tIcon
        tInterface
        tRootWBS +
                tRootAFS
                        tAbsMaterialControl
                                tScheduledMatCrtl
                        tAbsPhyDisplayMngr
                                tPhyDisplayMngr
                        tAbsResults
                                tActivityResults
                                        tMachineResults
                                        tPersonResults
                                                tOperatorResults
                                tStorageResults
                                tTruckResults
                        tAbsSkill
                                tAbsMcSkill
                                        tMultiMcSkill
                                        tNagareSkill
                                        tOneMcSkill
                                tAssemblySkill
                                tCellManagementSkill
                                tChangeOverSkill
                                tInspectionSkill
                                tRepairSkill
                                tTransportSkill
```

continued...

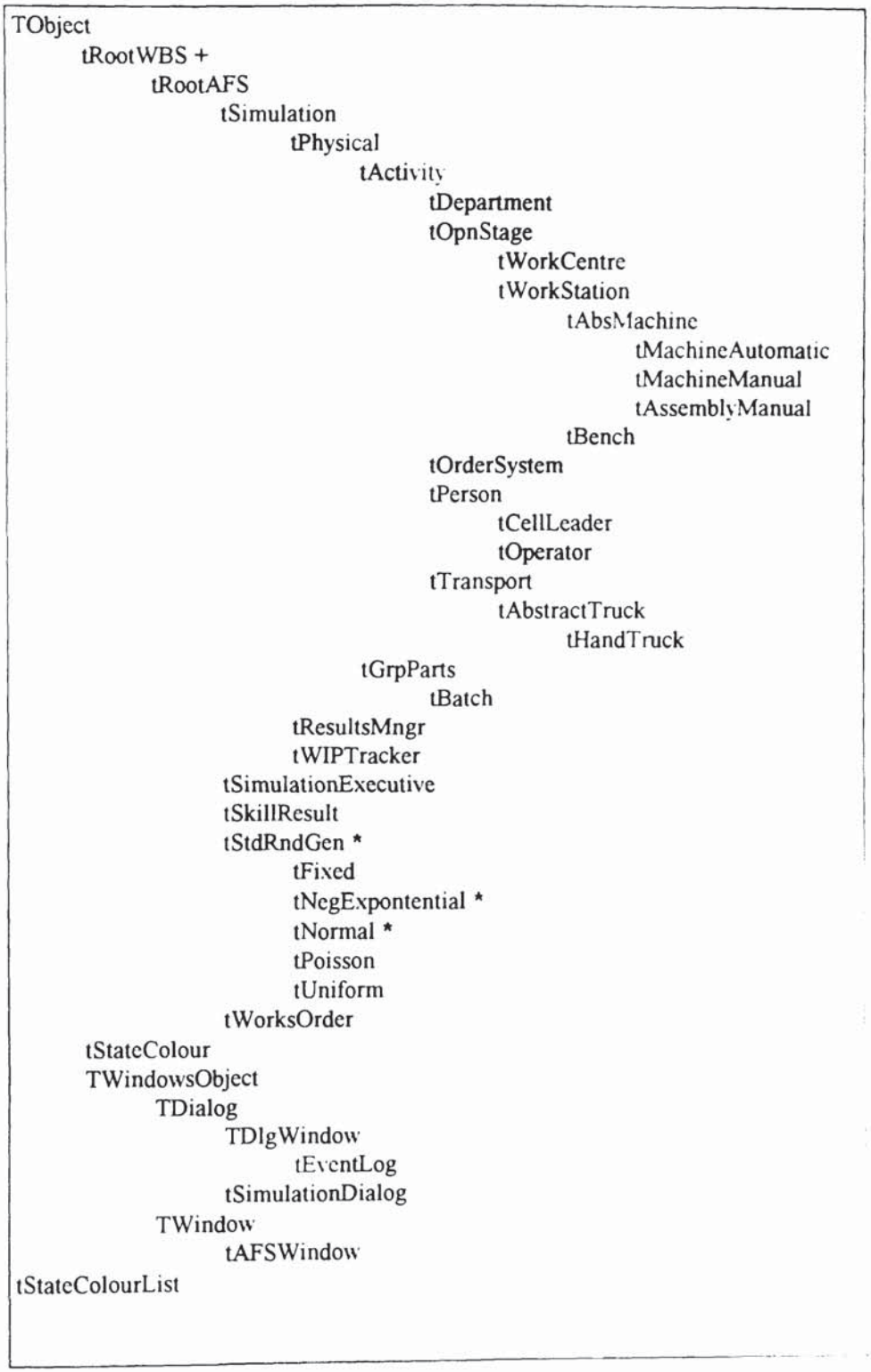
```
TObject
    tRootWBS +
        tRootAFS
            tAbstractLogicalDisplay
                tDisplayManager
                tLogicalDisplay
            tAbstractPhysicalDisplay
                tDDEPhysicalDisplay
            tBatchInfo

            tConfigtr
            tEnvironmentClass
            tEventList
            tInspectionResult
            tMessageClass
            tNagareSeq
            tOpnBasic
                tOpnNumber
            tOrder
            tPartNumber
            tRandomManager
            tRawMatlRequisition
            tRootMngr
                tAbsSkillMngr
                    tAssemblySkillMngr
                    tCellMngtMngr
                    tChangeOverSkillMngr
                    tInspectionSkillMngr
                    tNagareSkillMngr
                    tRepairSkillMngr
                    tSgMcSkillMngr
                    tTransportationSkillMngr
                tOrderMngr
                tPartMngr
                tRandomMngr
                    tFixedMngr
                    tNegExpMngr
                    tNormalMngr
                    tPoissonMngr
                    tUniformMngr
                tRouteMngr
                tShiftMngr
```

continued ...


TObject
tRootWBS +
tRootAFS
tRootMngr
tSimulationMngr
tPhysicalMngr
tAbsStoreMngr
tBatchStoreMngr
tStoresMngr
tActivityMngr
tAbsMachMngr
tAutomaticMachMngr
tManualMachMngr
tAbsTruckMngr
tHandTruckMngr
tBenchMngr
tDepartmentMngr
tOrderSysMngr
tPersonMngr
tForemanMngr
tOperatorMngr
tWorkCMngr
tBatchMngr
tSimulationMngr
tWIPTrackMngr
tWorksOrderMngr
tRoutingBasic
tRouting
tScrap
tShift
tSimulation
tAbsWorkingPattern
tShiftList
tDisplUpdt
tPhysical
tAbsStore
tStorage
tBatchStore
tStores

continued...



+ : Developed with DML & NH, * : Developed with JAB

Appendix 5: Registering a new class in the simulator

This appendix shows how a new class can be registered in the Advanced Factory Simulator.

Each new class must have a corresponding class manager. The class manager handles tasks that are common to all objects of the new class. For example, the class manager will enable data to be loaded from text file or interpret instructions from the user-interface for editing a particular object. The class manager hierarchy mirrors that of the class hierarchy hence much of the functionality of a new class manager can be inherited.

The class and class manager can be developed independently of all other classes. The process of registering is carried out in two stages:

- the class manager unit is entered into the 'USES' section: this enables the Borland Pascal compiler to find the source code;
- an object of the class manager is initialised and placed on the class manager list: placement of the list allows the user-interface to detect the presence of a new class and therefore allows the user-interface (without modification) to display the results of the new class, load up text files, etc.

There is one master class manager registration unit. Slave manager registration units can be included in the master. Each developer will have exclusive use of a slave manager registration unit. For example, the first developer may have exclusive use of the 'Dev1Mngr' unit. This 'Dev1Mngr' unit will be registered in the master class manager registration unit. This approach allows different developers to work on the different copies of the AFS system simultaneously and then merge the software with minimal effort. The format of the slave manager units is identical to the master.

The integration of the new class with the existing classes is shown below.

```
UNIT Managers;
{-------------------------------------------------------------------------
TYPE:
  LIBRARY   MasterClassManagerDeclaration

DATES:
  CREATED   PDB    4th April 1993
  UPDATED   PDB   28th February 1994

SUMMARY:
   This is a key unit in the AFS expansion mechanism.  This is the one
   and only place where existing code needs to be modified to insert a
   new class.  Inserting a class manager here forces the compiler to
   compile the new class into the existing system.
   This unit ties together different developers' class manager
   registration units.  New developers must create their own
   unit and register it here.  This unit has been created to simplify
   the merging of code from two or more developers working concurrently.
   This unit can contain general class managers that are not specific to
   an application.  For example the random class class managers should
   be registered here.

DETAIL:
  The registration process is very easy.  Simply follow the number
  sequence and format below.

------------------------------------------------------------------------- }

INTERFACE

USES
  { Borland Pascal Units }
    Objects;        { Turbo Vision & Object Windows general object unit }

{ procedures }
  PROCEDURE LoadClassManagers(pClassManagerList : Objects.PCollection);

IMPLEMENTATION

USES
  { LOCATION 1 : Enter the new class manager unit into the uses section }
  { Developers class managers : register general classes here }
    Fixd_mgr,       { fixed 'random number generator' class manager class }
    Norm_mgr,       { normal random number generator class manager class }
    Unif_mgr,       { uniform random number generator class manager class }
    NExp_mgr,       { negative exponential RNG class manager class }
    Pois_mgr,       { Poisson RNG class manager class }
  { register new class manager units here }
    MngrFact,       { PDB : factory class manager registration unit }
    MngrCtrl;       { NJB : control class manager registration unit }

  { --------------------  LoadClassManagers  ------------------------ }
  PROCEDURE LoadClassManagers(pClassManagerList : Objects.PCollection);
    BEGIN
      { LOCATION 2 : }
      { EITHER : create a developer specific manager registration unit here : }
      { register new class manager loading routines here! }
        MngrFact.LoadFactoryClassManagers(pClassManagerList);  { PDB }
        MngrCtrl.LoadControlClassManagers(pClassManagerList);  { NJB }
      { OR : enter the class manager here }
      { To register a class manager here, simultaneously initialise an object
        of the new class manager and place it on the class manager list. }
      { NB1: The order here is replicated in the user-interface }
      { NB2: Only place general classes here.  Application specific classes
             should be placed in an application specific managers unit.  A
             separate unit is not strictly necessary, however, it eases the
             task of merging code developed by different developers. }
      { general class managers }
        { Fixed random number generator }
          pClassManagerList^.Insert(NEW(Fixd_mgr.tFixedMngrPtr, Init));
        { Normal random number generator }
          pClassManagerList^.Insert(NEW(Norm_mgr.tNormalMngrPtr, Init));
        { Uniform random number generator }
          pClassManagerList^.Insert(NEW(Unif_mgr.tUniformMngrPtr, Init));
        { Negative exponential random number generator }
          pClassManagerList^.Insert(NEW(NExp_mgr.tNegExpMngrPtr, Init));
        { Poisson random number generator }
          pClassManagerList^.Insert(NEW(Pois_mgr.tPoissonMngrPtr, Init));
    END; { LoadClassManagers }
END. { Managers }
```

Appendix 6: Development limitations

There are some restrictions that have resulted from the choice of languages and commercial libraries. The main restrictions are detailed below.

User-interface

The user-interface has been implemented using Microsoft's Visual Basic version 3. The user-interface consists of a number of dialogs, for example machine, operator and truck dialogs. For the particular version of Visual Basic the number of dialogs is limited to 230. It is not thought that this limit will impose restrictions on development. Other restrictions are detailed in the user manuals. It is not thought that any of the other restrictions will affect the user-interface development.

Simulator

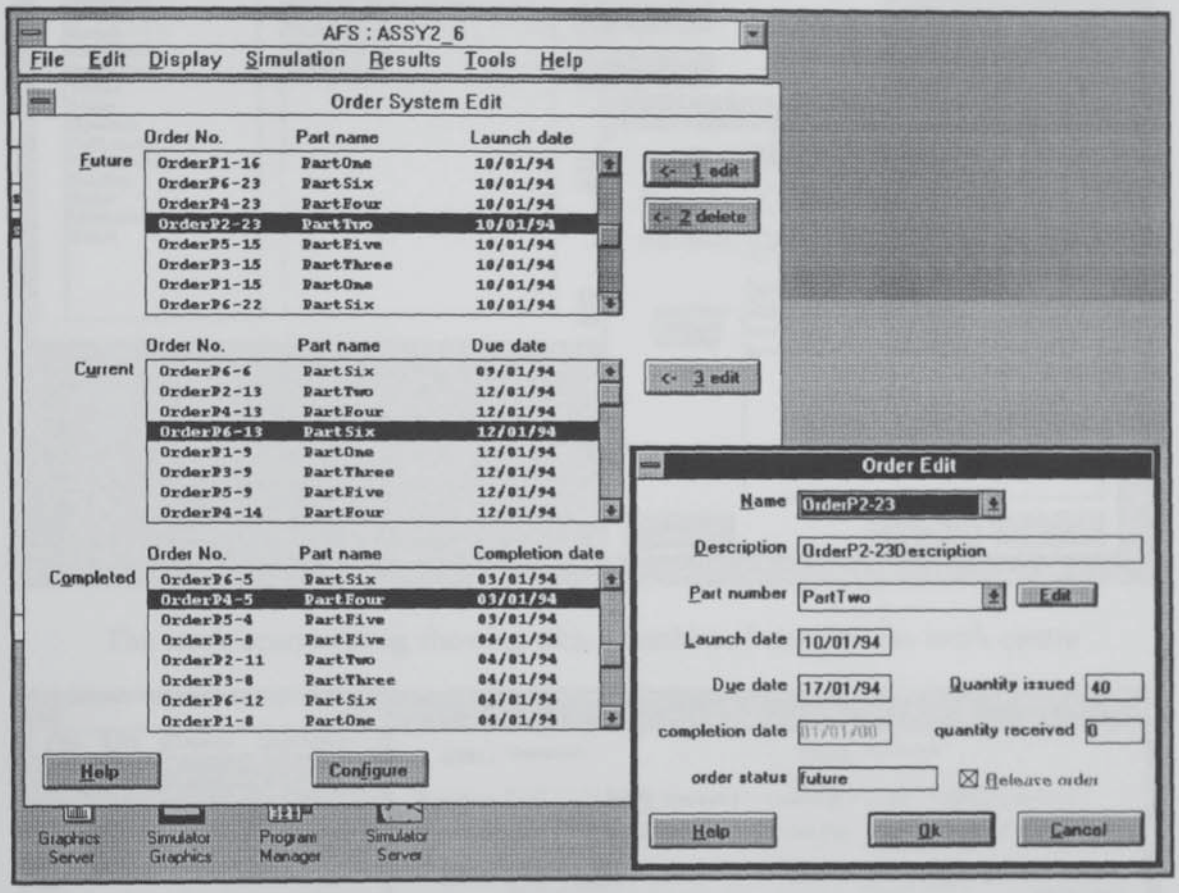
The simulator has been implemented using Borland's Pascal version 7. There are two key restrictions that have resulted. The first is the number of objects that can be created within a single model and the second is the number of classes a developer can add.

The number of objects is limited by the Borland Pascal library used for storing objects. The limit is approximately 10,000 objects for a single list. The main list for storing the objects has been encapsulated inside an AFS master list class. The encapsulation is such that the Borland Pascal list could be replaced at a later date without affecting the other software. It is envisaged that a list of lists will be used to overcome this problem. Hence in the short term the maximum number of objects within a model could be increased to 10,000 x 10,000 or 10^8 objects.

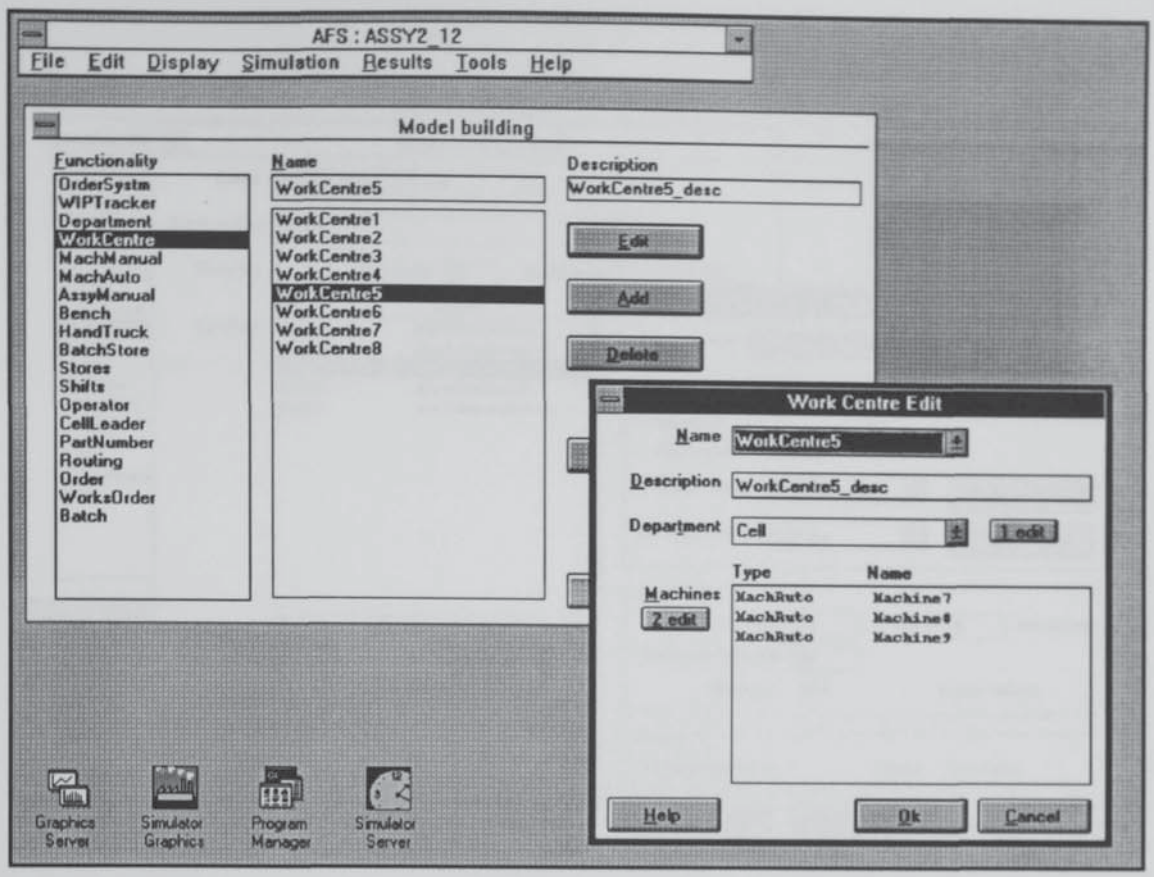
The other main limitation has been imposed by the Borland Pascal compiler. The compiler only allows a 64k global data segment for Windows applications. All global variables, the stack and the virtual methods table have to be stored within this 64k. The use of global variables within AFS is insignificant. The stack size has to be set at about 30k for successful operation of the software. This leaves just over 30k for the virtual method table. The virtual method table increases in size every time a new class is declared with virtual (dynamic) methods declared. The actual increase varies from class to class but could average 200 bytes. This therefore means there is a restriction on the number of classes that can be added. It is estimated therefore that AFS can only hold about 150 different application classes. This is considered to be the most serious limitation.

Appendix 7: Illustration of key user-interface dialogs

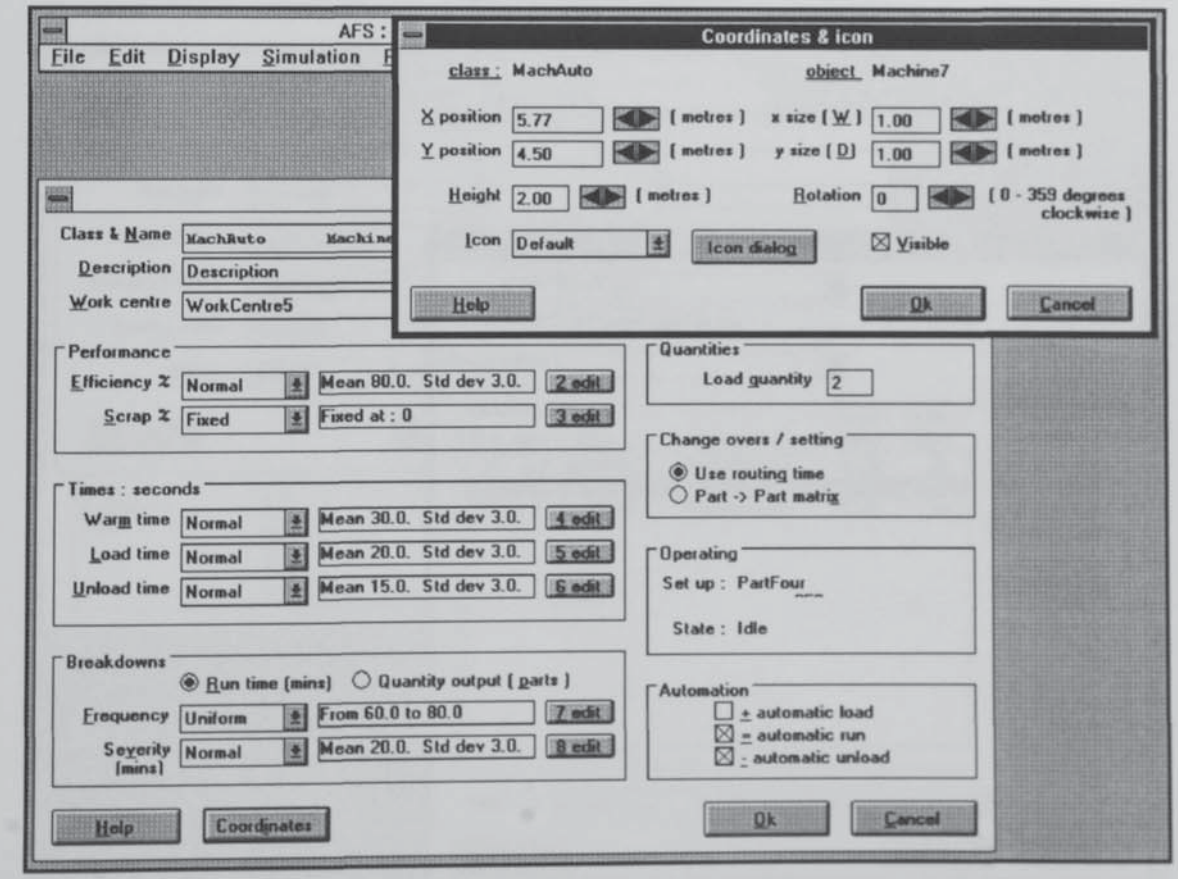
This appendix shows a number of the dialogs found in the user-interface of the Advanced Factory Simulator. The dialogs shown are termed application specific dialogs; the dialogs display information about modelling functionality that is specific to manufacturing system modelling.



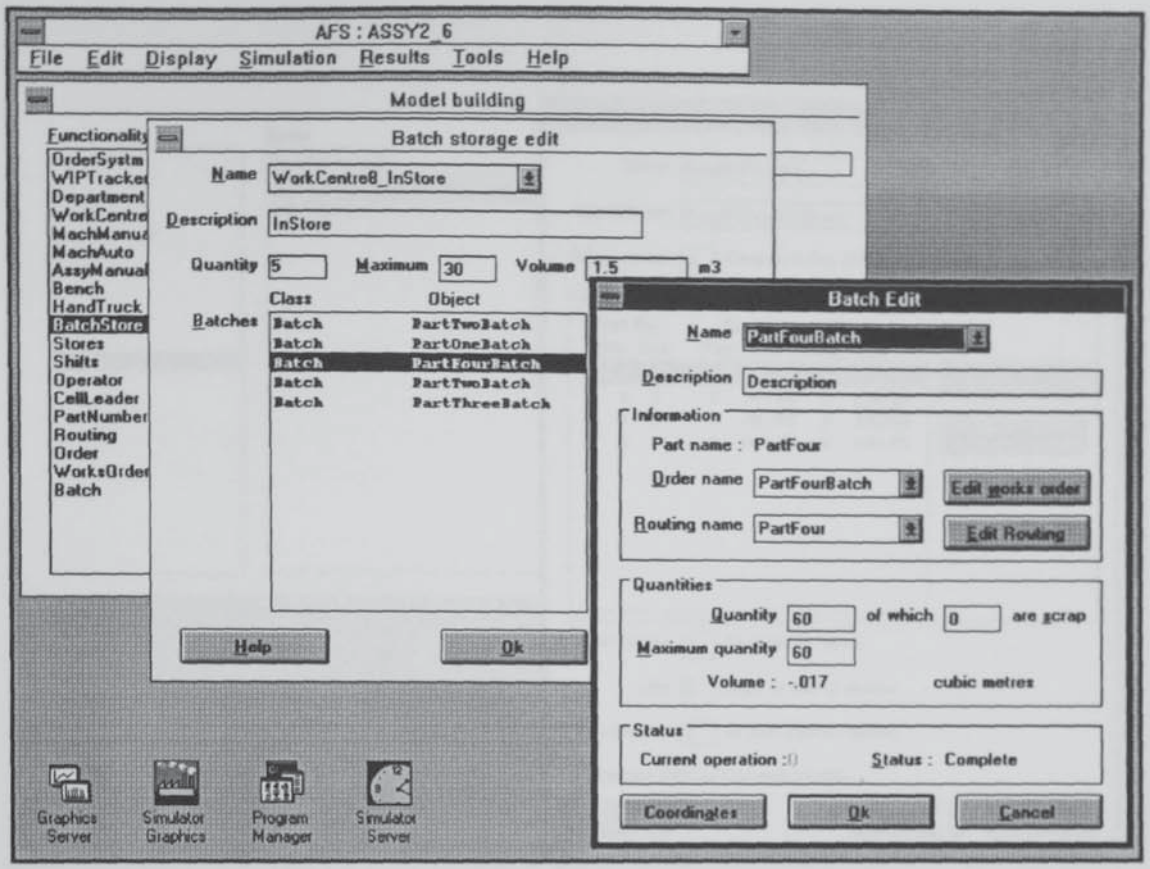
The order system and order dialog showing information about the orders



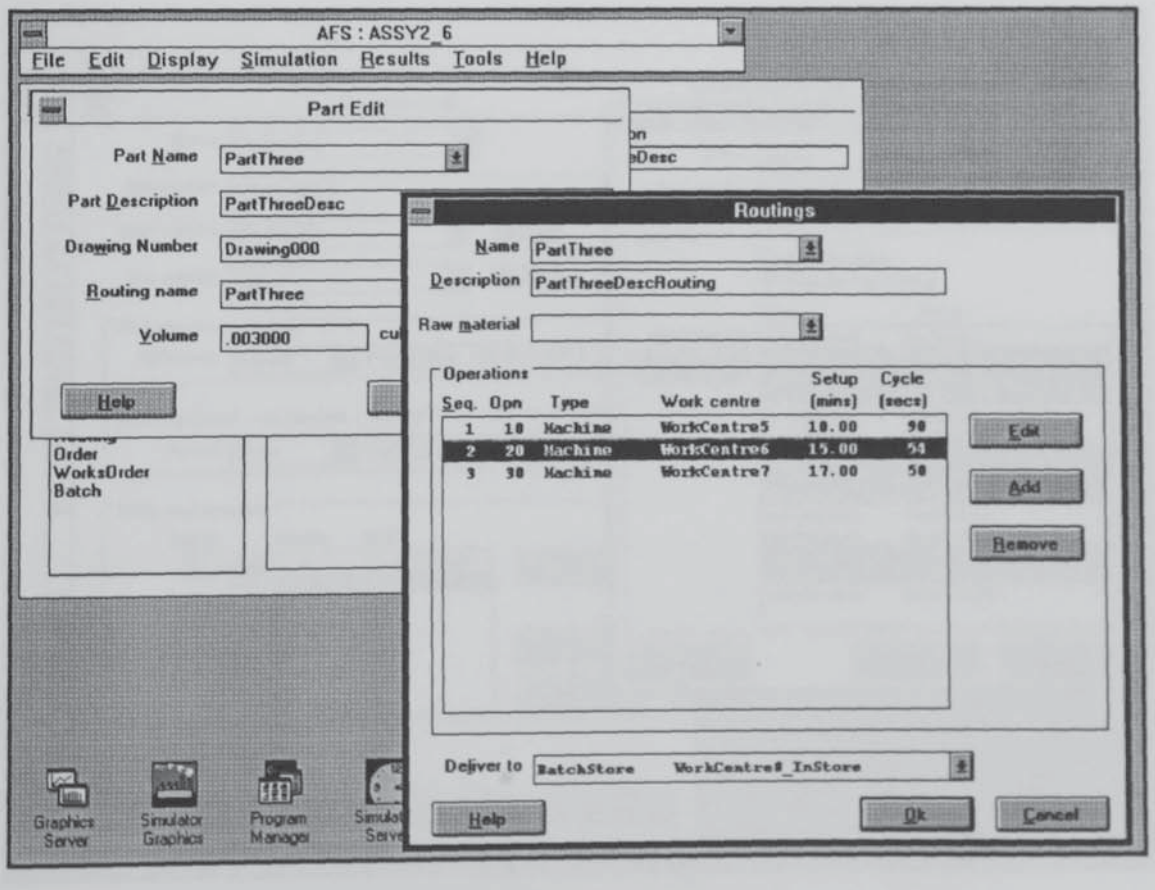
The work centre dialog showing which machines belong to the work centre



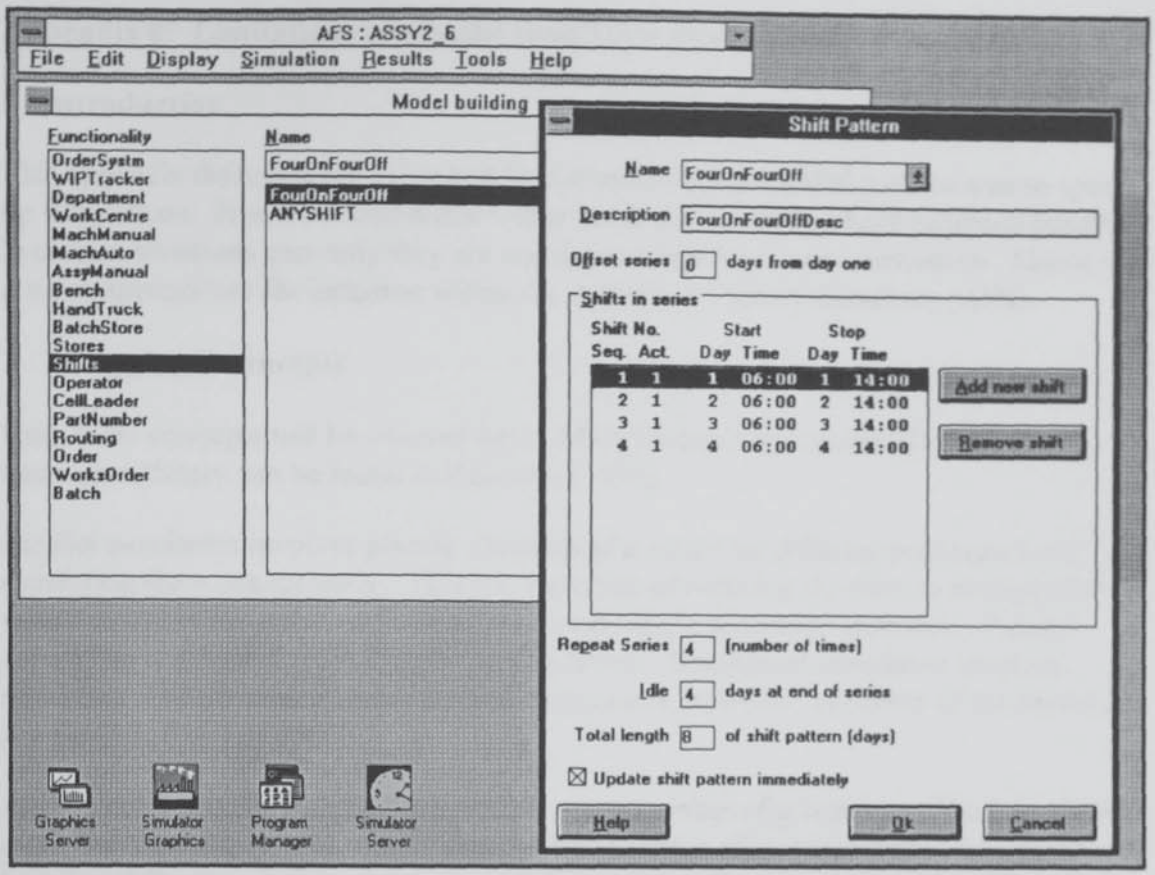
The machine and coordinates dialogs. They show detailed operational and physical information on each machine.



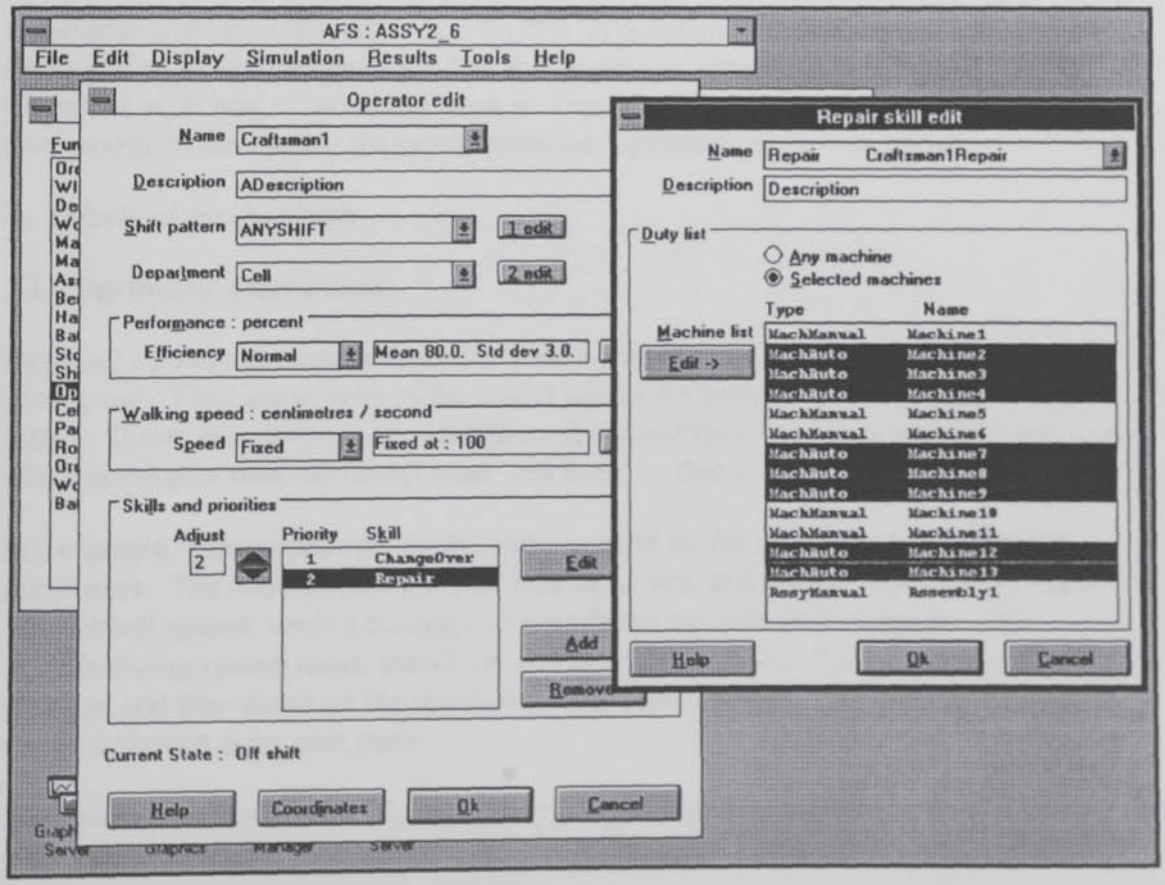
The batch storage area and the batch dialog. They show information about the location and status of batches.



The part number and routing dialogs.



The shift pattern dialog. The combination of offset from day one, repeat and idle days enables complex shift patterns to be set up.



The operator and repair skill dialog. An operator can have a number of prioritised skills.

Appendix 8: Limitations of parallel simulation mechanisms

1. Introduction

This appendix discusses the value and limitations of using parallel mechanisms to speed up simulations. It will be shown that whilst parallel mechanisms have potential benefits in certain situations currently they are not appropriate for factory simulation. Hence they are not appropriate for inclusion within the Advanced Factory Simulator (AFS).

2. Background concepts

Only basic concepts will be covered here. More in-depth discussion of parallel simulation theory can be found in Fujimoto (1990).

Parallel simulation involves placing elements of a model on different processors and simulating them concurrently. This has the effect of reducing the time to evaluate the whole model when compared to the traditional, single processor approach. Parallel simulation is different from distributed simulation. Distributed simulation involves spreading a single model across multiple processors, however, elements of the model are not simulated concurrently.

With parallel simulation modelling occurs over a number of processors. Each processor models a definable portion of the system. Mechanisms allow messages to be passed between different processors (or parts of the model). The messages are used for two reasons: firstly as a means of transferring data (such as a works order) and secondly as a means of synchronisation.

If a model is to be simulated using a number of processors then a mechanism must be employed to ensure that events within the model as a whole are in the correct order. Therefore each part of the model must be kept in step with the other parts. There are two classes of mechanism for synchronisation: optimistic and conservative.

3. Details of mechanisms

3.1. Optimistic mechanisms

Optimistic simulation mechanisms allow each part of the model to be processed at its own speed. Thus some parts of the model will be (in simulation time) ahead of the others. If one part of the model simulates ahead and then receives a message that is in effect in the past then the model must 'roll back' to that time and start again.

For example, a manufacturing model and a control model could reside on different processors. The manufacturing model may be at 3pm and the control system at 2pm. The control system sends a message to manufacturing with an instruction. The manufacturing system must, somehow, roll back to its state at 2pm, respond to the message and then continue the simulation from 2pm onwards. Meanwhile the control model advances at its own pace.

A common mechanisms for allowing the model to revert to a previous state is to save the model state at regular intervals. Hence when the model needs to revert to a previous state a previously saved model is used.

3.2. Conservative mechanisms

Conservative simulation mechanisms prevent the need to roll back by observing what is termed 'causality'. The causality constraint prevents events being processed out of order. The term asynchronous is used to describe conservative models that keep roughly in step, synchronous is used to describe models that are exactly in step. Synchronous mechanisms tend to employ a global broadcast mechanism, for example a central control mechanism tells all parts of a model to advance to 2pm, 2.05pm, etc...

Asynchronous mechanisms do not use a central clock but advance the time by agreement with one another. For example, if the manufacturing model tells the control it will not send a message before 2pm and the control tells manufacturing it will not send a message before 3pm then manufacturing can advance to 3pm whilst the control advances to 2pm. As the models reach their respective times they will generate more messages and hence new guarantees. If a part of the model has not received guarantees from all its connections then it will wait (or 'block' / 'deadlock') until such time that it does. It is possible for the entire model to block, each part waiting on the other parts. Hence mechanisms have been developed to detect such blockages and recover from them.

The main mechanism for preventing deadlock is the posting of 'null' messages. Null messages do not form part of the physical system but aid the simulation advance by providing guarantees between linked processes. Null messages can be sent either automatically by working processes or on request from blocked processes.

'Lookahead' is an important concept in conservative systems. Lookahead refers to a process's ability to predict its next event (or more importantly predict the time up to which it is impossible for events to occur). With the knowledge of the next event using lookahead a process can guarantee another process that it will not send a message before that time. This ability is vital in providing speed up. Systems offering poor or no lookahead will offer little or no speed up and hence fail to take advantage of any inherent parallelism.

The message population is an important issue also. Low message populations in conservative systems can lead to regular deadlock and hence the overhead of executing the simulation will rise accordingly.

4. Observations

The following observations can be made on parallel mechanisms from reading current literature:

- parallel mechanisms are in their infancy;
- parallel mechanisms are generally faster than conventional ones (i.e. ATOMS, ProModel, Witness);
- optimistic mechanisms are generally faster than conservative;
- optimistic mechanisms require more memory for storing roll back information;
- conservative mechanisms require some means of deadlock detection and recovery;

- conservative mechanisms require lookahead;
- feedback (interaction) between parts of a model seriously affects speedup;
- all illustrate mechanisms using simple examples;
- most assume single 'object' (logical process) per processor (process typically occupies only couple of hundred bytes);
- few if any examples of parallel manufacturing models (this is in contrast to the extensive use of traditional simulation mechanisms in manufacturing);
- most if not all use single hardware device with multiple processors;
- some multi-processor machines used offer shared memory.

5. Deductions

The following deductions are made in the context of factory simulation.

Optimistic mechanisms are inappropriate:

- complex to implement;
- state saving of elements of model is time consuming: an AFS model is potentially very large and therefore regular saving would be an extremely high overhead;
- use of anti-messages is inappropriate: it could be extremely difficult to undo what has been done in a system that involves complex interaction;
- memory requirements for rollback will be very large;
- memory requirements are many times that for conservative simulation;
- feedback / interaction reduces algorithm efficiency: again factory interactions could be significant.

Conservative mechanisms offer most appropriate approach:

- no need to store roll back information;
- approach will be faster than sequential approach.

Conservative mechanisms are not immediately applicable (if at all):

- application specific knowledge is required in mechanism: this will possibly lead to a mixing of the application specific and application independent complexity,
- lookahead is essential for speed up: this is difficult to achieve with complex models but decomposing complex models into simple elements requires high number of processors.

- there are problems with synchronisation: few examples in manufacturing / complex model elements;

- it is unclear as to whether each model would have to be built up with knowledge and consideration of the underlying parallel mechanism.

- there would be problems of ensuring the message population was high enough to prevent deadlock occurring;

- conservative speed up is typically a factor of two.

6. Concluding remarks

> *"If the application has both poor lookahead, and large state-saving overheads, all existing [Parallel Discrete Event Simulation] approaches will have trouble obtaining good performance, even if the application contains copious amounts of parallelism."*
>
> Fujimoto (1990, p. 50)

AFS has potentially poor lookahead due to the complexity of the models that will be produced. Obtaining simple sub-models by decomposing the overall model further is inappropriate since this will mean the model structure will dictate (possibly unobtainable) hardware requirements. Also due to the size of models that will be created using AFS the time required to save the state regularly will counter the benefits of optimistic parallel operation. To further this, the pace of developments in computer hardware weaken the argument for improved algorithms.

References

Fujimoto, R.M. 1990. "Parallel Discrete Event Simulation." *Communications of the ACM*, 33 (10): 30-53.

Appendix 9: Simulation speed

This section details the calculation of the simulation speed of AFS. The speed is compared to a commercially available, non-object-oriented simulator, ATOMS. The test was made by PDB on 11/01/94.

Overview

Comparison of simulators with differing functionality is difficult: cycle times, number of 'objects', operator duties, etc. will result in differences. Comparison is therefore based on number of 'events'.

The test consisted of 2 runs. The first run was used to ascertain the simulation speed, the second run to ascertain the number of state changes that occurred. Two runs were necessary since the use of event logs in both AFS and ATOMS seriously affects the simulation speed.

Test details

Version	**AFS v0.8**	**ATOMS v5 (Protected Mode)**
Environment	Windows v3.1 without graphics	DOS without graphics
Hardware	IBM PC 486, 33MHz	IBM PC 486, 33MHz
Model	4 line Nagare model. Model includes 24 machines, 4 operators & average of 4 operations per part.	Yoke flange model. 20+ machines, 15+ operators, 1 part, 7 operations
Run 1 (log off)	1 simulated day 2 minutes run time	1 simulated week 1 minute run time
Run 2 (log on)	2 simulated hours 6800 state changes recorded in log	1 simulated week 5000 events recorded in log

Calculation

AFS

Number of state changes in run (8 hour day) = 6800 x 4

State changes / second = (total state changes / actual time)

= (6800 x 4) / (2 x 60) = 230

Therefore **AFS** able to complete approx. **200 state changes / second**

ATOMS

Events / second = (total number events / actual time)

= (5000) / (1 x 60) = 100

Therefore **ATOMS** able to complete approx. **100 events / second**

Conclusion & Discussion

Orders of magnitude show that the speed of AFS is comparable with ATOMS. AFS speed is not a major issue.

The comparison made here is the worst case for AFS: AFS was operating Nagare whilst ATOMS was operating simple batch manufacture.

AFS state changes may be : operator to load to idle followed by idle to run therefore twice as many AFS changes may occur than events in ATOMS for equivalent result.

Architecture of AFS is such that expansion of functionality will have minimal impact on speed, i.e. event routines are unaffected.

The test of the simulation speed of AFS was conducted without the use of the graphical displays. This action was legitimate since the issue of simulation speed generally only arises during the process of carrying out simulation experiments. It is known that the speed of the graphics is 'slow'. There are a number of reasons for this: the DDE mechanism used to control the graphics application is inherently slow and Microsoft Windows graphical routines are 'slow'. The object-oriented nature of the display architecture could be 'slow', however, the inefficiencies commonly associated with object-oriented software do not appear to seriously affect the simulation speed. It is also worth noting that graphical displays are typically used to set up models and provide a detailed view of the progress of the simulation and therefore speed is not a critical issue when graphics are in use.

Appendix 10: Simple AFS Model

Overview

This basic model was built by the developer to demonstrate the speed and ease of model building. The model also demonstrates the way in which AFS models the production and detection of scrap.

This model uses a variety of functionality including: machines, trucks, people and people skills. The model was built using the dialogs and the progress of the simulation observed using the graphical displays.

Details

Builder	Experience	Location	Source of data
PDB	Developer	Aston	Synthetic

Model

Model	Approx. entities	Model purpose
Simple	50	Simple model to demonstrate speed and ease of use as well as some of the functionality

Times

Software training	Model build	Simulation
Not applicable	10 minutes	several minutes with animation

Model description

The model involved three work centres, each containing a single work station. Each work station was manned by a single operator. Orders for one type of part were released on to the shop floor. The batches were moved from work centre to work centre. The first two work centres carried out machining operations whilst the last work centre was used for inspection. The machine processed parts with a scrap rate of about 20%. The operator with inspection skills inspected the batches that arrived and attempted to detect the scrap. Batches were then delivered to stores.

Model results

A single simulation run demonstrated that inspection could be included in a model to detect scrap at given rates of success.

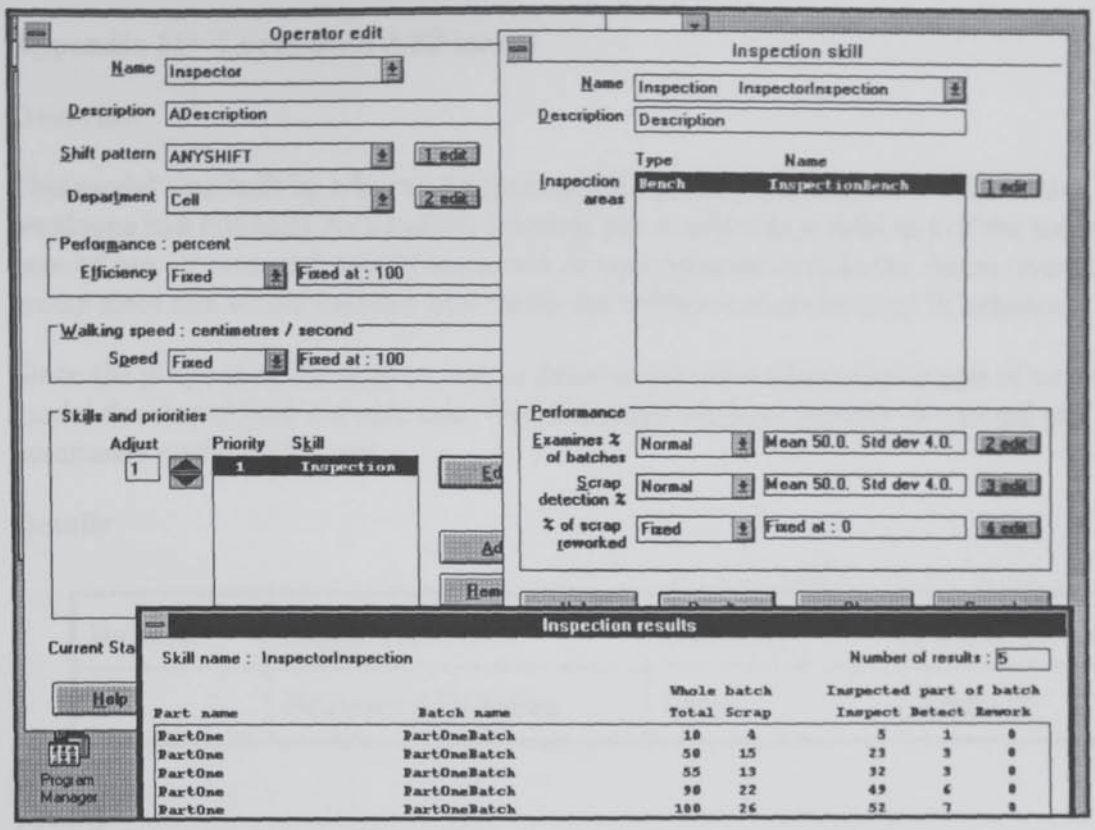
The table below shows a summary of the results. The first two columns show the size of the orders released and compound scrap after the first two operations. The middle two columns show the results of the inspection process (the inspection and detection rates were approximately 50%). The last two columns show the outcome of the process. Note that batches were delivered to stores with some scrap parts remaining.

Starting		Inspection		Remaining	
Total	**Scrap**	**Inspected**	**Detected**	**Total**	**Scrap**
10	4	5	1	9	3
50	15	23	3	47	12
55	13	32	3	52	10
90	22	49	6	84	16
100	26	52	7	93	19

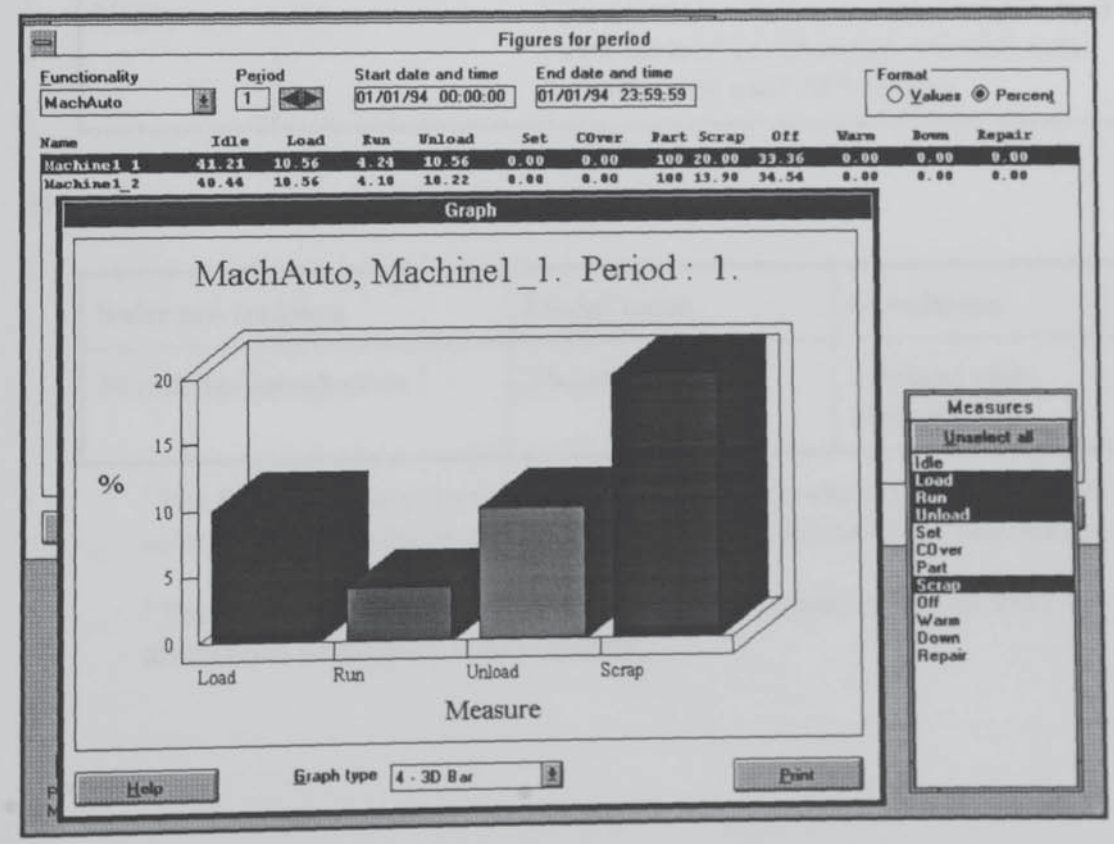
Software : Set up and results

The following screen dumps shows the model in the AFS software. The first picture shows the inspection process set up and results. The second shows the results for the machines.

Note: The scrap figures for the machines do not agree with the above figures. The mismatch is due to scrap parts being machined and being marked as scrap for the second time.



Screen dump of the inspection dialog set up and results



Screen dump of the machine results

Appendix 11: Lucas built AFS model

Overview

This model was built by a Lucas Engineering & Systems Ltd. employee. Since the employee had not used AFS before, building this model was a valid test of the software's ease of use. It was particularly important to use someone outside the Aston research group since this would indicate how easily the software could be used in industry.

Since the purpose of the session was to demonstrate ease of use (not scope of use) the model developed was a simple one. The following sections describe the model and comments made by the user.

Details

Builder	Experience	Location	Source of data
RJ	Not used AFS before	Lucas	Synthetic

Model

Model	Approx. entities	Model purpose
Nagare	50	To ascertain whether models could be built by a 'novice' (a simulation software user who had not used AFS before)

Times

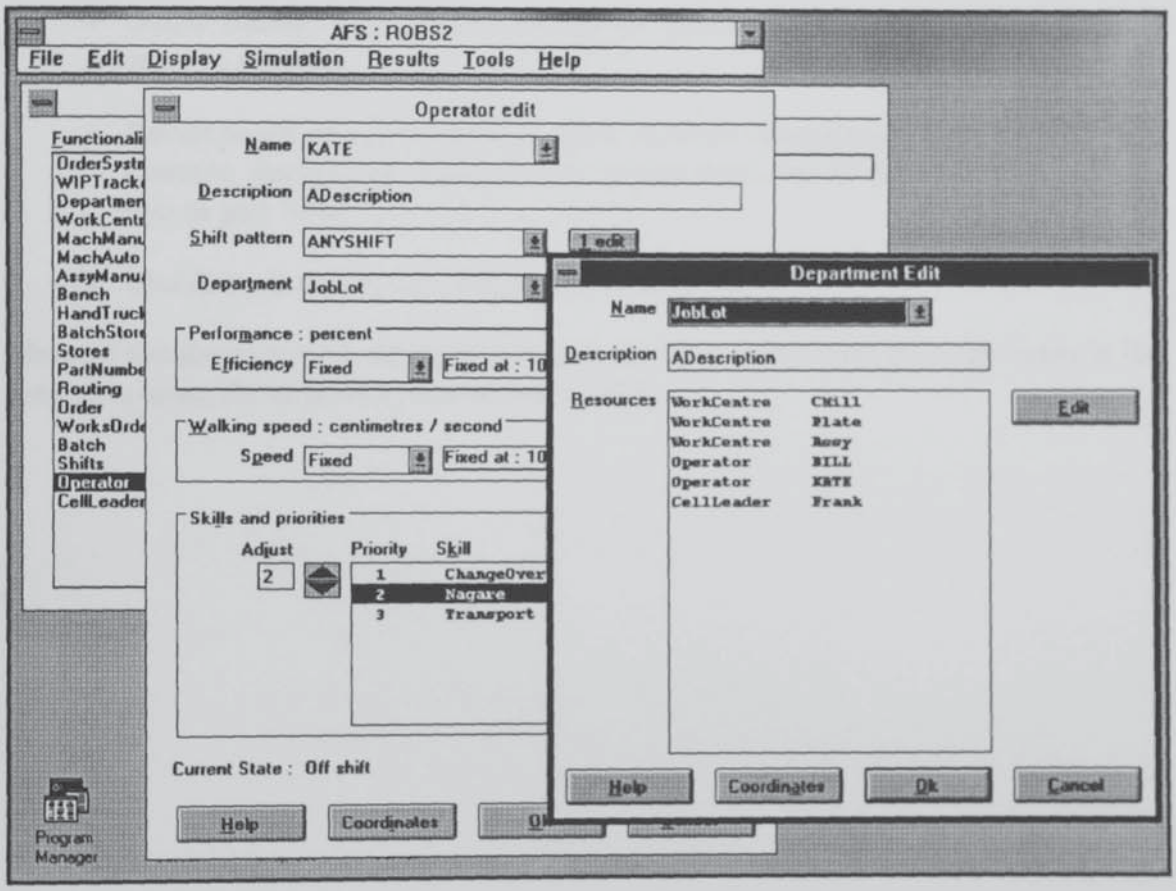
Software training	Model build	Simulation
30 minutes introduction [1]	2 hours [2]	1 minute with animation

[1] The figure indicates the time the author spent introducing the user to the software. During the model building the user made use of the help system.

[2] The model build time included help on specific issues as well as short discussions on possible improvements.

Model description

The model consisted of three work centres. Each work centre contained a single work station. Two operators were used. The work station in the first work centre was run by one operator whilst other two work stations were run by the other operator in a cyclic pattern. The two operators also carried out transportation and set up duties. Batches were moved from work centre to work centre by the appropriate operator. The duties of each operator were prioritised in the order from high to low: change overs, machining and transportation. Work was scheduled at the first work centre then 'pushed' to the subsequent work centres before being delivered to stores. The following screen dump illustrates the set up of part of the model.



Set up of one of the operators and the viewing of the contents of the cell.

Comments made / issues arising during the model building process

The following details comments made during the model building process. Some of the comments were due to inconsistancies in the software whilst others were related to the long term potential of the software. Whilst these issues were raised they did not prevent the completion and simulation of a model. The inconsistancies have since been retified. Long term development requirements noted:

- Development of forced input sequence to build model. The user would be automatically lead through a sequence of menus and dialogs to build a "first pass" model.

- Development of a fully automatic / abstract machine, i.e. one that will process work without the intervention of an operator.

- Development of a process machine.

- Development of abstract functionality to allow batches to be transferred between work centres automatically without the need for trucks or conveyors.

- Need to allow machines to be present in more than one work centre.

- Currently machines can only have one breakdown pattern. Request to modify to allow a machine to have multiple random number generators to allow more complex breakdown patterns.

- Global editing. For example, change the efficiencies of all machines from one value to another.

- Development of a part matrix to allow machine change overs to be varied. For example, change over from part one to part three may be different from part two to part three.

- Unable to display results during a simulation run.

Many of the above development requirements can be accomplished relatively easily in the long term using the expansion mechanism in AFS.

Appendix 12: AFS Nagare Model

Overview

This model was built by the developer to test the software and to demonstrate a particular aspect of the modelling that is possible.

This model uses the Nagare system dialog and the tailored graphical displays for building and demonstrating the model.

The Nagare functionality and the subsequent model were developed in response to requirements from Lucas Engineering & Systems Ltd (LE&S).

Details

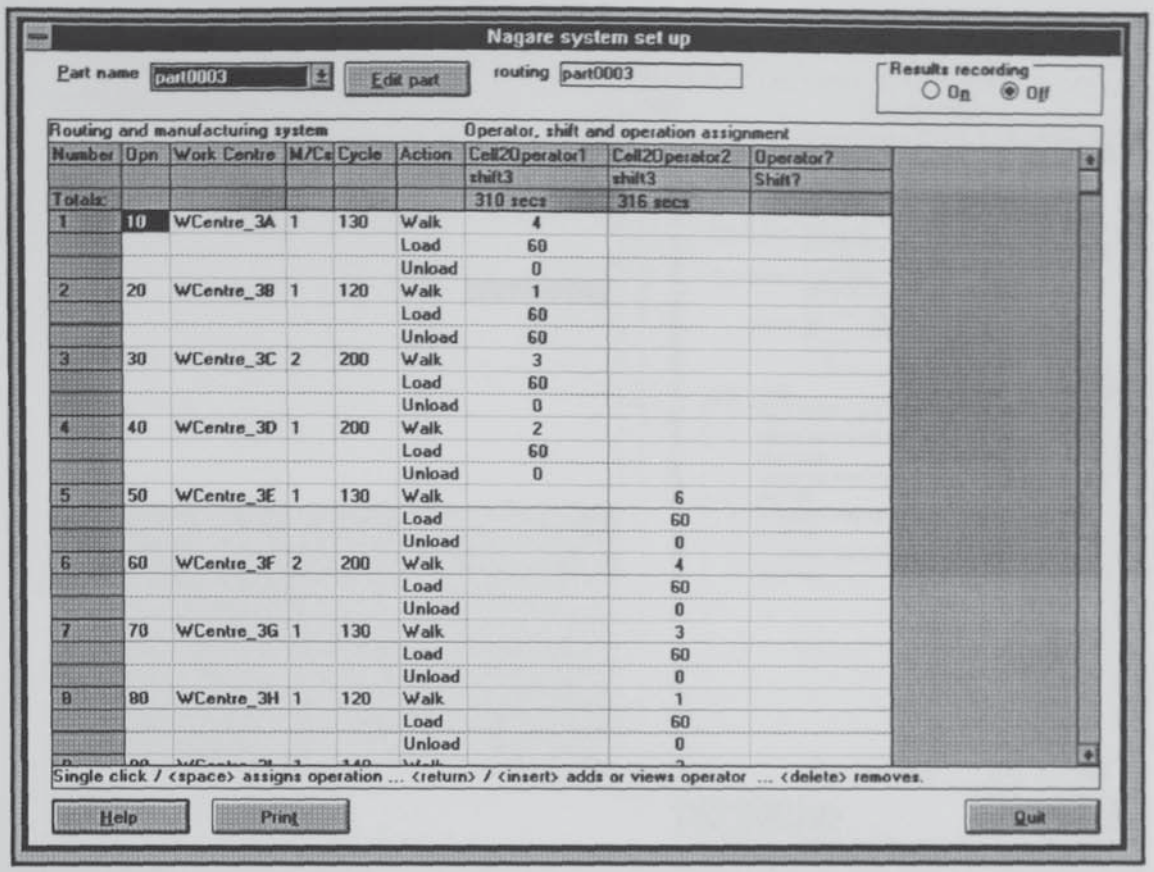
Builder	Experience	Location	Source of data
PDB	Developer	Aston	Synthetic

Model

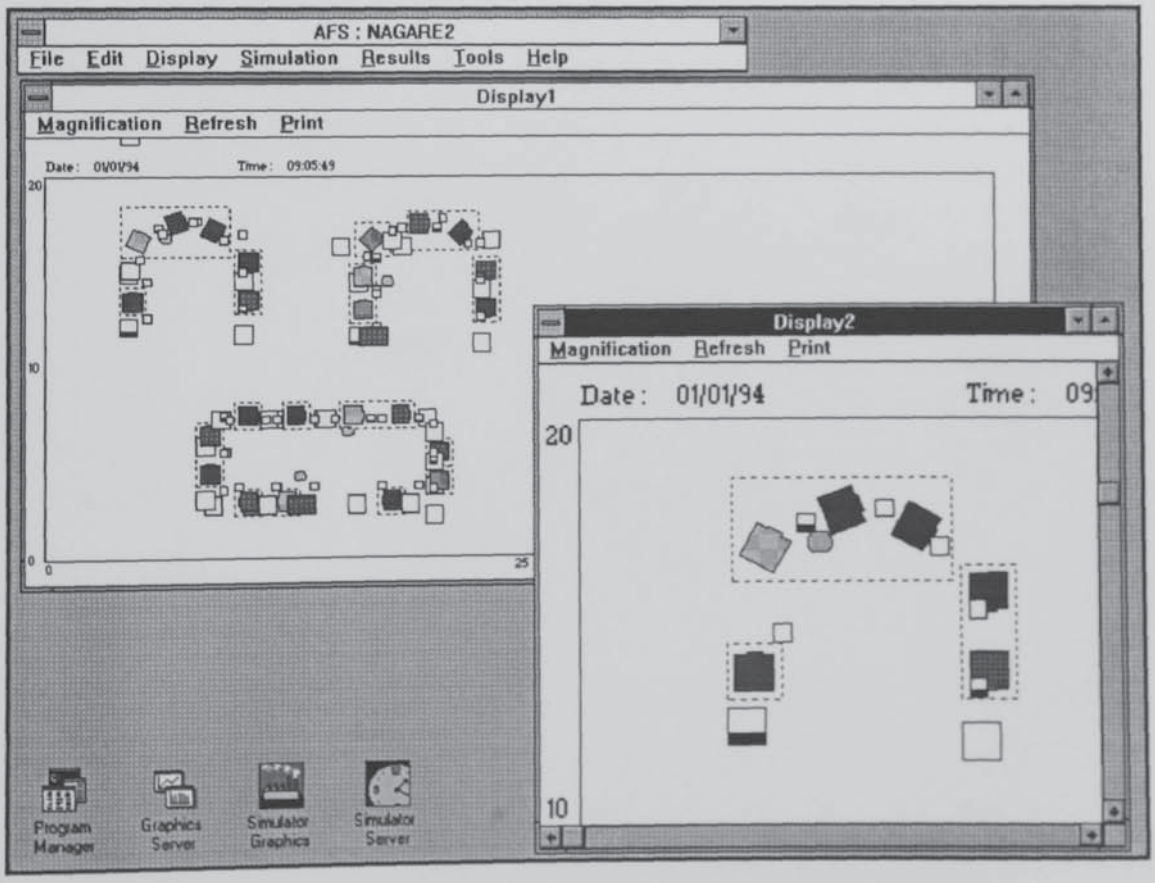
Model	Approx. entities	Model purpose
Nagare	200	To demonstrate complex part and operator movement and user-interface capabilities

Model description

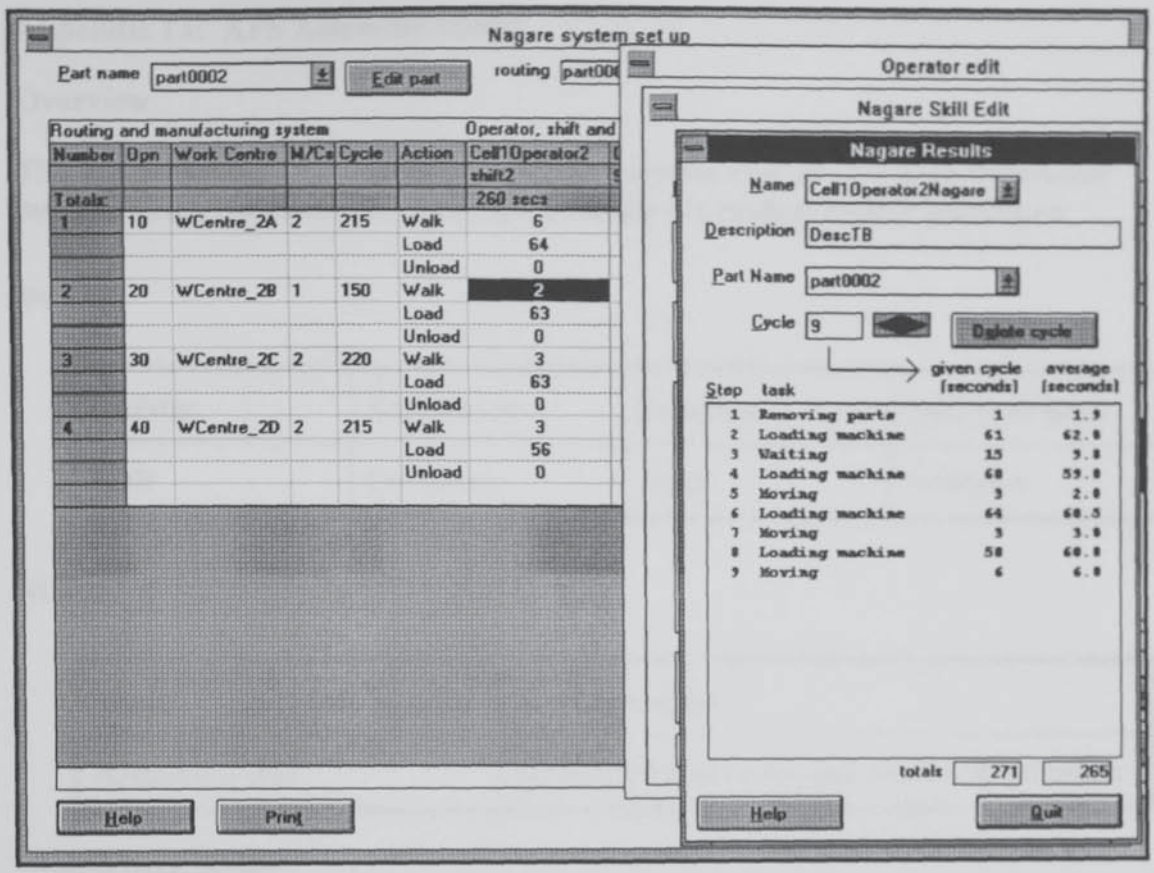
The model took about an hour to build using a combination of text files and dialogs. The result was a model containing three 'U' shape or Nagare cells. The first two cells consisted of at least three work centres and six machines whilst the third cell contained nine work centres and eleven machines. The first two cells were each run by a single operator. The third cell used two operators each carrying out different operations on the parts. Batches were released into each cell. The cells were gradually primed, run at steady state then gradually emptied. No attempt was made to simulate change overs between different parts. The Nagare system dialog was used to assist in the model building process whilst multiple displays were used to display different parts of the model during simulation. The results that can be generated are either aggregate (for example, total time a machine spent running during a day) or cyclic (for example, the activities of an operator from removing parts to loading machine to removing parts to waiting to loading machine, etc.). The following three screen dumps illustrate the Nagare model building dialog, the graphical displays and the cyclic results.



The Nagare model building dialog



Using multiple displays to view the model simulation



Comparision of static calculations and cyclic simulation results for the second cell

Appendix 13: AFS Assembly Model

Overview

This model was built by the developer to demonstrate a model of a more 'traditional' manufacturing system and to examine the simulator's random number generators.

Details

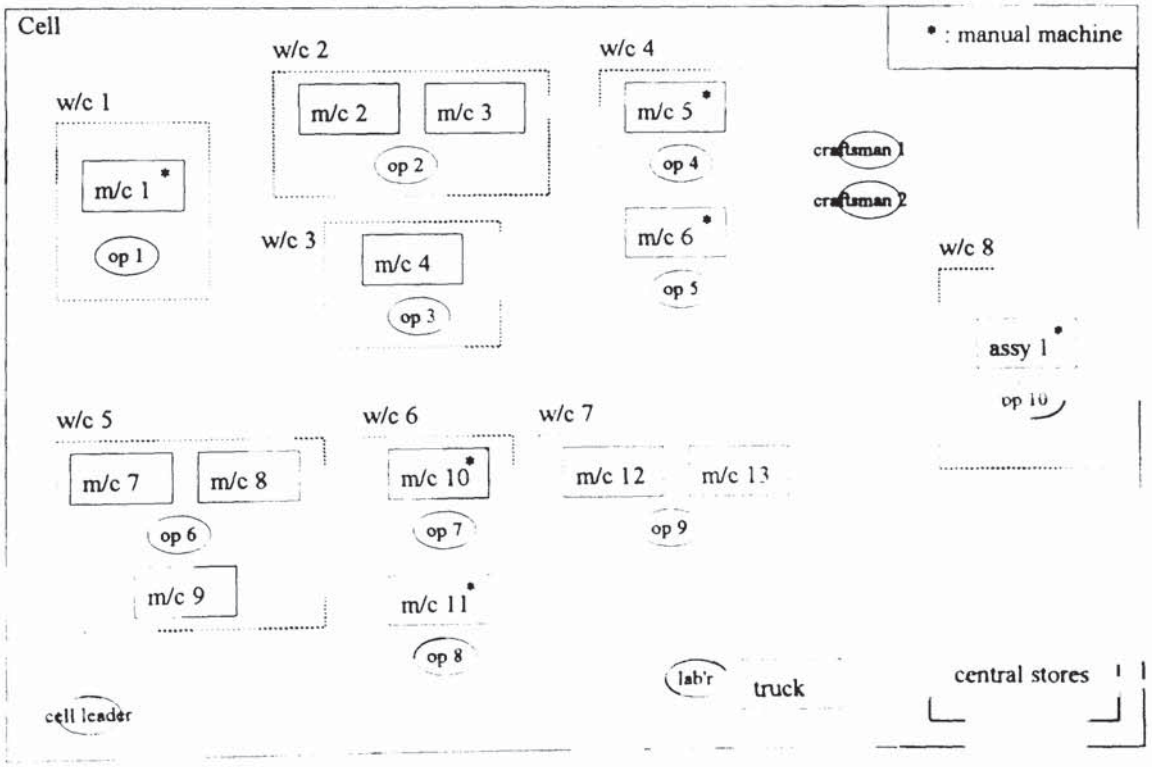
Builder	Experience	Location	Source of data
PDB	Developer	Aston	Synthetic

Model

Model	Approx. entities	Model purpose
Assembly	200	Demonstrate assembly and stochastic behaviour

Model Description

The model, which took about an hour to build, in shown below.



A schematic of the model showing machine groupings

It consists of a number of work centres each containing a number of machines. Operators ran one or more machines. For example, Operator2 ran Machine2 and Machine3. Most operators could only run machines. Two operators added to carry out

change over and repair duties and one operator was added to carry out transportation duties.

Batches were processed either by the top four work centres (1-4) or the bottom three (5-6) before being manually assembled at leftmost work centre (8). The routings and assemblies are shown in the tables below. Component manufacture was scheduled at work centres 1 and 5 and assemblies were scheduled at work centre 8. Operators worked strictly to the schedules.

Part name	Operation	Work centre	Operation
PartOne	10	1	machine
"	20	2	machine
"	30	4	machine
PartTwo	10	1	machine
"	20	3	machine
"	30	4	machine
PartThree	10	5	machine
"	20	6	machine
"	30	7	machine
PartFour	10	5	machine
"	20	6	machine
"	30	7	machine
PartFive	10	8	assembly
PartSix	10	8	assembly

The routings

Parent	Child	No. Off
PartFive	PartOne	2
PartFive	PartThree	1
PartSix	PartTwo	2
PartSix	PartFour	3

The assemblies

An initial simulation was carried out to assess the typical output of the manufacturing system. The capacity for a given mix of orders enabled the calculation of a daily schedule shown below.

Part	Batch quantity	No. batches
PartOne	60	2
PartTwo	40	3
PartThree	30	2
PartFour	60	3
PartFive	30	2
PartSix	20	3

The daily production schedule

All orders are released at the start of the shift. Whilst it is appreciated that such a schedule would not be used in practice its purpose was to necessitate frequent change overs and short batch lead-times.

Machines and operators possess a number of random number generators. For example, a machine has generators attached to the load and unload times and the breakdowns Varying types of random number generators were used. Whilst various parameters were set for each of the generators a brief summary of typical values is shown below.

Functionality	Parameter	Typical setting
Machine	Load / unload	15 - 35 seconds
"	Breakdown frequency (interval between failures)	either 60 - 120 minutes or 50 - 90 parts
"	Breakdown severity (time to repair)	5 - 30 minutes
Operator	Efficiency	70 - 90%

The random number generator settings

The experiments

Two experiments were carried out:

- A test of repeatability;
- A test of variation in lead times for the particular generator settings.

The first test would assess whether the simulator could reproduce results whilst the second would test the operation of the stochastic mechanisms.

The model run lengths were set at one day, within which the operators work an eight hour shift. Since the objective of these tests were to assess the stochastic elements of the simulator, not the model itself, attainment of steady state was not deemed necessary. Observations of the model showed that at the end of each shift there was work in progress (WIP) at various work centres. There was no significant accumulation of WIP.

The tests were based on the comparison of lead times. Because WIP was present at the end of shifts and not completed until the next shift, the lead times for some batches included the sixteen hours between shifts and in some cases the two days for the weekend. In the process of manufacturing system design and analysis the inclusion of the between shift time in the batch lead times is necessary to obtain a valid model. For the purpose of assessing the performance of the *simulator* it introduces complications. The graphical results that follow have therefore been adjusted to present a lead time based on operator working time not elapsed time.

Between two and three batches for each part were processed each day. The orders were released at the start of the shift and the lead time was measured from order release to completion, not start of batch processing to batch completion. If several orders are released at a work centre at the same time the lead times for orders will vary widely as one is processed immediately whilst others wait. Due to this initial bias, comparison of lead times between batches processed on the same day was not appropriate; only the lead times of like batches from different days were compared. For example, see the result table below. Two batches of PartOne were released each day. Typically the lead time for first batch was 6 hours and 25 hours for the second batch. Comparison of these two lead times would be misleading

Each simulation run was for two weeks (ten working days). For the first run the random number generator seeds were set to the initial seeds. For the second run the model was again reset to initial seeds. Comparison of these two runs provided a test of the repeatability. Four subsequent runs were carried out. For each of these runs all the initial seeds of random number generators were set to fifth, tenth, fifteenth and twentieth numbers in the stream (N.B. each source of uncertainty has its own random number generator initialised with a unique seed).

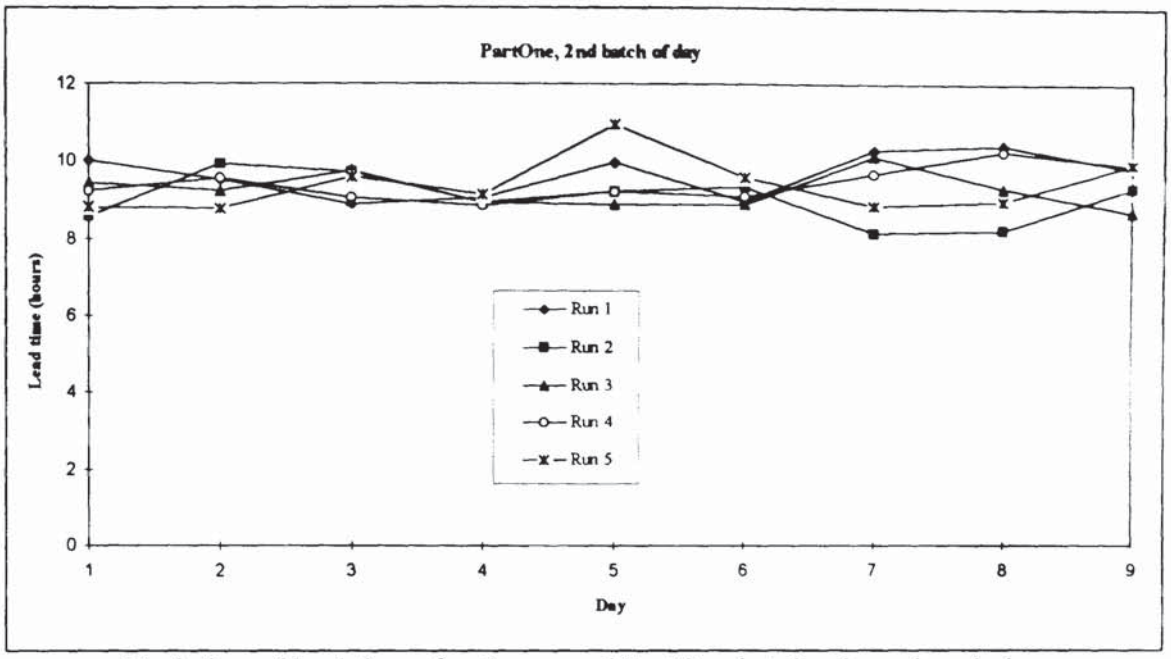
The results

As expected the first and the second run provided exactly the same results. The lead times for the first run are shown below. Times are in hours and in order of completion.

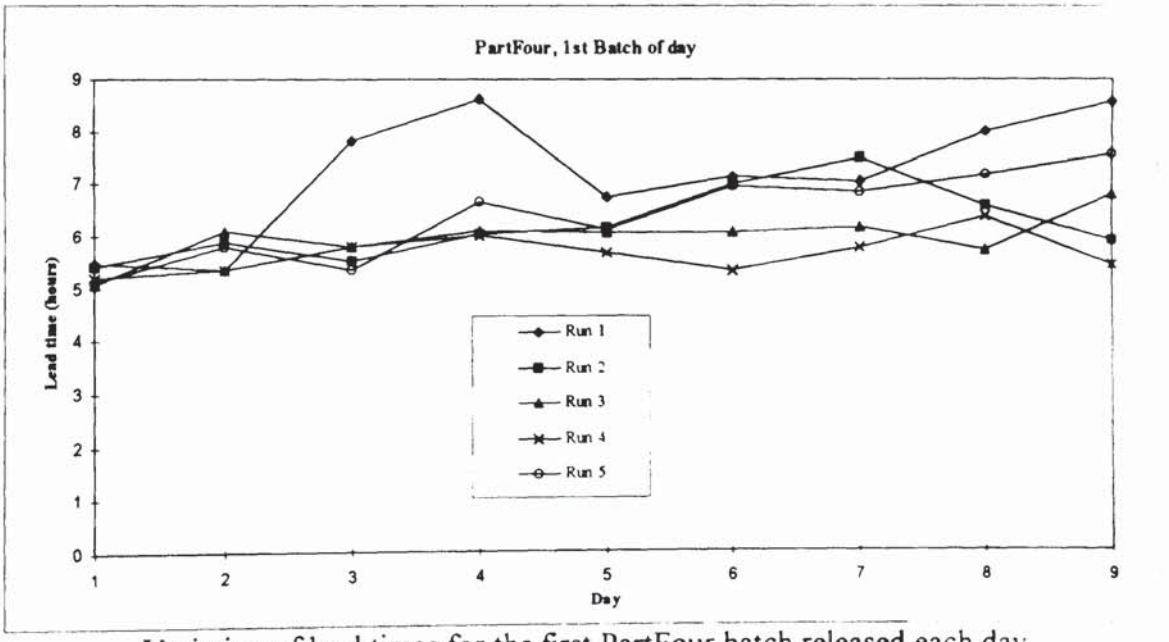
PartOne	PartTwo	PartThree	PartFour	PartFive	PartSix
5.94	3.62	6.68	5.46	17.32	0.92
26.03	7.33	25.38	24.37	1.41	0.94
6.02	26.89	6.47	27.68	17.39	0.96
25.52	3.90	26.26	5.36	1.44	0.93
5.69	7.10	5.88	24.92	1.43	0.94
24.88	26.84	26.05	28.33	1.42	0.96
5.88	4.16	6.14	7.82	1.39	16.93
25.07	6.78	27.07	24.95	1.38	0.92
5.93	26.09	7.90	28.12	1.36	0.95
73.98	3.88	74.97	24.60	1.39	0.93
6.03	7.54	24.09	25.63	1.43	0.91
24.97	26.36	27.89	29.22	1.45	0.95
6.16	3.58	24.01	6.74	1.39	0.94
26.30	7.38	27.48	73.36	1.39	0.93
6.07	74.96	24.65	77.61	1.41	0.94
26.44	3.95	28.75	7.12	1.41	0.92
5.84	7.32	25.67	26.75	1.45	0.93
25.79	27.04	28.84	29.71	1.43	0.93
6.41	4.22	0.00	7.05		0.93
0.00	7.24	0.00	25.78		0.93
	24.12		30.25		0.93
	4.26		7.98		0.95
	7.97		27.65		16.93
	24.93		30.82		0.92
	4.32		24.54		0.92
	7.94		27.77		0.53+
	26.83		30.96		
	3.59		0.00		
	7.66		0.00		
	0.00		0.00		

Note 1: lead times over 24 hours cross shifts.
Note 2: lead times over 72 hours cross weekends.
Note 3: lead times of 0.00 or with + are incomplete.

The results were adjusted before analysis. Two typical graphs are shown below. The first graph shows the variation in lead time for the second batch of PartOne released each day. The second graph shows the variation in lead time for the first batch of PartFour released each day. With the exception of sharing repair and change over personnel the manufacture of PartOne is independent of PartFour.



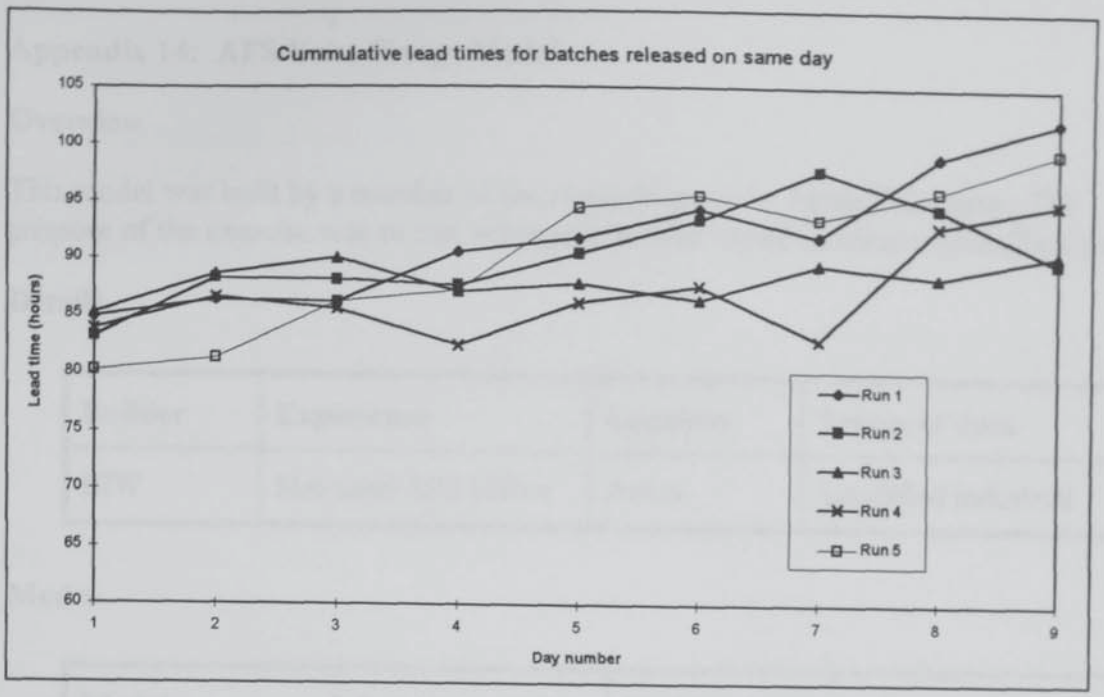
Variation of lead times for the second PartOne batch released each day



Variation of lead times for the first PartFour batch released each day

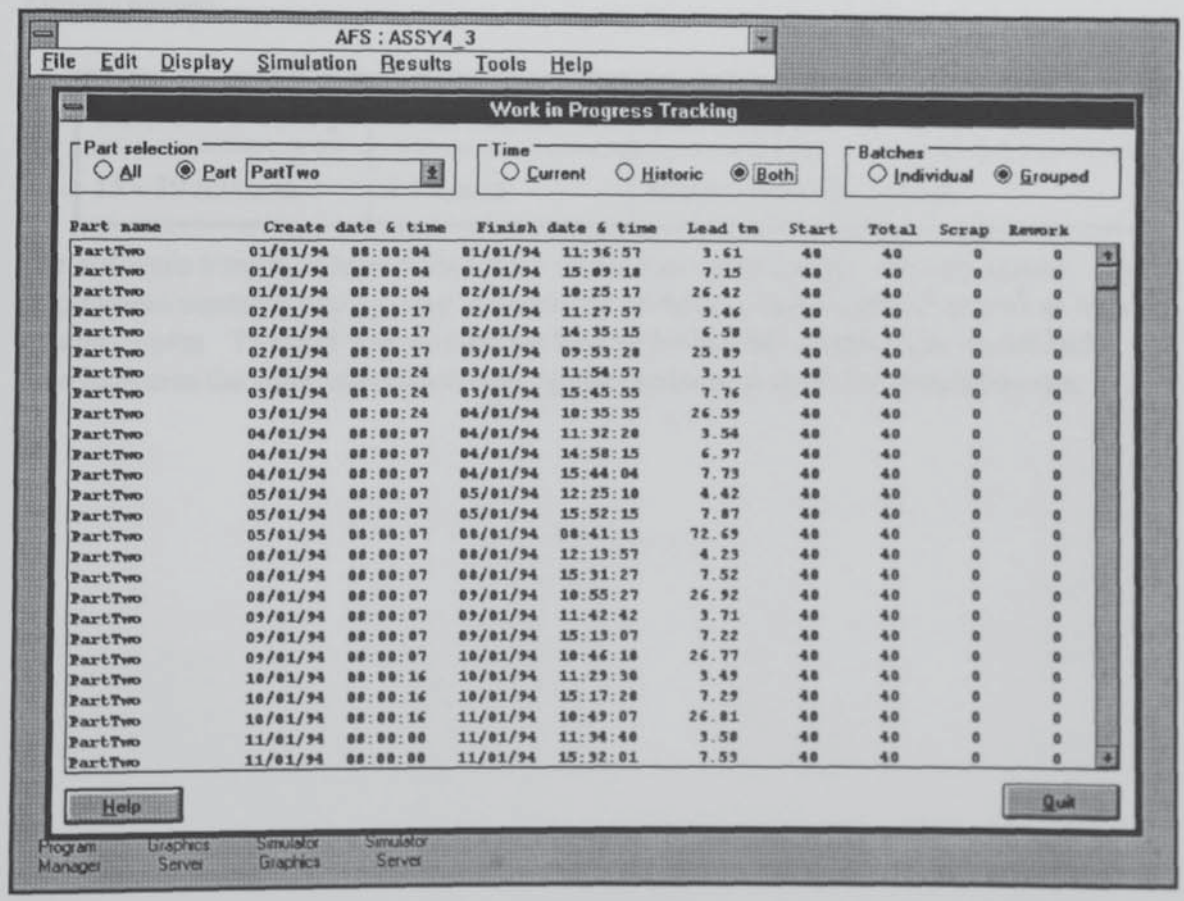
The above two graphs are typical of the results collected. They demonstrate that AFS has the ability to simulate the same model with different random number streams and produce different results. It is judged that the key influences on the variation is due to breakdowns, repair and changeovers. Minor influences, as seen from the table of results for assembly of PartFive and PartSix, arise out of operator efficiency variations.

As a separate analysis the lead times for all batches that were released on the same day were summed. The cummulative lead times for each of the simulation runs are shown below.



The model building and subsequent simulation also demonstrates the ability of AFS to model batch manufacture and assembly as well as change overs and machine reliability.

The screen dump below shows how the user would see the presentation of the lead time results. The dialog shown displays information collected by the WIP tracker object.



Part name	Create date & time	Finish date & time	Lead tm	Start	Total	Scrap	Rework
PartTwo	01/01/94 08:00:04	01/01/94 11:36:57	3.61	40	40	0	0
PartTwo	01/01/94 08:00:04	01/01/94 15:09:18	7.15	40	40	0	0
PartTwo	01/01/94 08:00:04	02/01/94 10:25:17	26.42	40	40	0	0
PartTwo	02/01/94 08:00:17	02/01/94 11:27:57	3.46	40	40	0	0
PartTwo	02/01/94 08:00:17	02/01/94 14:35:15	6.58	40	40	0	0
PartTwo	02/01/94 08:00:17	03/01/94 09:53:28	25.89	40	40	0	0
PartTwo	03/01/94 08:00:24	03/01/94 11:54:57	3.91	40	40	0	0
PartTwo	03/01/94 08:00:24	03/01/94 15:45:55	7.76	40	40	0	0
PartTwo	03/01/94 08:00:24	04/01/94 10:35:35	26.59	40	40	0	0
PartTwo	04/01/94 08:00:07	04/01/94 11:32:20	3.54	40	40	0	0
PartTwo	04/01/94 08:00:07	04/01/94 14:58:15	6.97	40	40	0	0
PartTwo	04/01/94 08:00:07	04/01/94 15:44:04	7.73	40	40	0	0
PartTwo	05/01/94 08:00:07	05/01/94 12:25:10	4.42	40	40	0	0
PartTwo	05/01/94 08:00:07	05/01/94 15:52:15	7.87	40	40	0	0
PartTwo	05/01/94 08:00:07	08/01/94 08:41:13	72.69	40	40	0	0
PartTwo	08/01/94 08:00:07	08/01/94 12:13:57	4.23	40	40	0	0
PartTwo	08/01/94 08:00:07	08/01/94 15:31:27	7.52	40	40	0	0
PartTwo	08/01/94 08:00:07	09/01/94 10:55:27	26.92	40	40	0	0
PartTwo	09/01/94 08:00:07	09/01/94 11:42:42	3.71	40	40	0	0
PartTwo	09/01/94 08:00:07	09/01/94 15:13:07	7.22	40	40	0	0
PartTwo	09/01/94 08:00:07	10/01/94 10:46:18	26.77	40	40	0	0
PartTwo	10/01/94 08:00:16	10/01/94 11:29:30	3.49	40	40	0	0
PartTwo	10/01/94 08:00:16	10/01/94 15:17:28	7.29	40	40	0	0
PartTwo	10/01/94 08:00:16	11/01/94 10:49:07	26.81	40	40	0	0
PartTwo	11/01/94 08:00:00	11/01/94 11:34:40	3.58	40	40	0	0
PartTwo	11/01/94 08:00:00	11/01/94 15:32:01	7.53	40	40	0	0

The WIP tracker dialog displaying information of batches of a selected part

Appendix 14: AFS Yoke Flange Model

Overview

This model was built by a member of the research group at Aston University. The purpose of the exercise was to test whether a 'novice' could develop a predefined model.

Details

Builder	Experience	Location	Source of data
IEW	Not used AFS before	Aston	Simplified industrial

Model

Model	Approx. entities	Model purpose
Yoke flange	200	To demonstrate speed and ease of use as well as some of the functionality

Times (hours)

Software training	Model build	Simulation
15 - 30 minutes	2.5 hours	about 1 hour for 1 week

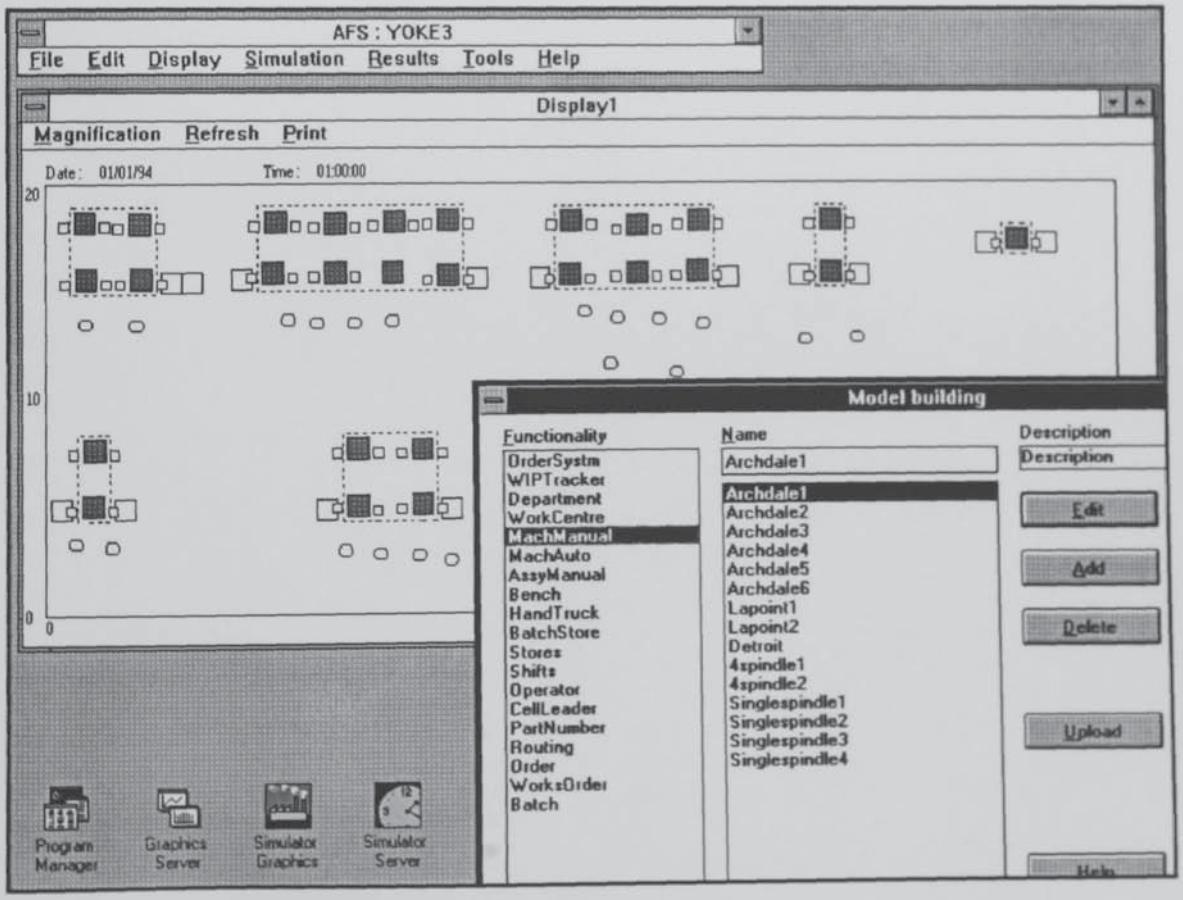
The software training covered the basics of the user-interface and the help system. The help system contains step-by-step instructions on how to build a model as well as help on specific topics. The user was *not* shown how to build a full model. The model build time indicates the time taken from starting the model to end of first simulation run.

Model description

The model consists of seven work centres, each containing a number of machines. Operators in each work centre run one or more machines. There are a mixture of manual and semi-automatic machines. The seven work centres are used to manufacture one component (a flange) in high volume. The table below shows a summary of the data used to build the cell.

Opn	Operation description	Machine type	No. m/cs	Staffing m/cs / oper	Production rate /hr / m/c
10	Turn OD	BSA 6x20 lathe	4	2	13
20	Turn face & spigot	Ryder Auto Wickman Auto	4 4	2 2	6 6
30	Drill, bore & ream cross holes	Archdale	6	1	12
40	Broach cross hole	Lapointe	2	1	29
50	Broach cross hole faces	Detroit chain broach	1	1	60
60	Drill bolt holes	4 spindle drill	2	1	19
70	Turn lock-ring groove	1 spindle drill	4	1	19

The following screen dump shows a picture of the model built prior to simulation.



Comments

IEW was able to build the pre-defined model quickly and easily. It took approximately two and a half hours from start of model build to end of the first pass simulation run. It must be noted that this was the first time that IEW had used the software and had not been shown the model building process before. Whilst some help was given, much of it was related to minor software faults identified during the model building process. IEW later spent several more hours building detail into the model that was previously covered by default values.