An Investigation into the Use of Knowledge Representation in Computer-Integrated Manufacturing

Simon Anthony Sherman Beesley

Submitted for the degree of Doctor of Philosophy

ASTON UNIVERSITY

September 1988

Aston University

An Investigation into the Use of Knowledge Representation in Computer-Integrated Manufacturing

Simon Anthony Sherman Beesley

Submitted for the degree of Doctor of Philosophy

1988

In a certain automobile factory, batch-painting of the body types in colours is controlled by an allocation system. This tries to balance production with orders, whilst making optimally-sized batches of colours. Sequences of cars entering painting cannot be optimised for easy selection of colour and batch size. 'Over-production' is not allowed, in order to reduce buffer stocks of unsold vehicles. Paint quality is degraded by random effects.

This thesis describes a toolkit which supports IKBS in an object-centred formalism. The intended domain of use for the toolkit is flexible manufacturing. A sizeable application program was developed, using the toolkit, to test the validity of the IKBS approach in solving the real manufacturing problem above, for which an existing conventional program was already being used. A detailed statistical analysis of the operating circumstances of the program was made to evaluate the likely need for the more flexible type of program for which the toolkit was intended.

The IKBS program captures the many disparate and conflicting constraints in the scheduling knowledge and emulates the behaviour of the program installed in the factory. In the factory system, many possible, newly-discovered, heuristics would be awkward to represent and it would be impossible to make many new extensions.

The representation scheme is capable of admitting changes to the knowledge, relying on the inherent encapsulating properties of object-centred programming to protect and isolate data.

The object-centred scheme is supported by an enhancement of the 'C' programming language and runs under BSD 4.2 UNIX. The structuring technique, using objects, provides a mechanism for separating control of expression of rule-based knowledge from the knowledge itself and allowing explicit 'contexts', within which appropriate expression of knowledge can be done. Facilities are provided for acquisition of knowledge in a consistent manner.

Keywords: FMS; IKBS; message-passing; object-centred; scheduling

I would like to acknowledge the advice of Dr. Peter Coxhead throughout this work and his constant encouragement. I am grateful to Messrs. Ian Soady and Brian Johnson of Istel, plc, for explaining details of the existing factory paint system and sparing time to help me understand their scheduling scheme.

Al mio piccolo Edwin ed alla mia carissimma Clelia, per la sua pazienza e simpatia.

Diagrams and tables:

Chapter 1 Artificial Intelligence in the C(omputer)-I(ntegrated) M(anufacturing) context

In manufacturing control, many processes appear to be difficult to represent adequately by a computer model. The need to simulate real-time activities may pose many problems. In circumstances where human judgement is required the problems may prove intractable for a traditional, procedural style of programming. These apparent difficulties prompted this work and form the basis for the discussion in Chapter 3.

1.1 The significance of AI for industry

Human operators controlling a complex process may not be able to respond sufficiently quickly to solve problems contemporaneously with the process. Several types of control that are of interest to people wishing to use 'intelligent' computer programs may be distinguished.

The first class might be scheduling programs which produce a static plan for perhaps one or more production shifts, i.e. deciding, from a given input of demand plus materials, what output - finished products plus their order of manufacture - there ought to be. A classic example of this might be in a factory assembling and/or machining parts. The output of the program is a plan of how the manufacturing cells process the parts and in which order they act. The input to this system would be the orders required and the 'raw' materials. This is a characteristic use for planning systems in manufacture, where a once-and-for-all plan is made at discrete intervals. The job-shop scheme, *SOJA*, described by le Pape (1985) is an example of a daily-used scheduler. Although this is a rather arbitrary categorisation, it facilitates drawing the distinctions made later in this and the following chapters.

A second type of industrial requirement is the on-line control system where the aim is to maintain some state of the plant as a whole. Oil refineries, for example, may have to control some system that reacts to changes from a desired state and diagnoses a difference between the observed and expected states, forming a strategy for returning the actual state to that required. As an IKBS the approach would typically be to plan an ideal route back to the required state. Such systems would normally have knowledge of many types and cover as much of the plant as possible. The *REACTOR* program of Nelson (1982) has such mixed knowledge used in real-time.

A third class of applications is to control a case intermediate between the first two. In this the intention is to control particular processes which require constant adjustment. They may involve a small amount of planning but would usually produce a new decision each time they responded. The reactive scheduler of Elleby *et al.* (1988) uses this type of decision making. The difference between the second and third classes - beyond that of scale - is that the second class requires each part of the system to produce a response in line with the whole, whereas the third is appropriate in circumstances where it co-operates with the whole's objectives but acts independently (as a localised controller). The system described in Section 1.2 is of this third type. In such a case, the behaviour is to minimise the cost of the processes that precede it in sequence.

The above categorisation divides systems in terms of how they do what they do. Another aspect is whether they 'close the loop' in controlling a system, as opposed to offering advice to a human operator. This is an important functional difference. An error-detecting system may be in control of the means of correcting the state or may just raise an alarm and inform observers as to the likely cause of the problem. The system investigated here is one where direct control is needed.

1.2 An example of a complex Flexible Manufacturing System

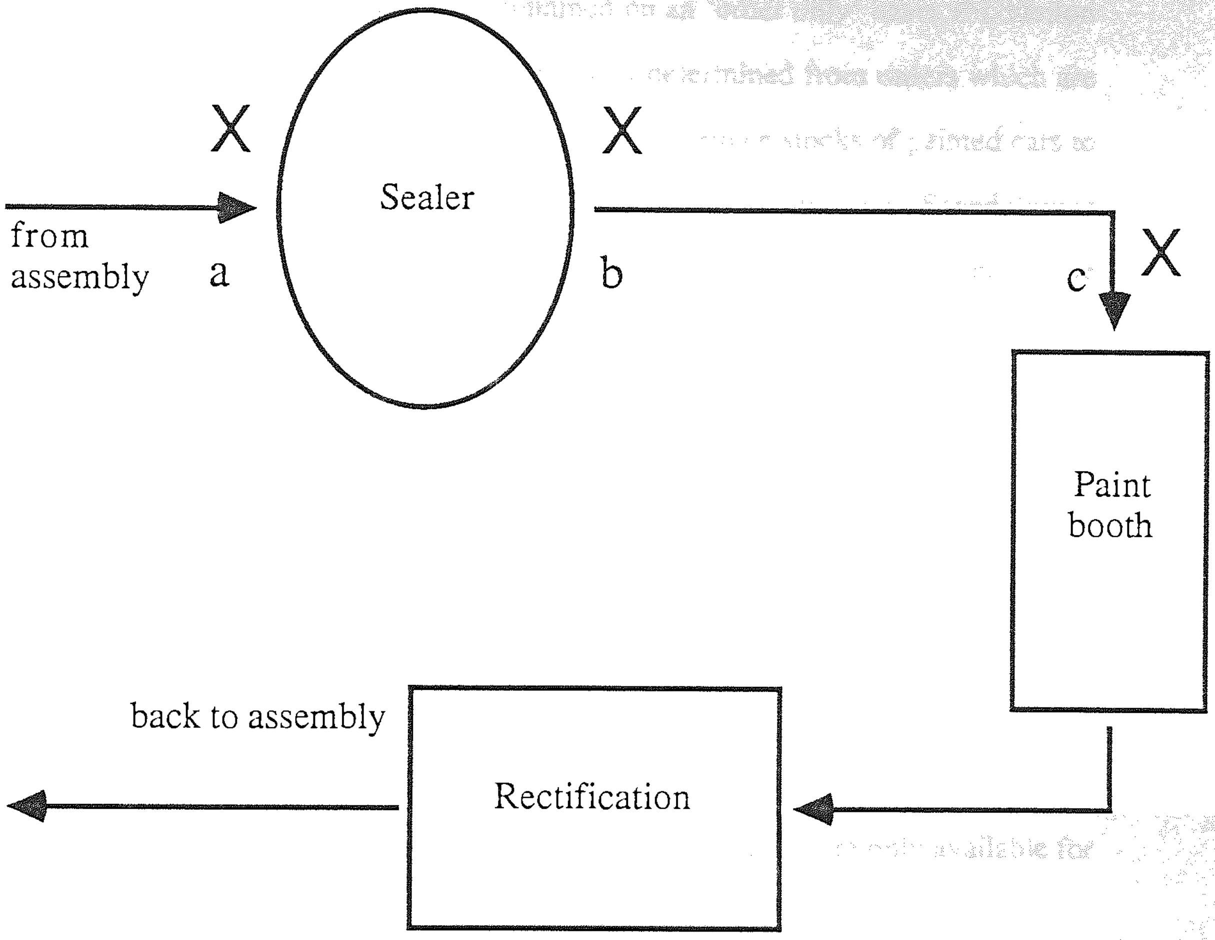
This section describes a scheduling problem, for which an algorithm has been implemented by a major car manufacturer in the U.K. - as part of a wider CIM suite of programs written to control an automated motor assembly line.

1.2.1 Paint Scheduling

A major car manufacturer has installed a fully automated plant controlling system in one of its assembly lines. It connects assembly line robots and sensors with plant-tracking, stock control, warehousing, order and other sub-systems. As part of the tracking scheduling it was decided to extend computer control to the selection of colour for car bodies being painted.

The section where paint is applied is supplied with car bodies less doors - B(ody) I(n) W(hite) - by conveyor from the assembly section. After painting inspection, bodies are removed for further stages of assembly. The BIWs form a queue (for scheduling purposes) and pass through heat sealing, cooling and then the paint booth, amelioration being required, in some cases, before removal. These stages are automated and, in optimal conditions, the robots will be able to spray BIWs without human intervention.

The BIWs are tagged as they pass through the plant and can be tracked at various stages by the tracking scheduling sub-system which, in theory, has a record of any car body present in the paint section. A BIW is given a unique number (ident) with which is associated the respective code for its body type (e.g. saloon, 3-door etc.). (See Figure 1.1)



X 'ident' point

a - b: region where BIW shells may be re-arranged

a - c: region of conveyor with contents 'visible' for scheduling

Layout of assembly line in the paint shop

Figure 1.1

1.2.2 Production

The production of bodies is maintained on an 'order only' basis. Production schedules are created and (painting) order cover determined from orders which are collected and held centrally. (The basis for this is to reduce stocks of painted cars to just those that have been ordered - at any time - from distributors etc.) Scheduling is on an 18 day basis. From the total order 'bank' *selected orders* are generated for the paint section (and the production of car bodies is maintained roughly in accordance with them).

1.2.3 Order cover

Requirement for painting is assessed for each possible body and colour combination. Up to 32 bodies x 32 colours might be used. A matrix of these (body x colour) would have less than or equal to 50% occupancy - many combinations not being used. The distribution is such that some body types are produced in many of the colours and others in few colours. (Similarly some colours are only available for a few types.)

Requirements for the possible body/colour combinations are assessed in terms of the 18 day rolling schedule: from *today minus 9* to *today plus 8* (days), in order to smooth out production. (This means that BIWs should be painted for real orders not more than two weeks old.) *Achievements* from orders in the schedule are calculated by summing production of painted bodies over the interval *today minus 9* to *today*. Updating takes place by dropping the 'oldest' day from the schedule and adding the change in requirement (one day's worth) usually on a daily basis although the system must be able to cope with an update after as short a space as a quarter of an hour.

1.2.4 Body sequence in the paint section

The system can map the details of perhaps 80 (but potentially more) BIWs, from the one entering the paint section up to that at the paint booth. The 20 or so

BIWs at the 'tail' of this queue may be 're-positioned' (by overhead cranes). The system should be able to keep track of sequence changes if correctly informed by users but will only be able to restore its map of the body population when a 'moved' BIW passes a tracking sensor. (The consequences of changing the sequence will not be discussed further but they complicate measures taken to deal with colour sequence problems.)

1.2.5 Colour allocation

The task of colour selection is to *commit* the *n* BIWs nearest the paint booth to a particular colour for spraying. (BIWs are *achieved* and update the achievement record after passing through the paint booth.) The problem of colour selection might seem to be trivial: however the case here is that (i) BIWs do not enter 'paint' from 'assembly' in an order favouring a simple matching of order cover to a colour and allocating the next *n* BIWs to a batch; (ii) car bodies may only be painted in colours for which order cover exists. If a decision has not been made when a BIW comes into the paint booth, then a possible colour is found for it, together with the following BIWs in the sequence, sufficient to make a batch of suitable size. Thus when constraints exist to prevent the next BIW being painted in its committed colour, another colour must be selected.

1.2.6 The algorithm

The algorithm favoured by the system designers, Istel (1985), was to find an *a(chievement) p(roportion)* for a batch of *n* bodies for each colour available and select that colour with the lowest score. The following method was used: from the sequence map, a batch of bodies is scanned starting from the paint booth. At least *minpb* (the least preferred batch size) for the colour must be possible. Given minpb the batch is extended one by one if the next BIW can also be painted in the colour (i.e. order cover exists for that body/colour combination). This is done until *maxpb* (the greatest preferred batch size) is reached or order cover is not available for any body. When all colours have been assessed, the batch a.p. is calculated for each, where a.p. = Requirement in colour / Achievements in colour. Achievements implies those BIWs painted from order cover up to date *plus* the *n* BIWs in the proposed batch - thus favouring the biggest batch size relative to (prior) least achievement.

1.2.7 Other constraints

Priority body/colour combinations (up to 9) of the form *a* bodies in colour b for orders may be specified. Allocation of colour considers these, if any, first, before the above algorithm. An attempt is made to achieve the priority in a batch, subject to order constraints.

Colours may be made unavailable while processing a batch, leaving 'committed' bodies unachievable at the paint booth. Sometimes a new colour decision is impossible so the first tactic is to try to extend the batch in the current colour where appropriate. This is done until order cover is exhausted or the greatest allowed batch size has been reached (whichever is sooner). *Crisis* re-allocation rules come into effect (similar to the algorithm) to attempt to find a valid colour for a batch within the existing constraints.

1.3 Reasons for using Intelligent Knowledge-Based Systems

This solution could present many drawbacks in 'real-world' operation, in particular being potentially vulnerable to conditions where awkward (i. e. non-ideal) input might occur. Another aspect which may cause difficulties is that the method chosen is not susceptible of accommodating new (types of) constraints on *performance*. Some such constraints *(vide infra)* have been acknowledged as likely to exist, although not modelled in the existing program. Were it to become desirable that the program be able to handle such new factors, it would seem to be a difficult body. When all colours have been assessed, the batch a.p. is calculated for each, where a.p. = Achievements in colour / Requirement in colour. Achievements implies those BIWs painted from order cover up to date *plus* the *n* BIWs in the proposed batch - thus favouring the biggest batch size relative to (prior) least achievement.

1.2.7 Other constraints

Priority body/colour combinations (up to 9) of the form *a* bodies in colour b for orders may be specified. Allocation of colour considers these, if any, first, before the above algorithm. An attempt is made to achieve the priority in a batch, subject to order constraints.

Colours may be made unavailable while processing a batch, leaving 'committed' bodies unachievable at the paint booth. Sometimes a new colour decision is impossible so the first tactic is to try to extend the batch in the current colour where appropriate. This is done until order cover is exhausted or the greatest allowed batch size has been reached (whichever is sooner). *Crisis* re-allocation rules come into effect (similar to the algorithm) to attempt to find a valid colour for a batch within the existing constraints.

1.3 Reasons for using Intelligent Knowledge-Based Systems

This solution could present many drawbacks in 'real-world' operation, in particular being potentially vulnerable to conditions where awkward (i. e. non-ideal) input might occur. Another aspect which may cause difficulties is that the method chosen is not susceptible of accommodating new (types of) constraints on *performance*. Some such constraints (*vide infra)* have been acknowledged as likely to exist, although not modelled in the existing program. Were it to become desirable that the program be able to handle such new factors, it would seem to be a difficult and highly complicated task to modify the existing code to account for these. Such considerations prompted this work.

Is it possible to produce an 'incremental' style of program capable of dealing with the above model and problems? By incremental is meant an A(rtificial) I(ntelligence), Knowledge Based form of modular code - with 'chunks' of knowledge separated out rather than embedded in the controlling logic. The work reported here is an attempt to build such a program, which is capable of performing the functions of the 'procedural' implementation and which can also be instructed to respond to new operating conditions. There are two areas to be discussed in this latter respect: firstly, is it possible to provide an adequate representation which is much more robust than the original from the point of view of extensions to its functional repertoire; secondly, how is extra knowledge to be acquired? The research issues might have involved some level of automation of the acquisition process with respect to allowing a non-technical user of the scheduling system to input the 'raw' new knowledge. (This is not intended to lead to much consideration of natural language processing, the internal handling of knowledge is more the question in this context.)

The system chosen for study may be limited in scope in terms of the factory. In itself, however, it poses complicated problems and is seen as a proxy for more general problems of applying 'intelligent' programming techniques in industry and CIM in particular. One of the difficulties in adopting intelligent CIM is the common need to alter the likely action of one or more sub-systems in an IKBS in response to dynamic changes in system requirements. The problem dealt with here is of that nature. A reactive scheduler, which is responsive to the local state of the factory and can improve overall performance without requiring the participation of other sub-systems, would be useful for more distributed control systems. The colour allocation task, in this context, could provide a useful example of how to achieve IKBS in industry.

Chapter 2 The context of AI and paint scheduling

There are two areas of research that are relevant to the work discussed here. Firstly, in the last few years there has been considerable interest in applying knowledge-based and, particularly, expert systems to control/planning in industry. Secondly, there is the more 'mainstream' theme of representation schemes and their implications for knowledge acquisition.

2.1 'Intelligent' programming for industry

The use of computers and micro-processors in process control is quite well established. Most process control is based on mathematically modelling the systems controlled. An illustration of the significance of mathematical models of processes is the proportion of papers in conferences on Computer Aided Design which describe software for this use. The systems described are used to design industrial controllers, etc. Of some nineteen CAD systems presented in the Third IFAC/IFIP Symposium on CAD in Control and Engineering Systems, all discuss linear or non-linear techniques. In the same conference of some seventy-six papers just two present knowledge-based techniques for designing and implementing control systems.

2.1.1 Industrial IKBS

An interesting example of the possible use of heuristics is the saw-mill planner of Harris and Zinober (1988). Their problem was to plan the sequence of cuts in large boards required to make sets of smaller panels. This is essentially a constraint satisfaction problem modified by a cost function, the aim being to minimise wastage of wood and time taken for the automated saw to cut it: formally similar to the well-known travelling salesman problem. A linear solution, for example using the Simplex algorithm, was solvable on the hardware in weeks, whereas the time allotted was of the order of minutes. Their approach was to use a variation of a branch-and-bound technique. The requirement was not for an optimisation so the first 'acceptable' solution would not necessarily be that produced by the full linear technique. The point at which a heuristic approach was thought to be helpful was in further opportunistic pruning of the search space - a facet not represented in their program. They stated their view that heuristics could be inferred by recording and comparing the success of various actions. This work is significant in that it shows the limitations of mathematical approaches to problems of exponential complexity.

Some authors have described 'knowledge-based' controllers which claim to model inexact (qualitative) descriptions of the system being controlled. Production rules are seen by some as a good formalism. It is not clear that the use of production rules, e.g. as defined by Buchanan and Shortliffe (1984), necessarily confers 'expert' status on a program. There is a fair literature on 'fuzzy controllers', see e.g. Mamdani (1982). This paper presents a proposal to interpret natural language descriptions of rules pertaining to the control of a process in order to formalise them into some computer intelligible format. The concept of 'fuzzy' descriptions is that attributed to the work of Zadeh (1979) as an extension of set theory. The idea of a degree of belonging to some category is used by Mamdani to attach evaluable status to an inexact collection of verbal terms. These modify the patterns of applicability for rules (mentioning them in their premises) which govern the behaviour of the controller. The degree of matching modifies the degree of assertion of the conclusion of the rules. This might seem laudable, but there are difficulties with this fuzzy approach. Humans are notably inconsistent in using terms like 'rather', 'very', 'small'. Also two people may disagree on such terms - so their value is reduced if more than one expert is consulted. The claim is that inexact descriptions do work. In human terms possibly so, but what manipulations are required for computer descriptions to work? In his paper, Schefe (1980) has a discussion of the nebulous concept of fuzzy boundaries and the uncertain subjective nature of linguistic interpretation.

Given that mathematical models are used for process control, are knowledge-based systems (KBSs) being used in an industrial context? For the most part, KBSs found are used for scheduling, i.e. for planning production at the factory and the sectional level.

ISIS-II (Fox *et al.* 1982) allows the specification of factory production schedules, where a large number of orders compete for resources and there are very many parts made, each of which requires several stages. The types of constraints described vary from the general goods level, which relates to optimising the value of the activity, to the more specific considerations of what individual objects such as millers can do. Fox *et al.* (1983) have used the notion of conflicting constraints in determining a schedule for jobs in a factory. These constraints can be relaxed selectively at various levels of a plan and are prioritised by a rule base. Bernstein (1987) also considers a flow-shop as a planning domain for scheduling. His *SH* program is used to generate heuristics for ordering sequences of processes to attain a management goal. The heuristic rules are made from templates of three types: a) to construct a rule to build a schedule; b) to mutate rules to build better sequences; and c) to find new heuristics. *SH* does this by generating test data and discovering instances of these templates to fit them.

A closer application to the paint scheduler is the control scheme of Newman and Kempf (1985) for a robot in a manufacturing cell . The robot's plans concern a limited set of actions to achieve an immediate goal, but, once asserted, the actions are defeasible and re-planning, in effect, is possible to achieve a new goal where a previous goal has failed or only been partially attained. This is an analysis of different constraints and regards them as having unequal importance, e.g. the machines should be kept as busy as possible, whereas the robot does not have to be so busy. They report that the productivity of the cells is significantly increased over traditional schemes, such as 'first-come-first-served'. It should be noted that the failure of a plan does not cause inefficient behaviour, because the order of moves may be changed or even undone - where no destructive change to a part has been made.

Such an approach cannot be used in the painting example because the shells stay painted once 'achieved', as far as the scheduling system is concerned, and the conveyor cannot be sent backwards or its sequence re-arranged, which would be an equivalent facility to the re-ordering or undoing of robot actions.

An analogue of the reactive scheduling requirement in the paint shop, in many ways, is the control system for a laser-cutter. Foulloy *et al.* (1985) present a use of a vision system to scan the progress of a cut being made in a metal workpiece. Continual fine adjustments must be made to the laser to produce an acceptable result. Their KBS monitors state variables and alters their settings by discrete applications of rules.

KBSs for process control are found in the chemical industry. The cement kiln controller of Haspel (1985) attempts to optimise the yield for minimal heat/kiln lining consumption using qualitative methods. Rules have been devised such as 'If temperature HIGH and quality X then LOWER temperature'. HIGH is a 'fuzzy' value but denotes a range of acceptable values for the variable 'temperature'. The program may reflect a fuzzy AI approach to a technical program but the complexity of the knowledge base is rather restricted. It is not quite clear why a mathematically accurate treatment of the 'inexact' descriptions would not give equally good results. (A chemical process must obey exact thermodynamic and other laws which are well understood). A more comprehensive approach to the management of a rotary cement kiln is described by King and Karonis (1988). Their 'synergistic' system, on three levels, maintains the plant control process at the physical level with a series of knowledge bases for different aspects of the process, e.g. the kiln and the pressure system, co-operating via a blackboard. At higher levels the co-ordination system constantly views the state of the process and provides corrective actions to compensate for observed deviations from desired behaviour, whilst the organisation part defines the management knowledge regarding the output and quality of product. These two levels are obtained from hierarchical structures for the heuristics and appropriate deterministic information. The structure allows the separate levels to behave as autonomous units in a functional sense.

Another example of industrial KBS is the modelling of chemical reactions to control a process as described by d'Ambrosio *et al.* (1987). In this scheme, knowledge sources and a blackboard are used to model possible states and then to account for the present believed state. Using time-stamped sensory data, the model relates cause to effect temporally. A similar approach is used by Herrod and Rickel (1986) in controlling a lehr (for annealing glass). In their program, a model of how glass cools (without cracking as it progresses through the oven) is used to drive a planner to achieve adjustments of the oven's settings (i.e. burners, dampers etc.) which should bring the profile of the temperature curve to that for the desired effect. The potential steps in the plans are referred to a simulator of the oven which should allow evaluation of the benefits of the action. Rules used allow definition of contexts of applicability.

The use of complex models, utilising multiple sources of knowledge is exemplified by *REACTOR* (Nelson, 1982). This control program represents a complex system in both exact and inexact terms: it was designed to ease the 'cognitive load' of nuclear power station operators. In abnormal, failure, conditions the program attempts to establish cause (and provide correction). It does this by assuming some deviation from a norm. This norm is provided by a very full model of the PWR. The potential benefit is the avoidance of the typical situation in which the operators are confused by excessive information (on error), as at Three Mile Island. The approach is to detect such deviations from the model and investigate possible reasons via a large knowledge base. Where gaps in knowledge exist, backtracking is used to detect where more information is required. It has two types of description: a) of the physical configuration; b) based on examples from past aberrant situations. The output is a suggested plan to re-establish norms safely. A similar error-detection (safety) scheme for a chemical plant has been developed by Chester *et al.* (1984) and responds to situations where an alarm should be given. This maintains a view of the plant through a series of tasks, written in Fortran, *C* and Franz Lisp. These control the sensors, and implement a supervisory control part and the knowledge base, respectively. The rule-based knowledge can be used to reason in a forward-chaining mode, driven by externally-collected data for example, and form a set of conclusions whose summary provides a basis for decisions. In the backward-chaining mode, the rules, combined with a model of the plant and currently believed values of plant data variables, are used to explain the actual values. This system also records a history of the plant's state to allow expert analysis of behaviour.

A system which reflects more of on-line real-time aspects of industrial control is the plant control system of Evers *et al.* (1984), which tackles problems posed by the requirement for instantaneous analysis and diagnosis. The domain of application is an assembly line treating various aircraft components. The use of an expert system implemented, for example, in an AI language such as Lisp would preclude, it was thought, effective on-line response to data from sensors. The task is to maintain optimum throughput of groups of parts moved by cranes through the plant. Small plans for a sequence of crane operations are the units of scheduling. Due to variations in the duration of process stages (because of crane movements) timing suffers and so the scheduler has to adjust subsequent decisions on the basis of recent performance. (In many ways this is the problem for the paint scheduler in the motor-car example.) The rule base that decides the critical timing relationships is written in LISP. The details of jobs are entered in a real-time system which is part of the control software - responsible also for the robotic element for moving objects within the plant. The paper is aimed at highlighting the interfacing between real-time and expert system symbolisms. The aim is to provide a clean communications mechanism between the different elements of the control system. The expert system itself and the 'lower-level' mechanisms are described elsewhere (Spruell 1981 and Smith 1983 respectively).

2.1.2 Architectures to support real-time IKBS

The standard AI environments include such languages as Lisp (e.g. Common Lisp), Prolog, POP and OPS. Prolog is viewed as being good for declarative representations based on a logical statement of system features and their relationships; see Kowalski (1979) for an explanation of the benefits of this approach when combined with a - separate - mechanism for the control of the expression of the facts.

In the paint shop example, the knowledge is not all declarative and, when it is, the suitability of single-level declarations of rules and facts seems in doubt. Rules about batch sizes are distinct from those which relate to availability of colours. Other means than declarative would probably be required for restricting invocation of many rules at undesirable times (and the rules themselves might become more complicated in order to avoid this.) An extension to Prolog allowing aspects of data to influence rules would help. *LAP* (Iline and Kanoui, 1987) extends Prolog to allow objects to be represented, using a set of primitives, as graphical structures. Winograd (1975) discusses the difficulties associated in representing procedural as opposed to declarative knowledge and offers frames as a means of exploiting the use of each.

Lisp itself offers no structuring other than lists, and although Common Lisp and relatives offer a record-like construct (defstruct) this offers little more than more conventional languages such as *C* and Pascal. One of the apparent requirements for a system comprising knowledge of different types is a convenient means of expressing the different structures of such knowledge. If we take the type of frame structure as elucidated by Winograd (1975) as an archetype, a suggestion that the structuring and representation of knowledge, declared in such frames, could be simple and powerfully expressive, might be supportable. On the face of it, a scheme that allows a complex and aggregated structure to be accommodated, with the possibility of code attachments to assist in the expression of the knowledge contained, offers a good representation for systems with widely differing types of data. What is not so clear is how to control the use of such diffuse knowledge. Although object-centred languages are not always considered in the context of KBSs, it is worth exploring the message-passing paradigm with relation to frames. For this purpose, Smalltalk-80, as defined by Goldberg and Robson (1985) will be taken as a model of object-oriented languages.

The use of a generalisation hierarchy is common to both frames and objects. The common benefit is the succinct attachment of attributes to items which may possess abilities by inheritance. One important difference is that of procedural attachment of code. Dæmons as members of a frame will be activated to establish, perhaps the evaluation of a frame slot. The methods that an object has, though, will not simply be implicitly invoked when a variable of the object's class is referenced. The paradigm of control is quite different. In the case of frames, these attachments require, perhaps, special control paths for activation. A method that provides the same facility for an object as a dæmon does for a frame is invoked deliberately as an explicit message event. This is true for all methods. The control of object expression via message-passing is a crucial feature of object-oriented languages and provides a clear control mechanism for access to objects. The data structuring nature of frames and objects may be motivated by different philosophies, but the aggregation of specific features of data in distinct entities common to both paradigms appears to be, in many ways, similar in effect. Object representations could therefore be of relevance to the type of KBS where knowledge is of different types and a complicated control mechanism for its expression and protection required.

A rule-based system written in Pascal by Wright *et al.* (1986) allows the invocation of Pascal procedures by a KBS. The conventional algorithms which control a satellite may be accessed directly by the conditional expressions in the knowledge base. Another rule-based scheme is provided by OPS implementations, some of which, like LOOPS, an object-enhanced programming language built on Lisp by Bobrow and Stefik (1983), have been developed to allow object representations.

More complex systems, like ART (Williams *et al.* 1985), are specifically intended for multiple representations. This tool provides some special facilities to assist more complex models of problems, e.g. *viewpoints*, which allow for reasoning about alternate solutions. Some of these are special languages/environments, e.g. ART, and others are enhancements to KBS systems, usually written in Lisp. Some of these and special representations for industrial use will be discussed briefly.

The method for monitoring complex real-time processes advocated by Rieger and Stanfill (1980) bases its expressive power on a realistic causal propagation model of the plant's activity. Deviations from the model generate goals for a planner to achieve to bring the state back to the norm. The *state-frames* in PICON of Hawkinson *et al.* (1985) offer a similar power, but represent the status of a plant, acting as semi-independent objects which have interactions, as well as bodies of corrective actions. Moore (1986) gives an account of how PICON represents the 'deep knowledge' of a plant's physical structure.

A more general scheme for a KBS in chemical plant monitoring or control than those of Chester *et al.* (1984) or d'Ambrosio *et al.* (1987) is the multiple KBS of Avouris *et al.* (1988). This is concerned with the complete automation of a response to a chemical accident. It includes a knowledge-base for many diverse types of information relating to models of chemical reactions, potential decontamination procedures, who to involve in the emergency measures, etc. as well as stochastic models of the likely effects of weather conditions. To link these 'knowledge sources', a blackboard-like structure together with a handler is added to control updating and focus-of-attention.

LeClair (1986) suggests a scheme for integrating several expert systems that features a central blackboard residing in an expert system on one processor, where direct access in real-time is allowed to the data by other expert systems via memory-mapping. The area of application is one of 'sensor fusion'; specifically in the control of heat-produced composite materials, such as graphite epoxy laminates. In this technique the sensors collecting data on the state of the process and the control interface are interfaced by the blackboard (and communications) processor. This features an interesting system for interpreting the real-time data. Its sensory 'parser' builds an interpretation of the semantic relevance of the input data stream by examination of a lexicon which records knowledge of the application of the given domain. The 'natural language' of the parser is the 'words' in the lexicon. By consulting the task knowledge of the domain as well, the significance of the data is inferred. Where ambiguity still exists, the stream is scanned and the following input used to clarify the meaning. Failures are then taken as indicating a deficiency in the parser's knowledge. Another expert has knowledge of user plans, which are reactive to the state of the blackboard and give control tasks. The concurrency of the system allows this analyser to view (via the blackboard) the integrations of data and their meaning produced by the parser and it can initiate goal-driven reasoning to develop command sequences or expectations of future events, using temporal and qualitative means. By considering the past sequence of data patterns and domain knowledge the detection of faults is possible using the expectations derived from that knowledge. With this 'kernel' and specific knowledge for a given domain, a control application can be built. The design of the expert co-processors is that developed by Park (1986) for *EXPERT-5*.

A method for reasoning about the state of processes is advanced by Kämmerer and Allard (1987). It allows on-line scanning of sensory data, in that data are sampled at regular intervals while inferencing is suspended. It then resumes after the values are obtained. The state of the system and reasoning is held in a network of augmented transition nodes. A table of permitted state transitions is then used to transform the network as evidence accrues, i.e. from the collected data and prescriptive knowledge of the process. Nodes may be generated, for example, to explore an aspect in greater detail and may then lend support to the parent or resolve the issue raised. Two mechanisms which deal with possible non-monotonicity of the model state (arising from perceived changes in the process) are used. The values of conclusions that may change unexpectedly over time are allowed to be represented as defeasible (i.e. if a change has occurred, the reasoning path may be re-traversed to account for the now incorrect item and the value of knowledge, which is generally less reliable as it ages, may be reduced as the state is analysed). To handle the latter problem time-dependent facts are allowed to have their supporting evidence reviewed. As well as this, the validity of propositions may be unknown in addition to true/false. The output of the network is a set of conclusions about what the state is and what to do about it.

2.2 AI issues for paint scheduling

Perhaps the central issue of using a KBS for the paint scheduling problem is the choice of a representation scheme. Beyond that, the acquisition of knowledge and control of its expression are important aspects for consideration.

Sloman (1985) reviews many of the problems associated with use of AI in real-time systems. He distinguishes two separate and necessary abilities for such systems: perception of events and response to them. He recognises this as a problem of 'concurrent monitoring'. To these he adds a third activity, goal evaluation. Difficulties with the real-time nature of data may lead to conflicts over continuing evaluation/action in the face of new/deleted data versus the importance of complete consistency.

In the rest of this chapter, the discussion is concerned with examples of how different workers have approached these areas. It follows from the assumptions that the paint scheduler must be able to represent different views of perhaps conflicting constraints and that it must be able to be extended and be understandable in its behaviour. A discussion of the basis for these presumptions and what AI design decisions were made in view of current expertise is left until Chapter 3, where further comment is made on the requirements of the problem examined in Chapter 1.

2.2.1 Representation schemes

XPL, a frame-based language (Barbuceanu *et al.* 1987) attempts to make a program description the program itself, incorporating semantic concepts in a declarative structure. This structure would allow the attachment of specific abilities to an item, a semantic *construct*, enabling it to be used in, and therefore sensitive to, different contexts, i.e. states of the data. Briot and Cointe (1987) discuss object-oriented languages as being useful for knowledge representation and suggest that Smalltalk-80 is not consistent enough in its treatment of objects; their ObjVlisp extends the object concept further.

A different treatment of relationships between separately identified 'discrete' entities in programs was put forward by Hewitt (1979). His *ACTOR* paradigm, where data and code are associated into blocks which have a co-ordinated response to messages passing in a system, might offer a method for universal control in a real-time environment. This scheme is the basis for *YAMS*, of van dyke Parunak *et al.* (1985), which models a factory as a hierarchy of actors and optimises message traffic between them. There are three groups of objects: pallets, moving between places as work is done on them; nodes, which represent work stations which transform the contents of pallets; and links, which are means for moving pallets between nodes. These form graphs representing the factory. At each level of abstraction, e.g. a pallet, the item may be an aggregate of the same type. The significance of the actor formalism is that for each actor there is a standardising interface governing information exchange in the system. This extends to the hardware level of machines which control processes (their functional code being logically represented as the terminal points of the graphical network.)

A distributed method of organisation is offered by the more developed forms of blackboard system, e.g. the generic blackboard, GBB, of Corkhill *et al.* (1986). In this, the blackboard itself has multiple levels for knowledge sources relating to different types of rules and allows filtering of the blackboard's patterns for matching to rule conditions. One potentially useful aspect for real-time processing is the explicit representation of space and time in the blackboard, so that values for items can be related to their existence at particular times. Velthuijsen *et al.* (1987) implemented a blackboard scheme for real-time robot control. Based on the approach of Erman *et al.* (1981) in *Hearsay-III*, they extended the system to make the blackboard-handler sensitive to real-time requests for input/output. At each stage of picking a knowledge source to solve part of the current problem, those 'triggered' and ready request activation. The list of requests is compared, after real-time events may have changed the state of (a) blackboard(s), to find any suitable knowledge source to activate, i.e. reject any that are now invalidated by blackboard changes. They note problems of data consistency where one knowledge source may compete with another to change/access blackboard elements and suggest common data-locking techniques. An interesting feature is that knowledge source actions may cause direct execution of code written in 'alien' languages, e.g. POP-11, to achieve events in the external world. There is no discussion of how to deal with interrupts which may arise during execution of a knowledge source and which, if responded to, might invalidate it.

The *EXPERT-5* design (Park 1986), is a practical illustration of how to achieve a real-time KBS facility with asynchronous updating of external data (i.e. with respect to reasoning about them). As mentioned above for LeClair (1986), the on-line collection of data and incorporation in organised structures (on a blackboard) is handled by an expert data collector (and communication system). This design allows other processors to access the blackboard directly. The blackboard forms the working memory of a 'production system'. It uses message-passing between objects to mediate control. These objects represent knowledge about domain items. Rules are represented by blocks of FORTH code, activated as methods. This would seem to imply that the control of the expression of rules is open to inspection, e.g. a truth message would cause rules to evaluate themselves, but that the knowledge contained in them is not (i.e. is hidden in code). Park states that a three co-processor system, with each processor running *EXPERT-5*, where the central one is the blackboard plus communications channel, is capable of significantly increasing the speed of execution of a (single-processor) real-time KBS: the tasks of accessing the environment and responding to it are separated. He suggests that another significant advantage of the design is the ability to group rules into suitably complex structures.

The organisation of knowledge into frames having production rules as part of their expressive power is favoured by Aikins (1979) in *CENTAUR*, an implementation of the diagnostic system, *PUFF* for pulmonary diseases (Kunz *et al.*,1978). She uses the concept of 'prototypes', which associate all the significant aspects of a condition in a structure, so that if there is a tentative possibility of considering a disease as a hypothesis, the prototype for that disease, with all its aggregated knowledge, is put on the list of hypotheses. Once consulted, the prototype has its rules for establishing itself activated, and all the relevant information for that condition is collected, if necessary, and the diagnosis can be established. It also may suggest access to related conditions via links of association. The control knowledge is represented in each prototype as distinct from other knowledge, so what to do in a context can be stated explicitly (see Aikins, 1983).

The knowledge areas (KAs) and recursive transition networks (RTNs) of Georgeff and Bonollo (1983) provide, respectively, a contextual mechanism for rules which are still declaratively expressed, and a method for attaching explicitly procedural knowledge. Invocation parts of KAs allow testing of known facts and current goals to establish their validity for use. Their 'bodies' are special inference procedures to process a sequence of subgoals or facts to be established. RTNs form a state graph whose arcs are tests which, if true, are traversed with the associated actions being done. The authors regard the KA structure as meta-level rules which constrain the expression of knowledge. This may be a development of ideas put forward earlier by Georgeff (1979) where he views a production system as a set of triples (production name, database and interpretations) which are influenced by a control language - i.e. providing a framework for control. For each set of triples there is then a quadruple where a control language governs them. A similar proposition of an appropriate way to handle procedural knowledge explicitly is put forward by Gallanti *et al.* (1985), who are concerned with the control of water-borne pollution occurring in a power generator. Monitoring of the turbine and other components will reveal a state of damage. It is difficult to attribute blame to the potential sources of pollution. Rule-based inference creates hypotheses of what has happened but modes for investigation of the system are known in procedural terms. Using a modified Petri net scheme (of 'event graphs') the steps required for gathering of evidence are represented as directed graphs. The events are nodes and the transition arcs show the conditions and actions that are attached as an arc is traversed. So, depending on the observed values from data, a path of enquiry is built up by traversing the network. Given a set of observations, the rules may explain the cause of the damage. The nets are used in two ways: firstly, for the interpretation of data (at the level of sensors and chemical knowledge); and secondly, for diagnosis of faults and intervention, i.e. to show how to detect and deal with them. The second network is an abstraction of the first. Two databases are used to record the actions of the nets. The second net looks at the results of the first and posts its results in the second database, from where the necessary data for the rules is obtained.

A different model of control in systems is advanced by Francis and Leitch (1985). They advocate the description of a system as a set of sub-systems which have only one input and one output and each influences the next. The interactions of each subsystem on its neighbour are modelled as direct causal effects. They point out that the rules for this could be obtained from observation of the plant's process, which would be likely to be captured accurately and quickly from the known causal structure of the plant, and could give accurate explanations based on the causal relationships between the subsystems.

A planning scheme for organising manufacturing cells in a factory is described by Shaw and Whinston (1985a, 1985b). This has two levels of representation. The communications between cells are represented by procedural descriptions. Within individual cells, knowledge is described in three ways: a) declaratively, with goals, the state of the domain and semi-complete plans; b) about actions in the world, as production rules or operators; c) of strategies for planning, i.e. how to select operators. They require incorporation of non-linear planners. Strategies such as *critics* in NOAH, the hierarchical planner of Sacerdoti (1975), are suggested for dealing with the conflicts arising from constraint satisfaction. A table of multiple effects would presumably be held to identify which preferences were used and the best choices of machines assigned to tasks, followed by elimination of incompatible operators. It is not stated how individual critics capable of refining different problems would be defined. This hybrid scheme uses Petri nets at the inter-cell level to capture the control aspects, which are augmented by rules fired as arcs are traversed. The events described declaratively have their temporal interactions and relationships modelled in the nets. They describe a notion of *task-bidding,* where a model of distributed inter-cell control is mediated by one cell broadcasting that a job is available and arbitrating between the competing cells offering to do the work.

A different model of control is suggested by James and Frederick (1985). To cope with the complexity of control engineering, use is made of partitioned rule bases for different areas of interest, under the control of a supervisory control rule base, which governs the invocation of particular rule groups. Though written in Franz Lisp with General Electric's *DELPHI* inference engine, the system allows direct calling of Fortran subroutines to supply data from sensors to allow the expert system to evaluate rules.

The multi-agent planning of Georgeff (1983) might suggest a means of handling complex interactions in a process control problem. The conjunction of single-agent plans without destructive interactions could be useful for the type of uses indicated by Shaw and Whinston (1985a, 1985b). They seem to assume that distributing work through a manufacturing system allows separation of machine-level and plant-level planning. The critics suggested would not be able to deal with interactions at different levels of abstraction. The technique Georgeff proposes is to look for unsafe interactions when one agent's plan is compared with another. If some deadlock arises then synchronisation primitives are employed to circumvent it. This requires the STRIPS assumption (see Fikes and Nilsson, 1971), that the effect of an action does not undo pre-conditions unless stated otherwise. In his later work (Georgeff, 1986), he avoids this difficulty by a mechanism allowing some events to be logically conjoined, i.e. occur simultaneously, while others may vary independently. In this, providing that two actions, for example, do not change the same world relation, they are allowed to be linked.

A related problem is that of constraint propagation in real-time domains. Collinot and le Pape (1987) tackle the use of time in directing constraints. They advocate dynamic adjustment of the amount of processing done in applying constraint-directed reasoning. They use this to produce job-shop schedules, allowing certain constraints, e.g. due dates, to be moved. The control rules which apply ordering over events may be altered.

Perhaps the most interesting approach in relation to this work, in providing a suitable environment, is that of Odette and Dress (1986) on a FORTH implementation of Prolog and OPS-5. This environment was intended for use in real-time control systems where code size and speed was of paramount importance. Their view was that special purpose hardware would not be available and the facilities of the AI languages were providable. They wrote compilers for these two languages which were compact and, at the same time, allowed the direct invocation of underlying FORTH routines to change the external world state, e.g. by FORTH words in OPS-5 actions or primitives in Prolog.

2.2.2 Knowledge elicitation and extensibility of KBS

The practical aspect of this area of AI for this project is the expansion of a system by acquiring new principles in a consistent manner. The idea of building an empty system up incrementally ("boot-strapping") would seem to be attractive but in the case studied a large part of the system is basic and needs to be present for the comprehension of new principles to be possible.

Davis (1977, 1979) produced *TEIRESIAS* as part of the *MYCIN* project (Shortliffe *et al.,* 1975) at a time when the size of the rule base was becoming hard to cope with. It was an attempt to provide a means of expanding the repertoire of diagnostic/therapeutic production rules for infections at a time when the *MYCIN* team experienced great difficulty in adding new rules to the already substantial knowledge base. *TEIRESIAS* is a knowledge acquisition assistant: an interactive tool with interrogatory capabilities.

In its acquisition mode it has several activities:

a) it attempts to classify the new information on the basis of classification hierarchies (schemata) that are known to the system. In case of a new object being recognised it places it in an appropriate category or creates a new category for it according to the known categorical hierarchy. The meta-knowledge scheme of Davis (1980a, 1980b), where knowledge of the nature of the knowledge contained is held explicitly, allows reasoning about what the new information may imply.

b) *TEIRESIAS* displayed another feature central to 'learning': new information once "acquired" is systematically examined for consistency with domain facts and such facts are used to refine new information if necessary to assimilate it. New rules are first identified with respect to a) and then fitted in with respect to b).

c) a related function to b) is the integration of information, e.g. a rule, via the principles above followed by possible reviewing of already accepted rules. A new rule can be slotted in once it is in a consistent form. Alternatively, if the user declares it to be satisfactory, it can be fitted in with other rules, if they are modified.

TEIRESIAS checks the consequences of a new rule thereby. These facets - classification and consistency etc. are crucial to a useful 'learner'.

Classification as a technique is not used by Davis alone, but *TEIRESIAS* seems to require less cueing than could be imagined. In his work on more general classification, Clancey (1985) urges the use of categorisation to order objects, relationships etc. and represent ways of relating independent hierarchies to each other explicitly, i.e. by heuristics, to state when and how one item may be opportunistically "mapped" onto a different group to provide more powerful reasoning. (This goes rather beyond the extent of classification used in *TEIRESIAS*.) The re-formulable approach of *TIMM* (Cooper, 1984) would seem to be irrelevant to this problem (having a fixed and static representation scheme) although apparently able to expand a knowledge base consistently.

Workers such as Michalski and contributors have apparently produced some successes in 'learning'. Michalski and Chilausky (1980) use the AQ11 algorithm to correlate disease descriptions with symptoms and environmental conditions. They then induce heuristic rules to provide diagnostic ability in a program. The *meta-DENDRAL* system of Buchanan *et al.* (1976) uses structural knowledge of chemical compounds to produce rules explaining the observed degradations of chemical compounds in mass spectrography used by the *DENDRAL* system (Buchanan and Feigenbaum, 1978).

By the use of automated 'discovery' in *AM*, Lenat (1979) tried to show that a program examining its own knowledge could link parts of it together in new ways. In this program, a base of 'concepts' of mathematics and a set of heuristics, with an added notice-taking feature could generate potential regularities and test them for relevance. They would then expand the knowledge. In his later work on *EURISKO*, Lenat (1983) used a similar method of randomly making small mutations in the values held in the slots of the (frame-based) generalisation hierarchy for the knowledge. The heuristics were distributed over the frames and their slots so by altering small portions of the hierarchy new heuristics could be created, which could then be tested for beneficial improvements. The mutative mechanism was therefore useful for acquiring new rules. The nature of the concepts and rules was described later as being intrinsic to Lisp - as an expression of the λ–calculus. Lenat and Brown (1984) admit that the 'discoveries' that *AM* made were not surprising because Lisp allows interpretation and examination of its own code. The ability to change heuristics slightly and examine the results in terms of effectiveness is claimed as a significant step by these authors, and could, indeed, be an important technique. One difficulty with evaluating its worth is, however, the recognition of a valuable new rule and the selection of appropriate data for the generative steps. Difficulties in the selection of training examples for learning programs are noted by Bundy *et al.* (1985). An interesting point raised by Lenat (1983) in his review is the speculation that natural evolution may progress by such methods. This has been the prevailing view since Darwin. A major point of difference is that a powerful external selective force exists in the case of evolution but would have to supplied by the author of any program. Without prior knowledge of the likely area and form of useful heuristics the program would be likely to produce them at a similarly slow rate to nature.

A different view of 'learning' is considered by Mitchell (1977). In the *candidate elimination algorithm*, he proposes the partition of the rules, given a set of training instances, into two sets: those for which the instance is positive, i.e. the rule is applicable, and those for which it is negative. The rules applying to these categories are regarded as a set. A distinction is made when one rule will apply to only a subset of the other, i.e. it is more specific. As rules are found to conflict with additional training instances they are removed. By making their conditions 'minimally' more/less specific as more instances are tried, the rules can be made to account for all the examples. Eventually the set of rules is consistent. A fuller account of how this could be used is given in the work on the *LEX* system (Mitchell *et al.*, 1981), an improved version of this technique. *LEX* acquires rules to control the selection of operators in symbolic integration. Initially there are no heuristics and *LEX* will first find out how to use the terminal operators applying for cases where there are no unsolved integrals. This is followed by an examination of the steps used to do this and the generation of training instances for operator-applying heuristics. The third step is to apply the instances to the heuristics for their refinement. Mitchell *et al.* suggest the use of a problem generator to look at the current state of knowledge and output more complex problems to obtain new heuristics. They do not state how this is to be created.

A study of learning techniques by Bundy *et al.* (1985) notes the twofold problem of the selection of training instances and knowledge of correct behaviour that the newly learned principles should observe. These functions may be obvious to the human observer but are not easy to automate. They also suggest that techniques such as candidate elimination may be very sensitive to the order of presentation of training examples.

A study of how best to acquire knowledge from domain experts was carried out by Sweickert *et al.* (1987). A formal interview provided the most reliable account of rules. They also examined the use of problem situations to see what data the expert would request, thereby inferring rules from the context of the data. A third technique was to ask experts to rank a set of outcomes with associated (initial) conditions with respect to their degree of relevance in a set of situations. Of these three methods, the third was significantly inferior. The first was the best, but is not a possible method for automation as is the second. The poor performance of the third technique tends to indicate that automated knowledge acquisition by some variety of rule induction might lead to poor results. (The use of this approach might be where test data from, say, an industrial process is compared with the results of expert control and a rule generator used to induce causal rules connecting the two - i.e. rules from observation. If human experts find it difficult to elucidate rules, because they take into account extra factors when adducing principles, it seems that in many cases this approach will not work in an automated system.)

TEIRISIAS, AM and *TIMM* are systems to extend the knowledge they represent. *TIMM* and *TEIRESIAS* are particularly concerned with maintaining consistency. In these, the knowledge gained is fixed and new knowledge must co-exist with the original. A different representation of the knowledge is attempted by Mark (1977). In this method the expert's description of the domain is transformed into a new form, re-formulating the problems to be solved. Instead of building abstractions of expert knowledge to be manipulated, the program makes a representation of the domain and maintains its interrelationships. Expert knowledge is then communicated to this model which forms the expert's view of the system and a transformation is applied to it to produce a working KBS that solves problems related to the real world. (The method in *TIMM* is to produce rules as it digests the description of the domain problem and then try to make the knowledge consistent - i.e. incremental consistency.)

An important part of the usefulness of an IKBS is to be able to account for the decisions it makes. One of the first expert systems, *MYCIN,* of Shortliffe *et al.* (1975), featured an ability to show what path the reasoning had taken. Its 'WHY' option invoked the tracing of the rules used backwards towards the original goal, i.e to any stage that the user requested. The explanation was given by an English translation of the rules themselves and allowed the user to query why a particular aspect was considered ('HOW') to get more detail of the strategy involved. It also could be used to examine the rule base itself. These methods formed the basis of explanatory facilities in developments of the *MYCIN* system.

NEOMYCIN attempted to extricate the domain knowledge of bacterial infections from the control aspects of expression of the knowledge (i.e. Davis' metalevel knowledge). Clancey and Letsinger (1981) produced a separate and domain-independent strategy scheme to do this which referred to the structure of the problem space itself, that is, how and where causal knowledge about data and hypotheses and world facts should be applied. Their 'psychological' model of reasoning means that it is possible to adduce a rule to show why a particular line of reasoning has been followed. In the context of medical diagnosis it was often difficult to understand *MYCIN*'s line of pursuit of information etc. Clancey (1983) considers the ways in which *MYCIN* can only re-iterate domain level knowledge and is ignorant of the medical strategy on which it is based. Hasling *et al.* (1984) explain how the strategic knowledge of problem solving is represented. A particular action to investigate a line of reasoning is a task made up of several steps which can be explained as such to the user. They also consider the problem of offering the appropriate level of explanation to the user. In *GUIDON*, Clancey (1979) improved *MYCIN*'s facilities to offer tutoring ability. The knowledge was re-organised so that rules were abstracted into patterns of related context. Rules were now adduced with reasons for their support and justifications. A record of past consultations with the student was kept to suggest the direction of dialogue. One way of elucidating the path of reasoning was adding canned text, derived from an expert, to a pattern of rules, which could then be presented as a justification for their application.

The *SOPHIE* system of Brown *et al.* (1982) was originally intended to be a trouble-shooting program for electronic faults, using physical laws and modelling constraint propagation to provide causal reasoning. It proved to be more useful for teaching students how to diagnose problems in circuits by demonstrating the causal mechanisms of its reasoning. It was also able to guess at the level of ability of the student from the pattern of guesses of loci of faults, using a model of how learning progresses. An interesting feature was the suggestion of useful areas of enquiry if the student was unable to discern the nature of the problem.

BLAH (Weiner, 1980) supplies three methods of querying to furnish explanations. A user can ask for an assertion to be substantiated given the state of the system's data base. Using the 'CHOICE' option, the system will accept two assertions and select, with justifications, the better of the two. In its third mode, 'EXPLAIN', the system is more sophisticated. It models the user and itself as having views of knowledge and deletes any information from an explanation path to which the user already has access, i.e. in the user's view. It knows, for example, that if an assertion is supported by one fact known to the user and by another which is unknown, it should find an explanation of the other justification. Weiner gives examples of these features but does not state the mechanism of partitioning into user and system views.

Swartout (1981, 1983) departs from the rule trace mechanism, and canned text is not used in *XPLAIN*. System function is represented as a tree of goals, starting from the highest of prescribing appropriate Digitalis therapy. Its intention is to explain why the system does what it does. From a description of the domain from an expert, an automatic program writer generates the working structure. There are two generators for explanations, one which collects words form the knowledge base output by the writer to make phrases, and another which is concerned with determining what to say. In doing this, it is guided by the state of program execution - seeking not to repeat issues that have been mentioned already - and by consulting a list of aspects that are relevant to the consultation. (These are in turn derived by association with domain principles originally enunciated to the writer.) The development state of the system is recorded (indeed the manner of functioning of the writer is by successive refinements). Neches *et al.* (1985) follow the method of *XPLAIN* and use a causal model of the domain which records the nature of the knowledge rather than how it should be applied in problem-solving. The range of questions supported extends to justifications of behaviour or what a result signifies, why some aspect was or was not considered at a particular stage, what definitions or functions mean and the extent of knowledge on particular aspects.

Rubinoff (1985) defines multiple categories of rules, through a front-end to a rule-based system, and, according to the category, attaches various types of explanatory potential to the rule when used in reasoning. In addition, any concepts in the rule are attributed importance in generating explanations according to their position. Another feature is the ability to define a rule that generalises a group of others, so that it can be used to explain them. The *CLEAR* system is in the tradition of *NEOMYCIN*, because it uses a translation of rules to generate explanations.

A common feature of the attempt to reproduce an expert's knowledge which is adduced to assist understanding of the knowledge is a modification of propositional validity by some attachment of an uncertainty. In general, these are based on a statement that 'if some premise is matched then assert a conclusion with some degree of belief'. This is often based on some prior assumption that, statistically, in a proportion of all cases where the pattern is found, the conclusion is valid. Alternatively, a degree of membership to some category is applicable given the pattern. This is the use of fuzzy sets (Zadeh 1979).

Perhaps the first use of uncertainty in KBSs was the certainty factors in *MYCIN* (Shortliffe and Buchanan, 1984). This has been imitated in many expert systems. It adds to a simple premise/conclusion rule an attachment of degree of belief. This may be evidence against, as well as for, a conclusion. By propagating, in a manner apparently influenced by Bayes' theorem of prior probability, along a causal chain of inference, a level of support for an eventual conclusion is arrived at, based on positive and negative evidence. (It is noted that Shortliffe developed the mechanism for propagation of evidence himself. The use of evidence in this way, both for and against a hypothesis, was new to KBSs.) In practice, weeding out of poor evidence is done by ignoring combined probabilities of less than a certain threshold. Where a conjunction of evidence is encountered the least value of certain is propagated. Conversely, disjunctions take the local maximum. It is questionable whether treating low values of derived confidence as true is safe; similarly, the combination of several pieces of weak evidence, taking the local maximum of them, may result in the neglecting of relevant facts, where they may indeed combine to something stronger. Cohen (1983) takes this as a failing of the scheme. Barclay-Adams (1984) tries to show that Shortliffe's certainty factors are equivalent to Bayes' Theorem, but notes that the evidence for and against a hypothesis should be equivalent to a level of absolute certainty, e.g 1, but is often not. In response to their perceived shortcomings in the certainty factor approach, an alternative is suggested by Gordon and Shortliffe (1984), this time using the Dempster-Shafer method of specific apportionment of belief to discrete partial elements in a hypothesis. The 'frame of discernment' allows that the evidence be explained by a set of exclusive sets and the proportion of belief be assigned explicitly to each.

Cohen (1983) attacks all these numerical methods of belief assignment as poorly based on realistic statistical measures - at least in the domain of medicine - mainly on the point that prior probabilities are practically impossible to be sure of (a requirement of the Bayes' approach, but not of the Dempster-Shafer theory) or are generally not known. He believes that the values assigned to certainties reflect more on the exigencies of making the system give the desired response rather than true statistical measures. He prefers to define two levels of belief: firstly relating to the nature of the evidence, and, secondly, to the qualitative strength of the evidence. Thus, the relative worth of evidence is described as well as where it comes from. This acknowledges that the apportionment of exact values to the origin of evidence is a relevant but impracticable requirement of the Demspter-Shafer theory. A significant improvement could be gained by using the 'endorsements' on both sides of rules. In this way an element of uncertainty is collected in applying a rule as well as making conclusions from it. He suggests that the accumulation of endorsements as rules are chained might cause problems in terms of space but that some heuristics might be applicable to prune the older endorsements as the line of reasoning lengthens. He suggests the categorisation of endorsements so that control rules could be made differentially sensitive to the contexts accessed in the rule processing. The main interest of his approach is to demonstrate by adequate, not quasi-statistical, attachments the nature, origin and reliability of the evidence and reasoning. Cohen and Greenberg (1983) suggest how this technique could be made to drive goal-directed reasoning to determine fruitless lines of enquiry. Sullivan and Cohen (1985) use endorsements to reason about a user's actions in order to interpret the likely intention.

Chapter 3 Issues addressed by the work

The object of this research is to find ways of handling the complex statements of constraints applying in the paint shop. Some knowledge is declarative, some best described by procedural representations. It is desirable that as little knowledge as possible is hidden in large blocks of code, where it is hard to explain its significance. Another aspect that presents difficulty is the possibility that new rules will be stated at various times and these are likely to have differing contexts of applicability. This suggests the need for multiple representation schemes. These types of problems are thought to be common to many different domains in manufacturing as a whole.

There are several aspects to forming a solution for the paint scheduling problem. The major problem is to find a representation scheme that will allow both declarative and procedural aspects of the knowledge to be captured. In order to be useful the reasoning path must then be explainable. A system that can allow the specification and use of new types of knowledge, in this case constraints, would be an improvement over the algorithm adopted in the factory at present. It would be more generally useful as well for systems where knowledge of improved operating practices were to be acquired over time. To aid acquisition of new principles the scheme used should provide some facilities to assist users to express new rules and concepts and alter existing ones.

3.1 Why not use a mathematical programming solution?

Finding the best available solution for the paint problem is formally equivalent to the travelling salesman problem. If the objective is to change paint colour in order to minimise the costs associated with deviation of the production profile with respect to the order cover, then the example given by Gilmore and Lawler (1985) should be a solution. The obvious difficulty is to prescribe cost functions associated with maintaining the colour for the next body and changing it. These would presumably be obtained from lengthy studies of the cost contributions in the factory from the interactions of the known constraints. The cost functions would need to be a set of linear constraint terms. It is indeterminate whether the costs in this case have linear interactions. One important difference between the well-known applications of linear programming, for example, and this case is that the program, at any instant, has only a view of a small amount of the total input that is significant. The shift order cover is known at all times and so is the production profile to date. If the whole sequence were known in advance then a good optimisation could be achieved, ignoring the chance of a random paint failure. There can be no guarantee that statistically-based projections of what the future sequence might be, given the current input, would allow a suitable level of optimisation by a method based on numerical calculations. This is because the use of probability considerably complicates a linear expression of the model. Also, there is a possibility that recognition, on a statistical basis, of what type of response should follow a certain input, will fail if the matching is close but inappropriate. Linearisation presents additional difficulties when constraints have multiplicative effects on each other. This makes a model inherently non-linear.

3.2 A representation scheme for paint scheduling

The main problem in handling the available knowledge is that the expression of 'constraints' depends on context. Thus, if used as rules, a mechanism would be needed to ensure their applicability as such rules. Some partitioning of the rule set is mandatory in such cases. A popular approach in the literature is to use a blackboard scheme with rules of the same 'interest' grouped together, where their expertise can be brought to bear on a given type of problem. Activation of a knowledge source could be used to cause expression of certain rules, for example those determining batch size. A global blackboard seems undesirable because a control scheme for blackboard and knowledge source(s) would require a complicated means of control of rule interpretation. This control would probably have to be resident in routines constituting a blackboard-handler or attached to knowledge sources themselves. A number of differing rule sets and contexts for expression in knowledge sources might increase the amount of procedural code that is obscured from sight. A motivation in this work is the wish to make as much of the processing as possible visible to examination. Thus, blackboard-type schemes following Erman *et al.* (1981) did not seem easily applicable, nor did the rule bases annotated with procedural knowledge, e.g. the approach of Georgeff and Bonollo (1983) or Gallanti *et al.* (1985). The logical association of concepts advocated by Aikins (1979) does seem a better framework for organisation of data and code, but a more structured and tightly controlled design seemed more practicable.

The use of a frame structure was considered. This would have allowed activation mechanisms to be attached to rules (and groups of rules, by inheritance). One disadvantage is that the nature of the slots of the frames would be open to examination by other frames, and for reasons of control, discussed later, this is not wanted. The characteristic structural association of context and rule suggested that object-centred representations would be helpful. The declarative means of grouping associated aspects of items was considered a significant benefit of object-centred schemes. The making of classes sensitive to particular messages, by definition, suggested a means of arranging for rules to be activated as groups.

With these considerations in mind, the OPS environment seems attractive, as does Smalltalk. A problem of using these would be that they could not function in the operating situation pertaining to most industrial applications. I am not suggesting that they would be impossible for implementing the representations that are required. In fact, the facilities for rule expression and the ability to define objects in OPS-5 would provide a good environment; similarly the structuring power and expressiveness of Smalltalk-80 could be used to good effect. The implementation of the representation scheme discussed in Chapter 6 was influenced by the philosophy, if not the coding methods, of practical object-centred language extensions suggested by Cox (1986).

The encapsulation of objects to be used is seen as a way of allowing extensions of the system to reflect additions to knowledge as they are made. This is rather like the declaration of new predicates as pieces of code in many AI languages, when combined with the creation of new classes to reflect new contexts. The structure detailed in Chapter 6 shows how these aspects are catered for. Rules could be instances of classes which are sensitive to interpretation by responding to messages asking them to interpret themselves. This allows the definition of many different interpreters according to the type of rule required. The apparent disadvantage in this is offset by the considerations that the interpreters themselves do not have to be anything but small methods, which could probably all use the same basic syntax, and that most rule classes only use one type of, inheritable, interpretation. The control aspects of rule expression, in this formalism, are abstracted to the variables of the rule classes themselves, so they are observable.

Most production systems use a single, complex, interpreter which hides a lot of control knowledge internally. The 'working memory' of my system will be the class structure itself. A slight disadvantage is that the database-handling mechanisms are not centralised, but are methods of the classes to which they belong. The control mechanism strongly adheres to the object-centred paradigm. All access to instances is via their own modes of control, using messages. This has the benefit of the flow of control being immediately traceable, both in rule expression and in other traffic. Rules themselves are mediated by evoking more message traffic. The conditional and action parts are expressions which form messages on evaluation. In this way a single representation scheme allows expression of rules, objects and control. If desired, the access mechanisms to all objects for enquiring and setting values could be attached to the super-object of the hierarchy, by two special methods which access the object definition. By inheritance, then, all memory references are unified as if the class structure were a more conventional working memory. Rule classes do not, then, respond to updates, and the production memory is logically distinguished from the working memory.

A benefit of this scheme is that procedural knowledge is clearly stated in methods attached to classes. The complexity of most methods is likely to be small and the declarative power of rules using them is greater because the methods express their origin which is likely to be a readily understood context, already defined. Control knowledge, e.g. of rule expression, is not necessarily put away in some hidden procedure. In this scheme, the control rules can be expressed as such, as and where appropriate.

AI environments available seemed to provide some of the important features but would not have been credible as delivery systems for a working solution. The necessary supporting software, a simulator and test-bed, would have been difficult to write using them so *C* was used. It seemed feasible to provide the special object environment also in *C*, so the KBS will be developed in that environment, with *UNIX*, as a part of the simulation. This also avoids some of the inherent difficulties in using mixed-language representations.

The literature provided many indications of what might prove useful for this type of problem. The techniques resulting reflect the distillation of a suitable environment from many parts.

3.3 Acquisition techniques

The use of 'context-sensitive' rules leads to the use of a knowledge definition scheme which will be discussed in Chapter 6. Davis' approach in *TEIRESIAS* offered a way to use the structure of the knowledge, implicit in the class definitions, to guide acceptance of new rules. The development of a 'meta-definition system' to do this seems close in spirit to his ideas of metalevel knowledge. It reflects my view that the nature of the knowledge in this kind of application is hierarchical and can be made available to structural inspection, perhaps in the way that Clancey (1985) sees some knowledge as 'epistemological'.

In terms of extending a KBS, the object-centred approach seems useful in maintaining consistency. Classes, when correctly defined, encapsulate a model of some aspect of the world domain that is a whole and has clear relationships with the other models. If new information about the domain is added, which does not contradict the existing model, it should be possible to add it by adding a new class to represent an extra model. In this sense, the paradigm is of a set of internally consistent models whose interactions cannot be affected by the addition of a new, internally consistent, model. Practically, rules reflecting a new set of principles can be adduced by adding a new class. This will be discussed more fully in Chapter 6.

3.4 Practical and analytical work

An analysis of the nature of Istel's algorithm for colour allocation is made in Chapter 4. The significance of the principles ('rules') of scheduling is assessed and is to be read as an introduction to Chapter 5, where the performance of the algorithm in tests is recorded, with an evaluation of such performance in terms of satisfying the criteria of flexibility, ideality of production etc. (as noted in Section 1.7)

The practical work done on the paint scheduling problem consists of four sets of related programs forming the necessary utilities for a knowledge-based approach. These are:

i. a simulator;

ii. a statistical analyser for paint performance;

iii. an object-centred programming toolkit; and

iv. an editor for meta-descriptions of object-centred systems.

The former two have been developed to test the performance of the knowledge-based solution with respect to the 'conventional' procedural approach and are described in Chapter 5. The latter two form the basis of an object-programming environment which has been used to develop a program to do paint scheduling in a knowledge-based manner: they are described in Chapter 6.

Chapter 7 discusses the objective approach to scheduling as described in Chapter 6 and evaluates this approach in terms of Istel's algorithm. Chapter 8 discusses the work as a whole, with relation to flexible manufacturing as well as providing a useful approach to reactive scheduling.

Chapter 4 An assessment of the scheduling algorithm

4.1 An examination of the colour allocation algorithm and 'optimisation'

Istel's algorithm for colour allocation could be summarised as 'pick the *least achieved* colour with maximum permitted batch size'. In this sense, least achieved means that colour which is, proportionately, the one with the lowest ratio of BIWs painted to orders outstanding, in the current schedule.

Orders for the various permitted combinations of body and colour are viewed in two ways: the *achievement* counts the number of BIWs successfully painted and the *requirement* shows how many BIWs are still to be painted. The intention is to keep production in the different colours balanced, so that one colour does not dominate over the rest. In practice, the ratio of achievement to requirement is, ideally, fifty per cent. It is also hoped that the same will be true of BIWs painted, namely that the achievement : requirement ratio of different BIW types should also be fifty per cent. The effect of achieving would be to maintain production in line with overall orders. It is noted that this ratio for BIWs is not catered for explicitly in the colour determination algorithm used by Istel. Colour allocation is done on the basis of the ratio *(achievement proportion)* for colours. To determine the best colour to paint a batch of BIWs, the achievement proportion for each colour is calculated by including the batch size in the value for 'achievement' (and, incidentally, not subtracting it from the requirement, which is still the denominator). The best colour is that with the lowest value for the achievement proportion. This is, apparently, to allow for the change in the colour 'achievement' when some BIWs have been allow for the change in the colour 'achievement' when some BIWs have been achieved when the batch has been processed. The criteria for establishing best batch colour are, then, partly derived from rules governing what size the batch must be.

The rules for size batching are:

i. A batch should be of at least a minimum preferred size, *minpb* (for the colour being considered);

ii. A batch should not be of more than a maximum preferred size, *maxpb*;

iii. Every BIW in the batch must have some requirement for the body/colour combination it represents;

iv. The next batch must be of a different colour, even when keeping the same colour would be possible and the most favoured (i.e. the achievement proportion would still be least for a new batch of the previous colour).

If no batch can be determined by these rules, a set of default rules applies:

v. If the previous batch can be extended by allocating the next BIW to the same colour, i.e. if it has some requirement in that colour, allocate this BIW to the colour of the previous batch (in effect, allocate a singleton batch, without changing colour). This rule applies only when the total number of BIWs painted in one sequence is less than or equal to the maximum batch size, *maxb*;

vi. If all else fails, find the least achieved colour for the next BIW (i.e. for a singleton batch).

There are extra rules which pertain (e.g. a colour is not considered if it has been switched off, through deterioration). It is also possible to prioritise a number of particular body/colour combinations so as to favour their painting. This is done by specifying, in rank order of preference, the numbers of BIWs, in particular colours, which should be painted as a priority. Thus, if any priorities have been specified, the colour scheduler looks (in order of preference) to see if a batch containing a priority body/colour combination can be scheduled. If a priority batch is found, the scheduler disregards achievement proportions of the colours and allocates the derived batch to the colour of the priority combination. In this way, the scheduler may depart from the ideal of attempting to keep production in line with requirements.

The algorithm attempts to reduce the amount that a particular colour has been under-painted. It provides for the largest possible immediate reduction in requirement for the colour, which may not produce the ideal state. It seeks, implicitly, to minimise the harmful effects on productivity caused by the limitations of the hardware's performance. This is done in several ways.

Colour constraints are of several kinds, of nominally two categories. First is the non-availability of a colour due to being unable to use it, e.g. when the supply feed becomes clogged; this is an absolute constraint. The second kind is the removal of a colour due to imperfections in its quality which will alter the proportion of painted bodies which are sub-standard and need *rectification*. These 'failures' in achievement are added back to order cover if rectification is unsuccessful. Though not represented in the reasoning, quantifiably, the behaviour of the scheduling algorithm is modified.

The first category of constraint has a random probability of occurrence, which is probably dependent on the length of time since the last clean out of the system. Thus a larger batch will, in any case, be more likely to be subject to interruption than a small one, and the risk of interruption will increase as the production shift goes on.

The latter category is stated to be due to be due to variations in the quality of paint during batches. These effects mean that optimisation is not being done on colour or batch size alone.

There is a possibility that a body may have to be be painted, by an operator, in a colour for which there is no order cover, because the scheduler has been unable to make a valid batching decision. This means that lack of order cover, i.e. requirements for the body/colour combinations on the conveyor, is a relative consideration in determining the batch size and colour.

4.2 Achievement and batch sizing

Colour and batch size are determined together. The above 'constraints' are abandoned when no 'valid' colour is possible in 'crisis allocation'.

The intent of Istel's algorithm is to maintain as closely as possible the distribution of colour/body achievement with respect to the distribution within the order cover. This suggests an ideal of a fixed ratio of one to the other.
Where achievement in colour i is Ai and total orders for i, Oi, the achievement proportion (a.p.) is $\dfrac{Ai + b}{Oi,}$ for batch size b. ..... (i)

The colour selected will be that i for which this value is least, if b is at least the minimum size (minpb).

To keep achievement in line with order cover, the ideal achievement at any time for colour, $\hat{A}i$, should be related to total orders O, Oi, and the total achievement, A: a proportion of A distributed *equally* in colour i.

So $\hat{A}i = Oi \,.\, A \,/\, O$ .... (ii)

The most under achieved colour is that for which $\hat{A}i - Ai$, the deviation from ideal production, is greatest. Consideration of b involves factors affecting both the colour constraints above and implicitly makes the distribution of orders for bodies in the various colours significant, b reflecting the bodies on the line. The use of a second measure of 'goodness', the difference between ideal and actual achievement derived from (ii) i.e. $\hat{A}i - Ai$, would have an effect, implicitly, also for body distribution because it takes into account all orders (across all colours and body types) but the first (i) does not optimise between body types. ('Good' is implied by the value for $\hat{A}i - Ai$ being small). The disparity is explained in part by the observation that batch size is biased by three 'constraints': minpb, maxpb and maxb, which vary across colours.

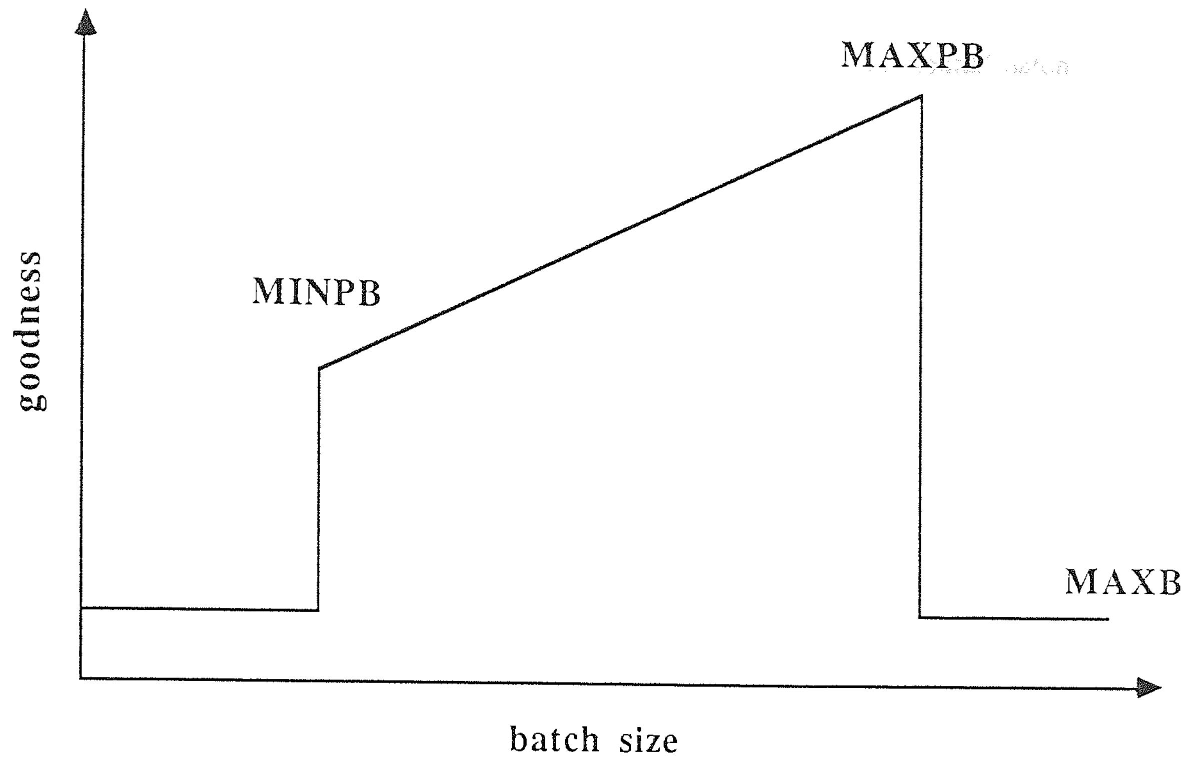


Figure 4.1

Using the 'goodness' of a decision as a measure of its desirability the batch size is regarded somewhat as shown in Figure 4.1, since, in the ranges between these constraints, no explicit preference is expressed on increasing size; the (random?) input determining what size the batch will be.

This means that between minpb and maxpb the desirability of a batch increases with the batch size. This is because the system attempts to obtain the largest batch it can. The dependence of 'goodness' of an option on increasing batch size is 'linear' in this range. Possibly undesirable consequences may follow as a result. The behaviour of the scheduler appears to be based on not discriminating between batch sizes between minpb and maxpb. So long as BIWs have requirement they are added to any potential batch, between these limits. The inclusion of each extra BIW in the batch depends on the probability of that BIW having requirement. This is probably not equal between subsequent BIWs, but the scheduler ignores any differences, so it is reasonable to regard the automatic addition of as many BIWs as possible to the batch as a linear increase of goodness with batch size. Consider three colours, pink, red and blue (ignoring body order cover distributions) with the following total initial requirements (orders) and achievements, for a potential batch of one body (Figure 4.2).

Colour	Achievement	Requirement	a.p.	ideal achievement
Pink	60	600	61/600 ≈ 0.102	127x600/1260 ≈ 60.5
Red	6	60	7/60 ≈ 0.117	127x60/1260 ≈ 6.05
Blue	60	600	61/600≈ 0.102	127x600/1260 ≈ 60.5

Figure 4.2

Thus for a batch size of one, the choice is pink or blue, based on their having the equal lowest a.p. The final deviation from balance is greater with increasing batch sizes. To see this, suppose that 40 bodies, of any colour, could be selected (Figure 4.3).

Colour	Achievement	Requirement	a.p.	ideal achievement
Pink	60	600	100/600 ≈ 0.167	166x600/1260 ≈ 79.0
Red	6	60	46/60 ≈ 0.767	166x60/1260 ≈ 7.90
Blue	60	600	100/600 ≈ 0.167	166x600/1260 ≈ 79.0

Figure 4.3

Red is *never* selected. There are two reasons: firstly, red has less (absolute) requirement than the other colours; secondly, a big batch is generally selected in preference to a small one, leaving red disadvantaged because of the relative distribution of requirement across colours and the likelihood of not having sufficient requirement left to afford any but the smallest potential batches. This gives rise to the possibility of leaving a 'pool' of requirement for body/colour combinations which is never achieved. The final deviations from ideal behaviour are generally maximised because a colour is penalised heavily by having less requirement *left* than an existing maxpb value for another colour. This says that maxpb*red* must be used in this context to ensure, *artificially*, production of red, i.e. it becomes so low as to force other colours out. Now this is exactly what maxpb does not reflect: it should be a measure of the desirability of batch size for a colour due to physical factors, such as how many cars of the same colour ought to be handled in a row, e.g. for adding trims. Indeed, the value of maxpb*red* would have to vary as requirements with respect to other colours change. As requirement for red becomes small and a red batch thus harder to select, to allow red to be scheduled, large potential batch sizes for this colour would be ignored, by using a lower value for maxpb. The lower batch size for red would then decrease its a.p. with respect to other colours and might then allow it to be selected. Maxpb and batch size cannot then determine *even* production unless all colours have similar availability of body types with similar requirements and equal batching parameters.

4.3 Ideal achievement as a measure of 'goodness'

If the above measure of deviant (non-ideal) and correctable behaviour, by using the criterion of closeness to ideality, is intended to be used as a meaningful index, how best then to use it, assuming that, in general, the bias towards batches of maxpb BIWs, wherever possible, means that it is not obeyed - although the stated aim?

In the example given, with pink and blue having equivalent order to production ratios, if, say a batch of exactly forty bodies can be done in either colour, several effects are possible. If, firstly maxpb*pink* and maxpb*blue* are greater than forty, say maxpb*pink* = 60 and maxpb*blue* = 50, and pink's a.p. is calculated before blue's, and if, being the same, it is preferred because pink is ordered before blue, say, the wrong decision is likely to have been made. This is because the values for maxpb express the convenience of having batches of a certain maximum size. The algorithm is biased towards giving a batch size as close as possible to maxpb, so, presumably, maxpb is the most desirable batch to achieve at a time. In this case the batch is two thirds of that 'optimum', whereas forty blue bodies would represent a decision of four fifths of the optimum; proportionately one fifth better to 'ideality', if the measure of that is maxpb.

Consider a second case. The proportions are as before; at this point they are in the nearest to ideal state for pink, red and blue. Assume possible batches for the colours are pink = 20, red = 3, blue = 25. $\hat{A}i$ values are approximately 69.5, 6.1, and 71.9, respectively, so the deviations from ideal, $\hat{A}i - Ai$, are approximately 11.5, 2.8 and 13.1 ($Ai\ pink = 80$, $Ai\ red = 9$, $Ai\ blue = 95$). In order to relate these to the relative distributions over production between the colours, if these differences are divided by the achievements, a clearer view is given of the differential behaviour: pink = 0.13, red = 0.05, blue = 0.15. This would suggest that red is the favourite and blue the least desirable. This might well be a realistic group of batch sizes: red = 3 is likely because few red bodies are required, implying that red is not required over many body types, which are likely to be those less in demand. Compare these values with a.p. values: pink = 80 / 600 ≈ 0.13, red = 15 / 60 = 0.25 and blue = 85 / 600 ≈ 0.14. This measure gives the same preference to pink over blue but really discriminates against red, which may be the hardest colour to allocate, thereby wasting an opportunity to achieve some difficult orders.

A third consideration is that the state after the decision is not considered accurately in the above calculations and it is an inaccuracy that may be costly. This is because the orders after the batching should have the batch size subtracted from them. The a.p. should be $Ai + b\ /\ (Oi - b\ )$. On the same basis, the ideal achievement should, more properly, be $(Oi - b) \times A\ /\ (O - b\ )$, given that, in the above calculations, b had already been added to the initial value for A. When values for Oi and O are large, relative to A and Ai, the difference may be small, but when the orders are becoming exhausted at the end of the shift the discriminatory power of the a.p. is lessened and, indeed, the value for b (as a component of the achievement quotient) becomes overly significant, and is forced to reduce. The limitation of batch size is governed by maxpb, or is supposed to be, so the algorithm would be ignoring maxpb values some of the time: the input would be determining that no batches of size maxpb were actually possible, because there would be insufficient requirement for that size of batch.

4.4 An alternative basis for ideal behaviour

The above discussion of batch sizes assumes that scheduled batch sizes are attainable. This cannot be guaranteed. Crisis allocation is able to be invoked as required. Since the physical minimum minpb*i* batch size is regarded as a constraint it might be best to schedule this number and this number alone. On successful completion of minpb*i* bodies a decision could then be taken to increase b or change colour to some more favoured value for *i*. By specifically attaching utility values to various colours according to the different criteria specified, reasoning could include more subtle and flexible methods, choosing the value with the highest score.

Chapter 5 A statistical evaluation of conventional paint scheduling

5.1 Simulator

The factory paintshop is represented by an incremental simulation of the progress of unpainted body shells through the paint section. The simulator program, **SIMULA**, is written in (un-enhanced) *C* and operated under 'standard' (BSD 4.2) *UNIX*. It functions like the control console for the paint sub-system. It comprises a database-handler, for the order cover and production totals for colours and body types and also the conveyor-line contents; a control panel for modifying constraints, priorities and order updating; and, finally, the paint allocation simulation.

From a series of menus the user has the ability to inspect and update the contents of files which contain details of the various colours and body types defined and also the map of the conveyor line. The purpose of the simulation is to demonstrate how the system should work, given the database contents. The map is updated from the end of the queue, i.e. going into the paint section from assembly; those bodies shown in the map are in the 'visible' region for paint-spraying purposes and those at the head of the queue are ready for painting.

The user can only add bodies to the queue as those at the head are processed. Details of outstanding orders, production totals and constraints on colours, etc. are updatable. The painting simulation can be operated in two ways. The current line sequence is analysed with respect to the order cover profiles of colours and body types - according to the algorithm in Istel's functional specification (Istel plc 1985). The user can select batch operation, in which the program finds the optimal batch size and colour, processing that number of bodies in the colour determined and updating the database accordingly, or single-step simulation. For single-step operation the user is presented with the optimal batch size and colour and then indicates if the program should paint the leading body as determined. The user can opt to over-ride that and is informed of the alternatives. If desired an invalid colour/body type combination may be selected but will be queried. By single-stepping through a batch the user can simulate the effect of a sudden colour constraint, e.g. due to degradation of paint quality.

There are several co-related 'functionalities' represented in the above simulator. The database-handling 'contains' the efforts of the tracking and other sub-systems, with regard to orders and location of system entities. It also reflects the action of a booth operator in changing system state (e.g. constraint posting) and over-riding of system decisions (e.g. colour allocation) as well as the painting sub-system itself.

5.2 Statistical analysis

A statistical simulation of the behaviour of the colour allocation algorithm is possible using two programs, ORDER and PAINT.

The menu-driven program, ORDER, allows:

a) the generation of semi-randomised line sequences for processing;

b) the production of an order cover profile to allow the product of a) to be processed;

c) the processing (colour allocation) of bodies on the line by calling PAINT;

d) analysis of the result of c).

PAINT is a stand-alone implementation of the allocation algorithm in SIMULA which incorporates the ability to update the map and orders in a trial database automatically after every batching decision. These two programs allow batch testing of potential operating situations. It should be noted that alteration of priority and colour availability constraints, and removal of orders is not done in the processing. The purpose of the statistical analysis is to examine how well the algorithm responds to a given database as a whole, in effect a simulation of ideal operating conditions.

5.2.1 Line sequence generation

The generation of line sequences is done according to the order profile of a given database (i.e. the relative proportion of orders for each body type/colour combination). The degree of randomness of the sequence also depends on the colour requirements across the different body types, in that its complement of body types is covered by the overall orders in the profile and can always, *a priori*, be processed without any bodies requiring painting without order cover (i.e. over-painting). The distribution of body types in the sequence is biased by probabilistic means: the first body is chosen randomly. The use of this is to see how order profile (mirrored overall in the line sequence) relates to production as the randomness of the sequence varies.

5.2.2 Order cover profile maintenance

A given profile can be scaled up to accommodate the volume of bodies in a sequence. This means that, if say a sequence of 10000 bodies has been generated and the orders total 5000, the profile can be kept relatively equivalent but orders (and production) for the individual body type/colour combinations are scaled by an appropriate factor up to, say, 15000.

5.2.3 Results of paint batching

The version of the paint scheduling scheme used (PAINT) records details of each batch found (size, colour, allocation failure et c.) and outputs a file of results.

5.2.4 Analysis of batching optimality

ORDER can produce various statistical measures of how a given sequence and profile behaved in processing. The (intra-)variation in achievements (production) for colours, body types and body type/colour combinations are computed and show how even production was. An evaluation of the effect of batching parameters (e.g. maxpb) on production is made by calculating what proportions of the batches fall into the size groups (singleton, minpb to maxpb etc.) These measures are taken as indicating some effect of the profile on production. A *UNIX* script is used to reproduce a set of sequences based on the same profile, and therefore statistically heterogeneous, which are then processed and analysed. The statistical data are then considered over all sequences to show the effect of a given randomness in input on performance, under the constraint of a known order distribution, on performance. Measures such as deviations of performance across colours are then examinable: there should be no significant difference between colours if the algorithm is producing ideal behaviour. This technique also shows what degree of randomness in input (of sequences) leads to poor behaviour, e.g. a large deviation in achievement, from the order profile, across body types. Such undesirable effects should be shown to be due to the algorithm, i.e. across a series of different profiles and randomness of input, when considering the performance of any knowledge-based scheme.

A discussion of the optimisation possible using the colour allocation scheme is contained in the following sections.

5.3 Statistical estimates of 'goodness'

When the PAINT program is run, it produces a batching decision. The colour and size for the batch are recorded. Any 'failures' are noted as singleton batches. For each batching decision ORDER calculates the deviation from ideal that this represents and also the proportion of batches that fall in the 'correct' ranges, i.e. minpb to maxpb, and in the failed ranges. The total deviations for each run as a whole are calculated to show the overall performance of the algorithm on that sequence. A similar statistical analysis is made of the body distribution, to see if the colours differ in their deviations of body achievement. The aim is to see whether there a significant deviation of production of different body types from orders.

The program, BIAS, allows a particular distribution of body types to be generated for the sequence file. This is done by selecting a body type and producing a sequence of this type. The number of each type that is repeated in the sequence is governed by a statistical distribution. This statistical distribution selects a sequence that is always entirely achievable. It determines the number of repetitions of each body type according to the proportion of groups that should be of particular ranges, i.e less than minpb, minpb to maxpb, greater than maxpb to maxb and greater than maxb.

Using scripts to call ORDER and PAINT repeatedly on different sequences, of the same overall body distribution, generated from bias values generated by the BIAS program - i.e. degree of randomisation - using the same order profile a number of sequences of the same overall distribution are produced. Analysis of the results of the individual runs is done to compare the effect of the distribution of body types in sequences with a given order profile on the performance of the algorithm. If this is done for a set of order profiles, with increasing matrices of body type x colour, the effect of increasing degrees of freedom from colour limitation can be examined. This means that if the deviation from ideal behaviour is dependent on those degrees of freedom, it should decrease as the matrix gets larger. If it is much the same for all matrices then it is evidence that the algorithm is sensitive to randomness.

The results of a set of tests on different order profiles are given and discussed in the rest of this chapter.

5.4 Experimental conditions

A series of experiments was carried out to estimate the effect of randomness in input (BIW types entering painting) on the performance of the conventional algorithm, in terms of evenness of production across body types and colours. It was stated in the original documentation (Istel plc 1985) that the totals (i.e for the rolling schedules) were accounted for on an eighteen-day basis and that updating would be at least at daily intervals. It now appears that a more general practice is to use a ten-shift (five-day) order bank (with possibly two updates per day, each shift starting with a revised schedule). For this reason, the simulation of the paint shop in these experiments was done on the basis of (as close as possible to) 10000 total BIWs to be painted and, assuming an ideal starting state, 5000 total bodies painted. The sequence to be processed was 2000 BIWs. This, then, represents some 20% of the initial total. This number was chosen to represent a significant proportion of requirement in order to test the algorithm under less than marginal conditions, i.e. when the total change in the ratio of required to achieved is not very small. The value of 20% of the requirement to be processed is somewhat arbitrary, but was felt to be not so big as to produce a distortion in the algorithm's behaviour due to lack of requirement left, at any stage, but large enough to test it under some realistic 'loading'. It is, in any case, the proportion of requirement represented by a whole day's processing (i.e. without an update half-way through). The same set of bias values was used to generate sets of sequences (of increasing randomness) for order profiles of increasing size. The smallest profile contained a 10 × 10 matrix of colours and body types. This was increased by stages to the largest profile of a 24 × 24 matrix. In all cases the occupancy (≈ 50%) was such that few colours had few bodies available, few colours had most available and most had approximately half.

The results were assessed in terms of whether colours had uneven production (with respect to their initial proportion of all requirements), whether any such unevenness increased with randomness and whether the amount of freedom of choice for the scheduler (as represented by the size of the matrix: the largest is close to an order of magnitude greater than the smallest in terms of body/colour combinations) produces a different sensitivity to randomness of input. The measure of randomness is not a scalar quantity. (It is based on the probability of one BIW being the same the one ahead of it in the queue.) The randomness is represented by an increasing tendency, so the overall trend is available, but the exact statistical relationship of one datum to another, when compared with 'randomness', is uncertain.

5.5 Measures of randomness

The biasing values used to produce the sequences are shown below (Figure 5.1). The three parameters in each biasing set determine the probability of the repetition of any body type in the sequence. This is done according to the batching parameters for the colours, in that the values of minpb, maxpb and maxb are averaged across all colours. If the values are *a, b* and *c*, for example, the chance of any body being in a group of less than (average) minpb (*minpb*) bodies of the same type is a: so that the proportion of groups of less then *minpb* bodies of the same type tends to *a*. Using the same notation for mean values as for *minpb*, the chance for a group of less than *minpb* is then *b*, for groups of *minpb* to *maxb* it is *c*. If *c* is less than unity, batches of at least *maxb* size have the probability of 1 - *c*. This is intended to give varying distributions for the sizes of groups of the same body type in sequences (and is applied to all order profiles). If the value of *a* is large, then most bodies will not be repeated often in a group and if *a* and *b* are small relative to *c*, a body is likely to be repeated many times. It can be seen that the first bias set (#1) is moderately random, in that most body types are biased to appear in large groups, whereas the last (#34) should produce a very random sequence. The reason for the use of a mean value for these grouping bounds (i.e. the colour batching parameters) to decide how often a body should be repeated in sequence is not immediately obvious. This strategy is based on the fact that the batching parameters are related to the number of achieved BIWs in a batch that can be tolerated, in that these painted bodies are stored in 'batches' and there is a size of 'batch' for different colours that is more convenient than another. Thus the behaviour of the paint scheduler is made dependent on factors not immediately concerned with orders or painting criteria. (The grouping of BIWs placed on the conveyor may be related to the storage grouping of painted bodies in the factory.)

When making a sequence of BIWs for these experiments, the total BIWs of each body type are constrained to be within the requirement in the order profile. To achieve this (so that no painting 'failures' occur due to lack of requirement for any BIW in all colours) the following scheme is applied. A body type is chosen according to the proportion of total requirement that it has. If body types 1, 2 and 3 have requirement in the ratio 1:2:1, for example, then type 1 is given a 25% chance of being selected (and type 2 50%, type 3 25%, respectively). This is done by giving a range of values to each type (e.g. 0 to 0.25, > 0.25 to 0.75 and > 0.75). A random number is generated (between 0 and 1) and the range that this value falls in decides which type is selected. As the number of BIWs selected increases, these ranges are adjusted so that the (body type) requirement, *less* the number of BIWs already allocated of that type, is used: the outstanding requirement decreases so the chance of picking the body type is reduced accordingly. Having selected a body type to allocate, a certain number of BIWs of that type will be entered into the sequence. The number of these BIWs depends on the biasing parameters (*minpb* etc.) above. A size for a group of the selected BIW type is first selected. This is done in a similar manner to the selection of body type. For the above values *a, b* and *c*, the range for a group of size less than *minpb*, is 0 to less than *a*, for greater than *minpb* to *maxpb*: *a* to less than *b*, and so on. Again, a random number (in the range 0 to 1) is used to select one of these. The size is initialised with the (integral) minimum of these ranges. It is then increased towards the upper limit. For a range of *n* bodies, the chances of increasing by 1..n bodies is determined by taking another random number, in the range 0 to 1. The interval 0 to 1 is then partitioned into *n* equal ranges and the range that the random number lies within is selected. Since the range represents one of 1/n, 2/n, 3/n etc., the quotient of these shows how many extra bodies (than minimum) should be added.

Given this method of biasing, there is no even degree of increasing randomness from biases #1 to #34 (below). Sets of parameters produce a range of randomness in sequences. A linear estimate has been made of the randomness of a sequence by calculating the average size of all groups of the same repeated body type. If the number of consecutive BIWs of the same type is decreasing, on average, then it is taken that the randomness is increasing overall.

5.6 Statistical measures of performance

For each profile, 12 sequences of equivalent distribution were generated and processed (for each of the 33 bias value sets). Statistics for deviations are collected for colours, bodies and also deviation from ideal behaviour (see Chapter 4). For both colour and body measures, the value of requirement (initial) *less* achievement (final) *divided by* the initial requirement is calculated (i.e. for each individual colour instance, and for the total bodies processed in each colour). This should show how even or unbalanced production might be across the various colours (each compared on the same basis). This ratio (i.e. achievement proportion) is set to 50% at the start and should remain so if the scheduling algorithm achieves its purpose. The corresponding values for each of the 12 sequences for the same bias and profile are added together and their averages are recorded. (This is done for colours and bodies in colours). This is to see whether a general trend exists from one bias to another (the effect of randomness) and also to see if the size of the matrix changes the sensitivity to randomness (i.e. across different profiles). As indicated in Chapter 4, the 'ideal deviation' may give a more accurate view of the balance in production. This is calcualated with respect to the whole achievement and requirement rather than just within individual colours. Thus, it should be a more sensitive test of unbalanced production and comparisons between values for different colours should be more significant. (This tests whether the algorithm behaves as it should, i.e. no qualitative difference should be present between the ideal value deviations and those calculated from a.p.) Deviation from 'ideality' is also estimated for the bias type as a whole (across all colours). The total number of batches in each sequence is recorded for each colour and the cumulated total is saved for the overall summary. This is done also for the singleton batches found.

The sets of bias values used to produce sequences are detailed in Figure 5.1 and part of a specimen sequence file for the first profile (and moderately random distribution) is listed in Figure 5.1.

The values stated in the tables (Figures 5.9 to 5.14 inclusive) were generated by the program ORDER or calculated afterwards from values recorded during the runs. The measurement used as ordinal for many of the graphs, mean BIWs/group, is a derived value. The number of BIWs processed, i.e. 2000 in every case, was divided by the number of groups of BIWs in the sequence. This value was taken to be the average group size. The 12 replicates' values for mean BIWs/group were then averaged to give the stated result.

Using the same principle, the mean batch size produced was calculated from the number of batches resulting from each run and the results for the 12 replicate sequences averaged to give the values stated The values given for singleton batches were also averaged for the 12 replicate sequences for each bias set .

For the colour deviations, the reduction in the initial outstanding requirement achieved for each colour was found and divided by the (initial) outstanding requirement, i.e. a form of achievement proportion. The mean value of these observations was found and subtracted from each achievement proportion in the sequence. Each value of difference was then squared and the sum recorded. This sum was then 'normalised' by dividing it by the number (minus one) of colours used - as it is over a small sample - and taking its square root. These numbers (one per sequence) were summed and averaged over the 12 replicates. (For ideal results, since 2000 BIWs are painted, the initial requirement of 10000 reducing to 8000, the expected number of BIWs painted in each colour is 20% of the initial requirement. The deviation from this, across colours, is a good indication of how even production was.) The values are listed under 'colour' (deviation).

The 'body' deviation is a measure of how evenly production was distributed across different body/colour combinations. Like the colour value above, the proportionate change in initial requirement is calulated for each individual body/colour combination. The difference from the mean of all results is squared and the sum of these values divided by the number of combinations (minus one). The square roots of the resulting values for each sequence are then averaged across the replicates and the result given under body (deviation).

For the results stated, assume the following definitions:

n_c = number of colours in the profile;

n_t = number of body/colour types in the profile;

c_i = colour of the i'th colour in the profile;

t_i = i'th body/colour type in the profile;

oc_i = initial (outstanding) requirement of the i'th colour in the profile;

ac_i = number of all BIWs painted in colour c_i (for the sequence);

ot_i = initial (outstanding) requirement of BIWs of type t_i;

at_i = number of all BIWs painted of body/colour type t_i (for the sequence);

pc_i = 'colour achievement proportion' (after batching) for the i'th colour;

pt_i = 'body achievement proportion' (after batching) for the i'th body/colour type;

$\overline{pc}$ = mean of the pc_i values;

$\overline{pt}$ = mean of the pt_i values.

$pc_i = (oc_i - ac_i) / oc_i$, thus $\overline{pc} = \sum_{i=1}^{i=n_c} pc_i \;/\; n_c$;

the colour deviation is then $\sqrt{\left[\sum_{i=1}^{i=n_c} (pc_i - \overline{pc})^2 / (n_c - 1)\right]}$.

$pt_i = (ot_i - ati) / ot_i$, thus $\overline{pt} = \sum_{i=1}^{i=n_t} pt_i \;/\; n_t$;

the body deviation is then $\sqrt{\left[\sum_{i=1}^{i=n_t} (pt_i - \overline{pt})^2 / (n_t - 1)\right]}$.

Distribution biases:

up to minpb	minpb to maxpb	more than maxpb
0.00	1.0	1.0

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  .... group of 24 ....

3 3 3 3 3 3
  .... group of 6 ....

1 1 1
  .... group of 3 ....

6 6 6
  .... group of 3 ....

1 1 1 1
1 1 1 1 1 1 1
  .... group of 11 ....

8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
  .... group of 29 ....

2 2 2 2
2 2 2
  .... group of 7 ....

1 1 1 1 1 1 1
  .... group of 7 ....

3 3 3
  .... group of 3 ....

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  .... group of 27 ....

3 3 3 3 3 3 3 3 3 3 3 3 3
  .... group of 13 ....

6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
  .... group of 25 ....

4 4
4 4 4 4 4
  .... group of 7 ....

3 3 3 3 3 3 3 3 3 3 3 3 3 3
  .... group of 14 ....

1 1 1 1 1 1 1
  .... group of 7 ....
```

Part of a sequence generated from bias set #1

Figure 5.1

Item #	Low	Medium	High
1	0.00	1.00	1.00
2	0.00	0.75	1.00
3	0.00	0.75	0.75
4	0.00	0.50	1.00
5	0.00	0.50	0.75
6	0.00	0.50	0.50
7	0.00	0.25	1.00
8	0.00	0.25	0.75
9	0.00	0.25	0.50
10	0.00	0.25	0.25
11	0.00	0.00	1.0
12	0.00	0.00	0.75
13	0.00	0.00	0.50
14	0.00	0.00	0.25
15	0.00	0.00	0.00
16	0.25	1.0	1.0
17	0.25	0.75	1.0
18	0.25	0.75	0.75
19	0.25	0.50	1.0
20	0.25	0.50	0.75
21	0.25	0.50	0.50
22	0.25	0.25	1.0
23	0.25	0.25	0.75
24	0.25	0.25	0.50
25	0.25	0.25	0.25
26	0.50	1.0	1.0
27	0.50	0.75	1.0
28	0.50	0.75	0.75
29	0.50	0.50	1.0
30	0.50	0.50	0.75
31	0.50	0.50	0.50
32	0.75	1.0	1.0
33	0.75	0.75	1.0
34	1.00	1.0	1.0

Details of bias distributions

Figure 5.2

5.7 Testing the Istel algorithm

In this section, tables showing the initial conditions for order profiles are presented. The results of tests of the painting scheduler are discussed in section 5.4. The values given show the colour order for each of the matrices used (Figures 5.3 to 5.8, see Appendix for the respective body file profiles). The summary values (overall) for each bias for each profile are presented in Figures 5.9 to 5.14. (The individual results for each colour are not given, some examples are shown for the first profile - the 10 × 10 matrix.) Following these results, a series of graphs are included to show the effect of randomness of input on various of the above parameters of performance. A large number of sequences were generated for each order profile. For this reason, individual results of the processing of each sequence are not stated. The average results of each twelve replicates per bias set are stated for overall production (rather than for each colour, the deviation across colours being used to assess individual colour's effects on behaviour).The biasing scheme means that some points have very similar ordinal values (mean group size) but are generated by very different bias value sets, which produce similar mean values, however.

Colours allowed	Max. used	(Total colour items set)
32	10	50

Colour	minpb	maxpb	maxb	Required	Achieved
1+	3	50	75	125	250
2+	5	34	53	1512	756
3+	4	40	60	908	454
4+	4	20	58	1256	628
5+	3	35	80	1160	580
6+	3	35	60	1572	786
7+	5	15	40	1064	532
8+	4	35	60	900	450
9+	3	40	55	684	342
10+	4	47	60	694	347

Profile of order cover #1: 10 ×10 matrix

Figure 5.3

Colours allowed	Max. used	(Total colour items set)
32	12	70

Colour	minpb	maxpb	maxb	Required	Achieved
1+	3	50	75	422	211
2+	5	34	53	452	226
3+	4	40	60	740	370
4+	4	20	58	356	178
5+	3	35	80	982	491
6+	3	35	60	1226	613
7+	5	15	40	886	443
8+	4	35	60	884	431
9+	3	40	55	1156	578
10+	4	47	60	1692	846
11+	6	40	80	1646	823
12+	3	31	49	1620	810

Profile of order cover #2: 12 ×12 matrix

Figure 5.4

Colours allowed	Max. used	(Total colour items set)
32	14	94

Colour	minpb	maxpb	maxb	Required	Achieved
1+	3	50	75	444	222
2+	5	34	53	698	349
3+	4	40	60	460	230
4+	4	20	58	752	376
5+	3	35	80	694	347
6+	3	35	60	910	455
7+	5	15	40	598	299
8+	4	35	60	1040	520
9+	3	40	55	762	381
10+	4	47	60	1330	665
11+	6	40	80	1160	580
12+	3	31	49	1276	638
13+	3	37	62	766	383
14+	4	53	80	490	245

Profile of order cover #3: 14 ×14 matrix

Figure 5.5

Colours allowed	Max. used	(Total colour items set)
32	16	124

Colour	minpb	maxpb	maxb	Required	Achieved
1+	3	50	75	420	210
2+	5	34	53	568	284
3+	4	40	60	440	220
4+	4	20	58	436	218
5+	3	35	80	718	359
6+	3	35	60	522	261
7+	5	15	40	960	480
8+	4	35	60	670	335
9+	3	40	55	966	484
10+	4	47	60	890	445
11+	6	40	80	918	459
12+	3	31	49	1118	559
13+	3	37	62	1000	500
14+	4	53	80	410	205
15+	5	56	69	406	203
16+	4	34	55	424	212

Profile of order cover #4: 16 $\times$16 matrix

Figure 5.6

Colours allowed	Max. used	(Total colour items set)
32	18	165

Colour	minpb	maxpb	maxb	Required	Achieved
1+	3	50	75	322	161
2+	5	34	53	396	198
3+	4	40	60	338	169
4+	4	20	58	418	209
5+	3	35	80	596	298
6+	3	35	60	526	263
7+	5	15	40	588	294
8+	4	35	60	610	305
9+	3	40	55	666	333
10+	4	47	60	714	357
11+	6	40	80	912	456
12+	3	31	49	824	412
13+	3	37	62	760	380
14+	4	53	80	400	200
15+	5	56	69	356	178
16+	4	34	55	404	202
17+	6	50	75	784	392
18+	3	35	54	404	202

Profile of order cover #5: 18 $\times$18 matrix

Figure 5.7

Colours allowed	Max. used	(Total colour items set)
32	20	198

Colour	minpb	maxpb	maxb	Required	Achieved
1+	3	50	75	260	130
2+	5	34	53	510	255
3+	4	40	60	326	163
4+	4	20	58	386	193
5+	3	35	80	566	283
6+	3	35	60	422	211
7+	5	15	40	560	280
8+	4	35	60	582	291
9+	3	40	55	560	280
10+	4	47	60	602	301
11+	6	40	80	704	352
12+	3	31	49	748	374
13+	3	37	62	574	287
14+	4	53	80	396	198
15+	5	56	69	290	145
16+	4	34	55	342	171
17+	6	50	75	568	284
18+	3	35	54	296	148
19+	4	34	45	528	264
20+	5	20	60	780	390

Profile of order cover #6: 20 × 20 matrix

Figure 5.8

5.8 The effect of randomness on production

Differences in production result from different biases. This becomes marked as the degree of randomness of input becomes large. Graphs showing various parameters plotted against the mean size of groups (of the same body type) in the input sequence show an increased randomness affecting behaviour (for all order cover profiles) There is also an effect of the increasing degree of freedom of colour choice as the matrix becomes larger, although the overall trend is similar. Figures 5.15, 5.16, and 5.17, 5.18, respectively, show the effect of increasing randomness on the size of averages batches produced and the number of singleton batches.

Values for batches and singles are given as the total over all (12) runs, and their averages taken for plotting graphs.

10 colours

#	Group size	Batch size	Singletons	Colour dev.	Body dev.
1	12.8	14.6	10.8	0.021	7.90
2	13.9	15.7	9.92	0.018	7.29
3	14.7	15.6	12.4	0.013	9.49
4	19.3	18.2	7.83	0.012	7.22
5	21.1	18.3	8.92	0.015	7.91
6	16.8	17.5	9.00	0.020	5.70
7	27.8	18.7	10.4	0.015	6.07
8	31.0	20.3	6.67	0.016	5.22
9	33.6	20.3	8.58	0.020	5.07
10	35.9	19.0	9.92	0.022	4.28
11	53.7	23.5	5.92	0.017	4.57
12	53.0	24.0	6.58	0.020	5.23
13	53.7	24.0	4.50	0.015	8.72
14	60.1	23.8	7.67	0.023	6.83
15	74.1	24.7	3.00	0.018	7.16
16	7.20	10.4	63.0	0.020	5.63
17	9.13	12.0	50.3	0.019	4.65
18	9.91	12.4	48.9	0.022	5.18
19	14.5	15.3	35.8	0.016	6.70
20	16.0	16.3	32.8	0.019	8.34
21	11.9	16.1	26.3	0.024	8.52
22	27.9	19.2	18.6	0.018	6.01
23	26.5	19.8	16.6	0.024	8.35
24	27.1	17.4	32.7	0.023	4.26
25	3.90	7.46	127	0.023	4.37
26	5.57	9.26	96.8	0.021	5.12
27	8.50	10.7	74.3	0.020	3.23
28	9.27	13.4	55.3	0.019	7.29
29	18.3	14.3	48.3	0.018	5.00
30	10.2	14.0	49.0	0.028	8.45
31	2.75	3.94	302	0.029	1.98
32	3.73	5.63	204	0.025	2.22
33	3.99	8.00	119	0.022	3.84
34	1.74	2.81	441	0.034	0.995

Statistics for processing order profile #1 (10 × 10 matrix)

Figure 5.9

12 colours

#	Group size	Batch size	Singletons	Colour dev.	Body dev.
1	12.7	16.9	1.25	0.020	7.72
2	14.6	17.7	1.33	0.017	8.73
3	16.3	18.5	3.25	0.018	7.39
4	22.0	20.1	1.83	0.016	8.26
5	22.5	21.3	1.08	0.017	9.17
6	22.2	21.0	1.50	0.022	11.8
7	25.6	21.0	3.08	0.023	14.0
8	28.6	23.1	1.42	0.019	11.7
9	33.9	23.2	3.42	0.021	11.9
10	33.9	23.9	2.17	0.023	11.5
11	51.3	31.5	0.917	0.021	10.7
12	52.6	26.8	1.17	0.021	13.5
13	52.6	26.1	3.33	0.028	14.5
14	60.6	27.9	1.17	0.032	16.1
15	71.5	29.2	0.833	0.029	15.6
16	7.66	14.6	22.8	0.032	10.7
17	10.2	14.6	15.8	0.036	9.09
18	9.86	19.4	9.50	0.035	11.6
19	17.9	21.1	73.8	0.026	11.0
20	21.1	22.4	6.50	0.026	10.1
21	19.2	22.8	5.58	0.028	10.3
22	22.5	24.5	2.50	0.025	9.98
23	25.3	25.4	4.00	0.027	11.2
24	27.0	25.0	7.50	0.033	14.7
25	5.25	13.2	35.9	0.047	12.9
26	7.17	18.0	15.9	0.040	10.7
27	7.58	18.7	14.5	0.038	11.7
28	14.2	23.2	2.50	0.031	8.32
29	14.3	22.8	4.83	0.032	9.30
30	12.8	25.1	4.33	0.024	9.80
31	2.68	7.17	151	0.056	12.7
32	4.81	14.6	137	0.041	10.9
33	4.69	17.2	116	0.041	10.1
34	1.69	3.28	375	0.347	22.1

Statistics for processing order profile #2 (12 $\times$ 12 matrix)

Figure 5.10

14 colours

#	Group size	Batch size	Singletons	Colour dev.	Body dev.
1	15.1	15.0	1.50	0.013	4.25
2	17.1	17.1	1.08	0.012	3.72
3	17.1	17.1	0.667	0.011	5.59
4	19.6	19.4	0.417	0.010	6.15
5	18.9	18.9	1.42	0.010	7.92
6	20.1	20.1	1.17	0.009	5.71
7	21.6	21.6	0.500	0.011	6.31
8	21.6	21.6	1.42	0.011	6.03
9	21.6	21.7	2.83	0.012	8.81
10	23.7	23.7	1.08	0.011	6.47
11	25.5	25.5	2.08	0.012	9.50
12	25.9	25.9	1.17	0.011	8.97
13	26.0	26.0	1.17	0.010	8.34
14	26.1	26.1	1.50	0.011	9.44
15	27.5	27.4	0.417	0.013	11.4
16	15.0	14.9	15.8	0.038	8.16
17	17.9	17.9	7.67	0.031	7.54
18	18.7	18.7	6.08	0.029	6.86
19	20.3	20.3	6.42	0.024	6.65
20	21.8	21.8	4.25	0.022	7.69
21	23.5	23.6	1.83	0.019	7.26
22	24.2	24.2	2.17	0.019	7.50
23	25.0	25.0	1.17	0.017	7.13
24	26.0	26.0	1.08	0.016	6.48
25	11.0	11.0	52.7	0.040	9.63
26	15.6	15.6	22.9	0.037	8.86
27	16.8	16.8	18.9	0.033	8.66
28	19.8	19.8	10.9	0.027	9.27
29	21.8	21.8	5.25	0.024	6.89
30	23.1	23.0	6.08	0.020	11.2
31	6.30	6.29	151	0.050	10.5
32	11.6	11.6	55.8	0.038	8.87
33	16.2	16.2	24.3	0.038	9.43
34	1.64	3.68	307	0.050	10.3

Statistics for processing order profile #3 (14 $\times$ 14 matrix)

Figure 5.11

16 colours

#	Group size	Batch size	Singletons	Colour dev.	Body dev.
1	12.9	14.1	2.33	0.006	7.68
2	15.2	15.7	0.333	0.007	6.70
3	15.2	16.4	1.00	0.007	6.29
4	22.1	17.9	1.25	0.007	8.09
5	22.5	18.5	0.917	0.008	6.39
6	4.4	19.4	1.42	0.008	8.20
7	28.8	20.7	0.500	0.008	7.75
8	28.9	19.8	0.750	0.008	7.67
9	35.1	20.9	1.00	0.009	8.33
10	32.9	22.2	1.25	0.010	8.29
11	51.0	24.4	0.750	0.013	10.8
12	51.0	25.0	0.250	0.010	8.55
13	53.4	24.5	0.667	0.011	9.57
14	57.2	25.2	0.667	0.011	10.0
15	71.5	25.9	0.750	0.013	11.8
16	8.00	13.6	16.8	0.021	6.39
17	9.65	16.7	5.08	0.018	5.73
18	10.8	17.7	2.75	0.017	5.58
19	17.6	19.8	4.58	0.014	6.05
20	19.0	20.0	4.92	0.013	7.62
21	19.0	21.6	4.67	0.010	6.74
22	25.1	22.8	2.25	0.014	6.90
23	25.2	23.2	3.42	0.014	8.34
24	29.2	24.0	1.50	0.013	7.41
25	5.28	11.9	32.4	0.028	6.86
26	6.90	15.0	19.3	0.022	7.10
27	10.2	16.1	16.1	0.023	6.73
28	14.9	20.1	5.75	0.016	7.38
29	17.5	20.1	6.67	0.018	7.21
30	17.5	22.4	3.58	0.014	7.12
31	2.71	6.27	110	0.027	6.88
32	5.05	12.0	34.8	0.023	8.39
33	1.61	13.9	26.2	0.019	8.72
34	1.65	3.39	345	0.473	14.7

Statistics for processing order profile #4 (16 $\times$ 16 matrix)

Figure 5.12

18 colours

#	Group size	Batch size	Singletons	Colour dev.	Body dev.
1	14.2	14.1	0.750	0.007	6.00
2	16.4	14.7	0.500	0.006	6.29
3	16.7	14.7	0.333	0.007	5.98
4	22.3	17.9	0.583	0.007	5.57
5	25.1	17.2	0.417	0.008	6.33
6	28.6	18.3	0.167	0.007	6.77
7	29.1	19.2	0.083	0.008	6.93
8	31.8	19.2	0.833	0.009	7.31
9	37.4	20.0	0.417	0.009	7.07
10	36.2	21.5	0.500	0.013	7.37
11	50.7	23.2	0.667	0.011	8.59
12	51.6	23.3	0.583	0.013	8.81
13	56.8	24.0	0.250	0.011	8.01
14	56.8	24.9	0.083	0.012	7.78
15	69.5	25.9	0.750	0.012	8.21
16	8.54	13.8	7.92	0.022	5.51
17	10.2	15.3	6.17	0.018	5.09
18	12.8	16.4	6.17	0.018	6.05
19	15.9	19.0	3.00	0.011	6.07
20	16.1	20.0	3.33	0.013	6.35
21	24.2	20.4	2.75	0.012	6.99
22	26.3	22.0	2.33	0.015	6.59
23	32.1	22.5	1.67	0.014	7.13
24	33.8	23.2	2.75	0.012	7.60
25	6.10	11.4	29.0	0.027	5.95
26	8.00	14.2	13.8	0.023	5.96
27	9.92	15.7	8.92	0.021	5.84
28	12.8	18.2	6.75	0.018	7.08
29	14.3	19.6	2.92	0.018	6.40
30	18.7	21.2	2.75	0.014	6.99
31	3.05	6.97	74.0	0.032	5.40
32	5.50	12.7	31.4	0.027	5.98
33	2.07	10.8	24.3	0.024	5.97
34	2.01	4.73	133	0.034	4.67

Statistics for processing order profile #5 (18 × 18 matrix)

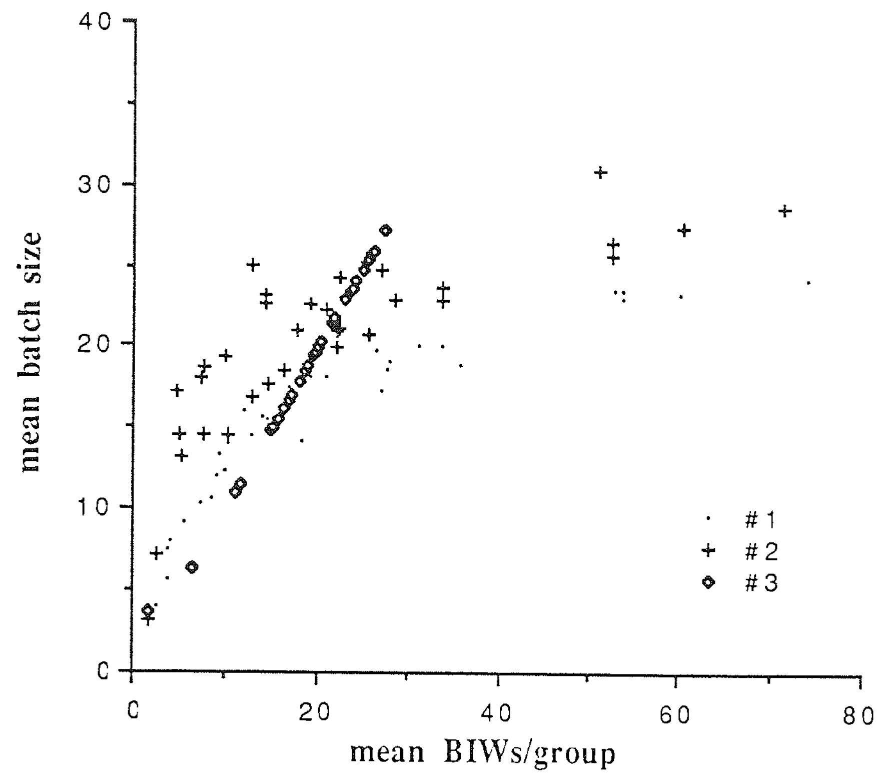
Figure 5.13

20 colours

#	Group size	Batch size	Singletons	Colour dev.	Body dev.
1	13.5	12.7	0.250	0.006	5.30
2	15.1	13.5	0.500	0.006	5.71
3	16.8	14.0	0.000	0.006	5.44
4	22.2	15.6	1.42	0.008	6.41
5	20.2	15.6	0.167	0.007	5.94
6	25.0	17.4	0.250	0.008	6.40
7	26.2	17.7	0.250	0.009	6.31
8	29.6	16.9	0.833	0.007	6.21
9	33.0	18.0	0.583	0.009	6.51
10	35.0	18.9	0.417	0.010	6.58
11	49.7	21.1	1.17	0.013	7.17
12	53.0	21.9	0.750	0.012	7.31
13	56.8	21.7	1.08	0.012	7.47
14	61.1	21.9	1.25	0.012	7.86
15	71.5	22.8	1.17	0.010	7.26
16	7.99	12.9	6.58	0.012	5.10
17	9.30	14.1	5.58	0.012	5.39
18	11.0	14.6	5.75	0.013	5.43
19	14.6	17.2	3.83	0.012	6.67
20	15.7	17.9	3.92	0.012	6.29
21	18.6	18.3	3.50	0.010	6.27
22	25.5	19.4	1.25	0.012	6.23
23	25.2	20.6	0.667	0.010	5.98
24	26.5	20.5	3.00	0.011	6.73
25	5.70	11.1	17.5	0.015	5.45
26	6.41	13.3	8.17	0.014	5.60
27	8.26	14.1	8.75	0.013	5.05
28	13.3	17.4	5.33	0.011	6.27
29	12.3	16.7	8.17	0.015	6.97
30	14.2	18.2	3.50	0.012	6.93
31	2.90	7.07	46.9	0.016	4.81
32	5.21	10.3	24.5	0.013	6.29
33	6.05	12.5	9.92	0.014	6.63
34	1.97	4.99	95.2	0.021	4.31

Statistics for processing order profile #6 (20 $\times$ 20 matrix)

Figure 5.14

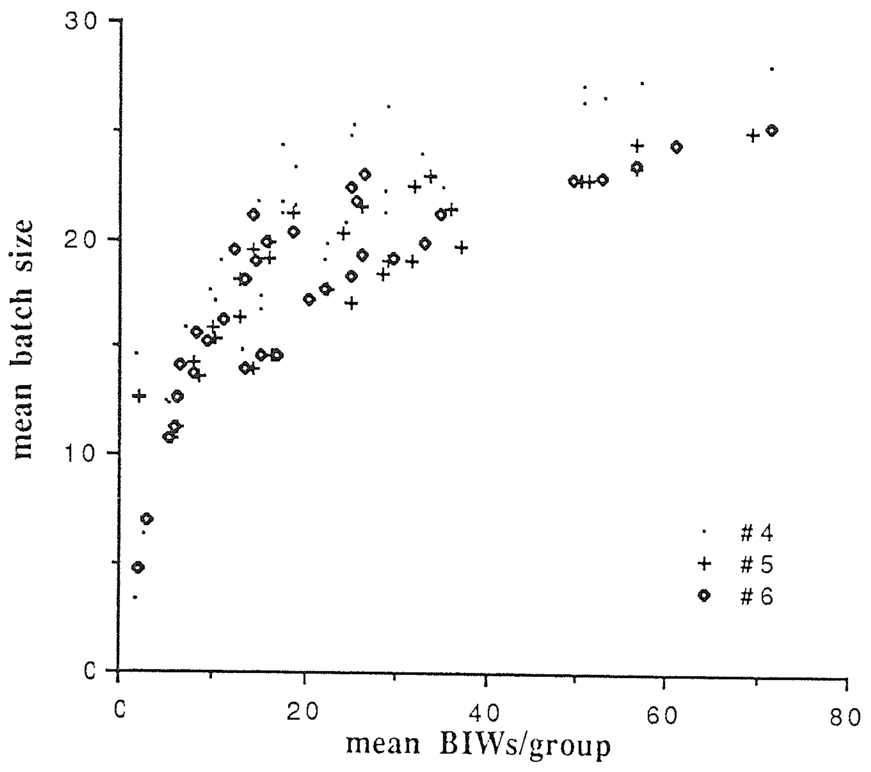


Mean batch size vs. mean BIWs/group (profiles #1 to #3)

Figure 5.15

In Figures 5.15 and 5.16, the results from different order profiles are shown together. The plot points are distinguished by the use of different symbols. The figure preceded by the hash sign indicates which order profile is associated with which symbol. This convention is used for the other graphs where three profiles are compared.

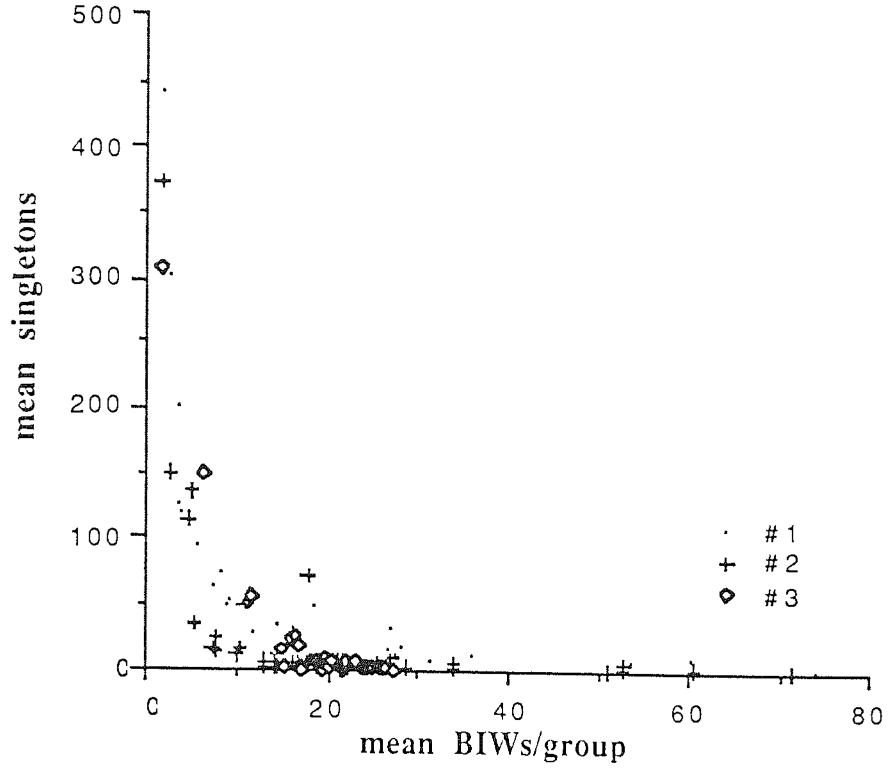
As might be expected, the size of repeating groups of the same body type in the input determines the size of batches of BIWs then painted. If a body type is repeated many times, then, in general, the scheduling algorithm will be able to assign a large batch to a colour. It should be remembered that the algorithm looks for the largest possible batch for any colour (and popular colours will be favoured for large batches over less popular ones, see Chapter 4).



Mean batch size vs. mean BIWs/group (profiles #4 to #6)

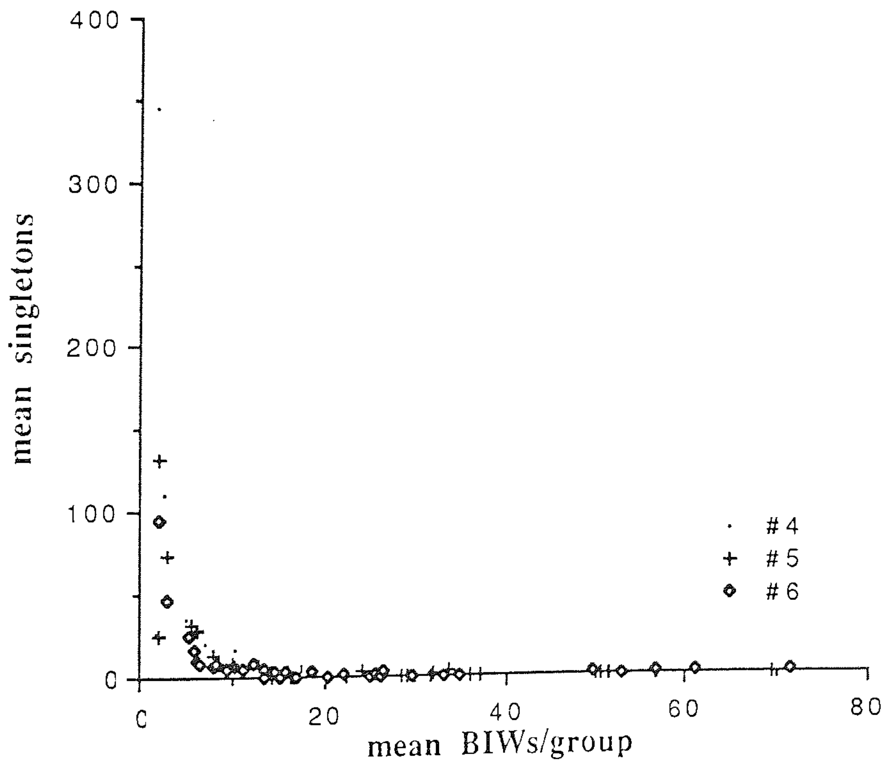
Figure 5.16

In the case of the third profile, the dependence is approximately linear (and more steep) but in general the relationship seems similar across all profiles. This might be expected, because the bias in favour of large batches applies to all profiles. Increasingly random input would be expected to reduce the size of average batches painted, because the algorothm would not be able to schedule large batches. The default option, of painting one BIW at a time - a singleton, would then be expected to become more significant when the input is more random because of the increasing failure to form acceptable batches (i.e. of at least minpb size for the colour). If this is true, then it could explain the sharp fall off in batch size as the input becomes more varied (the average size of groups of the same body type decreasing). The next graphs (Figure 5.17 and 5.18) should illustrate this.



Mean singleton batches vs. mean BIWs/group (profiles #1 to #3)

Figure 5.17

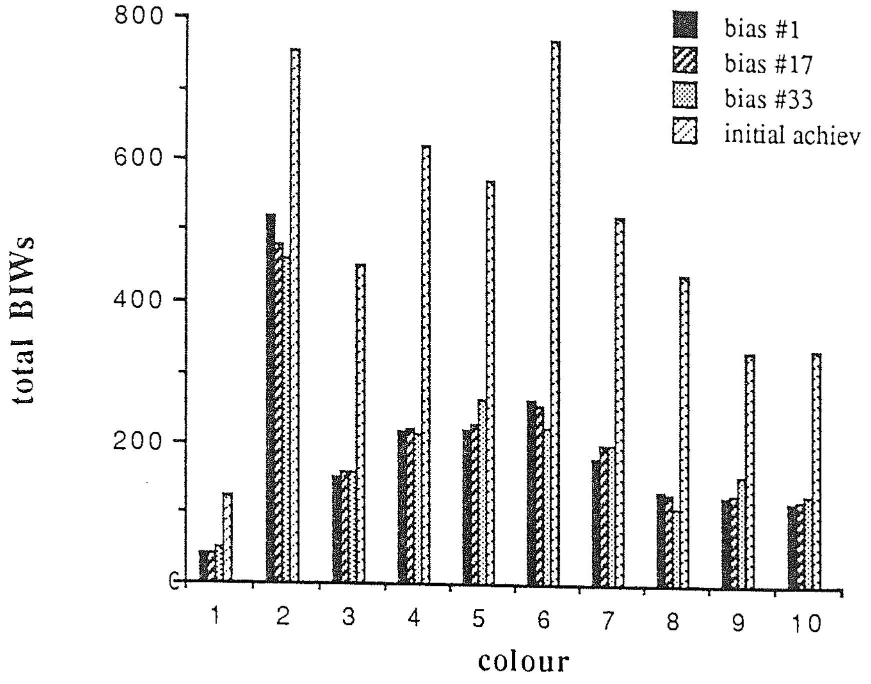


Mean singleton batches vs. mean BIWs/group (profiles #4 to #6)

Figure 5.18

Figures 5.17 and 5.18 indicate an apparently exponential decay relationship between singletons produced and randomness of input. In profile #2 there is a point at approximately 18 BIWs/group which does not fit this well, but as mentioned before, the values for the ordinal are generated by a method which can produce similar (mean) values from different test cases (see the comments, above, on the generation of test sequences.)

The above graphs show randomness (as perceived by the mean size of groups of BIWs in the sequence - randomness increasing along the ordinal toward the origin) having an effect on the batch size which, under these circumstances, could be significant. The intention of batching, as discussed in Chapter 4, is to ensure that not too many bodies of the same type are in sequence after painting. It has been suggested (Istel plc 1985) that a certain batch size is desirable, e.g. 20 bodies. Given the effect of randomness, it seems that the scheduler would produce the desired result for a limited range of input only. If this is representative of working conditions, the scheduler will respond to any significant deviation in sizes of groups by producing a wide range of batch sizes of painted bodies. This would then mean that the intention to produce a homogeneous 'batch distribution' would be thwarted. Thus, the scheduling response could not exhibit the flexibility necessary to give desired production given an input of varying randomness. One type of flexibility that might be desirable is the tolerance of occasionally very random input or input which is not very constant in its degree of randomness.



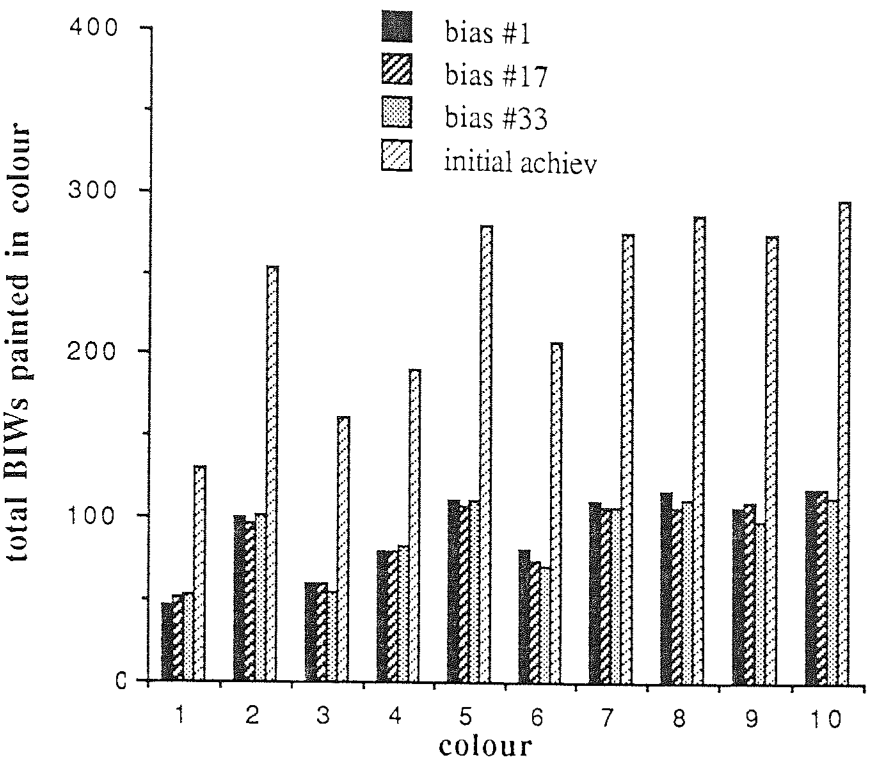
profile #1: variation of achievement with body distribution across colours

Figure 5.19

Deviation from even achievement proportion occurs across colours. This is shown in Figures 5.19, which shows order profile #1 (the 10 × 10 matrix) and 5.20a, 5.20b (profile #6, 20 × 20 matrix). Different colours can be compared with each other - the initial achievement, 50% of requirement, is shown for all colours, with the actual achievements for three bias distributions, representing a range of randomness. Some colours are produced in proportions quite different from their relative abundance in orders (requirement). The total bodies painted in the colour are shown *versus* the colour.

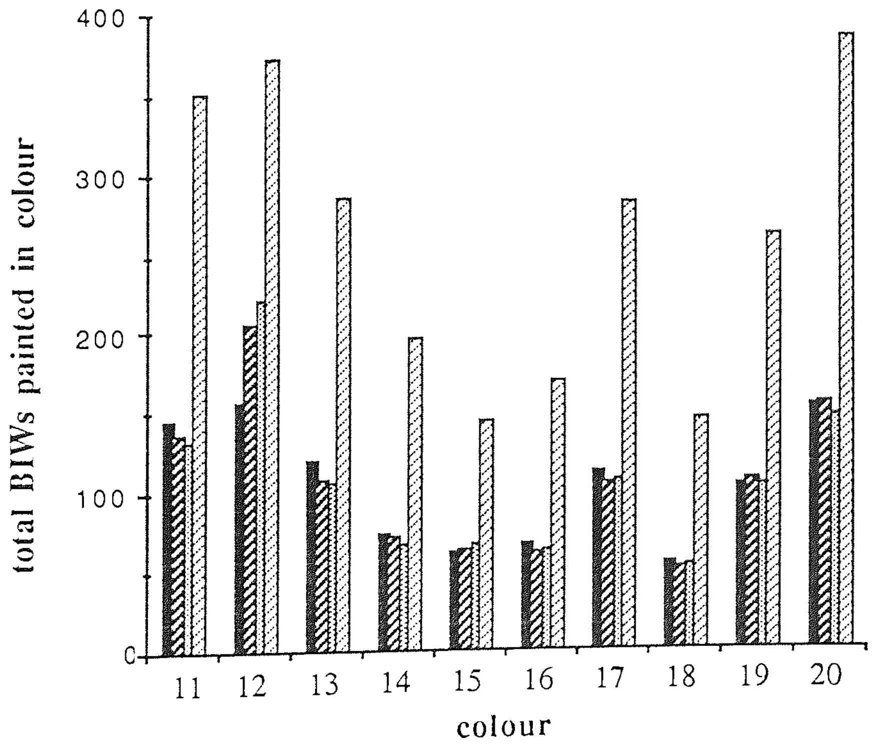
The qualitative effect is that the algorithm does not produce even (ideal) production across colours (e.g. colour 2 in profile #1 - Figure 5.19 - is relatively over produced compared with others) and the effect depends somewhat on the randomness of the input. A further effect is that the degree to which the production is uneven between colours (varying to a different amount with randomness) depends on the colour.

Colour 2 of profile #1 - Figure 5.19 - is most over produced for bias #1 and least for bias #33: for colours 6 and 8 in this test, this is the same relative order of achievements for the biases, whereas the ordering is different for the other colours. The results for deviations in behaviour with respect to mean body group size do not give such clear indications as for the above cases. The absolute values of deviations are large for all parameters compared (see Figures 5.21a to 5.23 inclusive). This is, presumably, due to the fact that the algorithm does not discriminate between body/colour combinations on the basis of achievement. Any balance in the relative proportion of combinations achieved is due to the fact that the batching limits how many bodies are allocated at a time. There is a limit to the potential for a body type to be consistently painted in the same colour (even if it is repeated many times, consecutively, in the sequence).



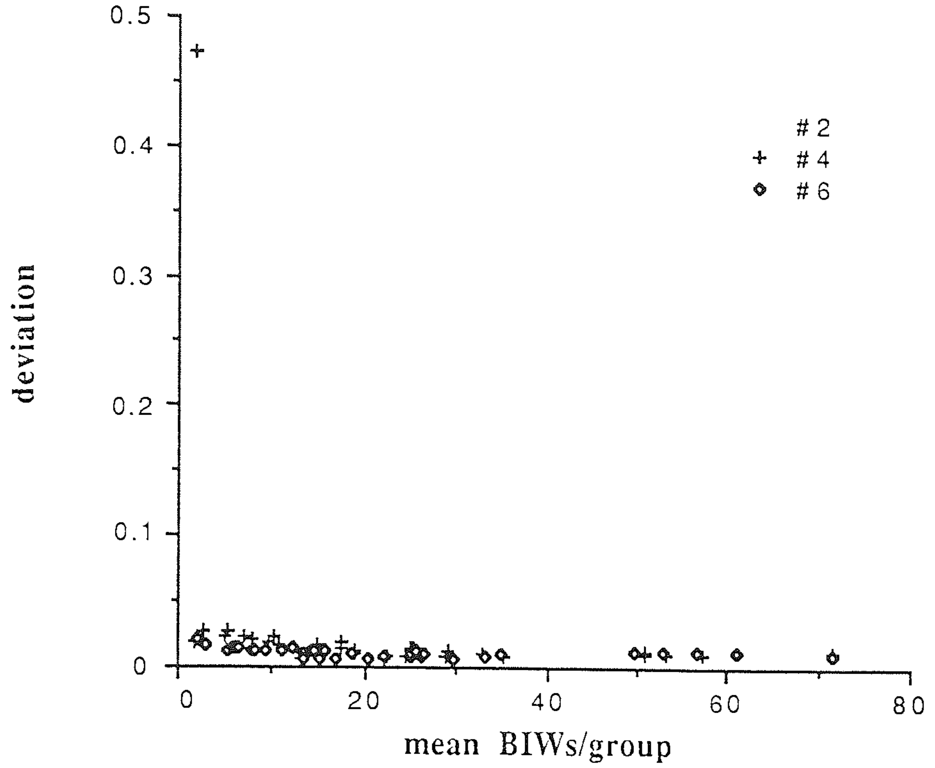
profile #6: variation of achievement with body distribution across colours (part)

Figure 5.20a



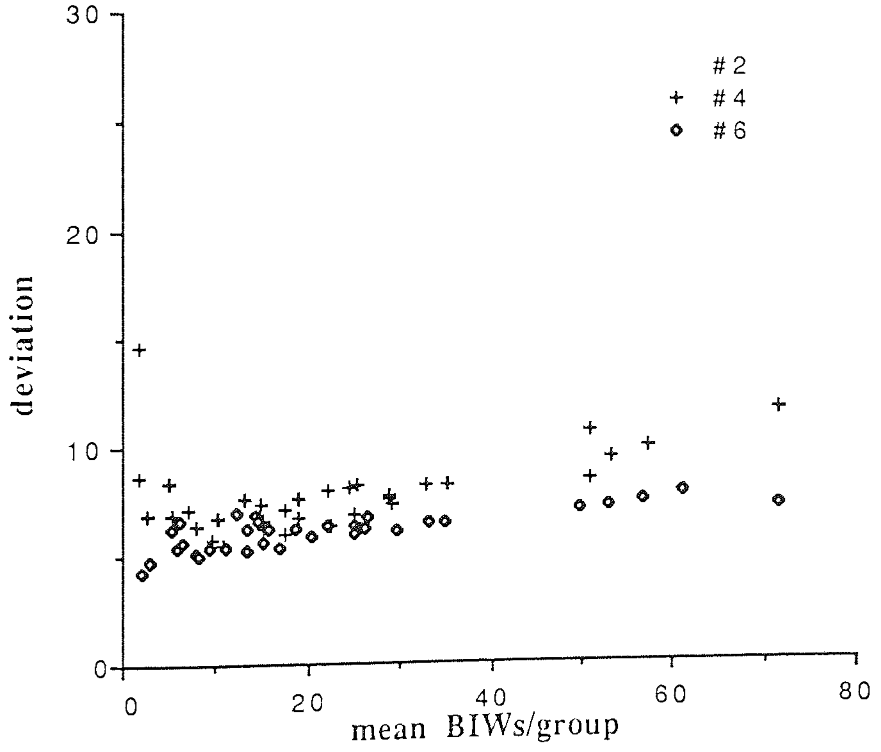
profile #6: variation of achievement with body distribution across colours (rest)

Figure 5.20b



Deviation of mean achievement for colours vs. mean BIWs/group

Figure 5.21a



Deviation of mean achievement for bodies vs. mean BIWs/group

Figure 5.21b

The colour deviations (Figure 5.21a) reflect differences in relative production (and therefore balance) of colours. The more uneven production is, the higher the deviation should be. The results show that production is relatively unbalanced (because the deviation should, ideally, be nil). The effect is apparently greater for profiles with less choice of colours (i.e. #2 > #4 > #6). This is probably because colours with more than usually uneven production in one colour will be more significant among twelve than among twenty. The absolute sizes of the deviations appear to be small. They are, however, significant in production terms.

In all experiments, a fixed total of 2000 BIWs were processed. This means that the total achievement proportion (as calculated) is 80% of the initial outstanding requirement expected average (i.e. (10000 – 2000) / 10000). Assuming a normal distribution, it would be expected that 95% of the colours would have an achievement proportion within ± 2 standard deviations of the overall mean. The overall mean, of actual production, is fixed at 20% of initial outstanding requirement (i.e. 2000 / 10000). Taking a mean batch size of (approximately) what might be desired in the factory (i.e. ~20) such as bias #28 for profile #3 (Figure 5.11) where the mean group size in the input (19.8) was 'averagely' random, the deviation observed was 0.027. The variation between colours could be in the range 0.173 (i.e. 0.200 – 0.027) to 0.227. This would mean, given a total of 2000 BIWs processed, that the 'disproportionate production' across all colours might be ± 108 bodies. Considering the importance placed on even production (e.g. for storage, parts build etc.) these bodies might upset the overall production schedule quite significantly. The test represents perhaps a day's production, so, on a daily basis, some 5% of production might well not go on as planned. For high-cost items such as cars, this level of unscheduled production would almost certainly be significant in terms of cost. Thus, at modest levels of randomness of input, it seems likely that the algorithm's behaviour might be less optimising than desired.

For body/colour production, as might be expected, the tendency to produce small batches with random input allows fewer BIWs to be painted before the body type changes. Each time a colour is selected, fewer BIWs are allocated (to it) and the proportion of each body type produced should be better distributed across all colours. As randomness decreases (left to right, in Figure 5.21b) the distribution is generally less balanced and the deviation increases. The deviation is greater when less colours are available (there being fewer body/colour combinations and more constraint on choice). Qualitatively, comparison of the colour with body/colour (deviation) results shows that the deviation of body/colour combinations is relatively high, as might be expected, and implies that the algorithm does not balance production with respect to this criterion under any degree of randomness of input. In contrast, the colour results show that very random input vastly increases the deviation (typically, a factor of perhaps 15 times from the least to most deviant). In 'ordinary' operation, very random input is not significant but must be considered because it is a way of colour-batching unusual body/colour combinations. (The idea is that such orders can be 'held back' until a large total requirement is collected, then a sequence containing all the ones and twos of these BIW types is put on the line and processed in one go. The input, by the terms measured in the above tests, is very random. The results suggest that batching of such input would not be very successful. A high proportion of singletons would be expected, with concomitant degree of inferior paint quality.)

The overall results are that:

i. randomness produces increasingly uneven production;

ii. colours are affected differentially in the extent that effect i. is present;

iii. 'ideal' (as above) production is not maintained by the scheduler in all cases;

iv. the non-ideal behaviour is greatest at extremes of randomness;

v. limited flexibility with sensitivity to randomness is a feature of the algorithm.

Chapter 6 A knowledge-based toolkit for flexible manufacturing

This chapter sets out details of the programs that were written to emulate the algorithm examined in the previous chapter. The basis of a technique which could allow improvement of the application program (for scheduling) along the lines mentioned there (and in Chapter 3) is supported by the utilities described here. Extensions to facilities available in *C* have been written to facilitate the definition of data classes and their attached functions to allow a more object-based programming style. These extensions and the applications written with them form the basis for a knowledge-based representation scheme which is intended to be a model for solving the difficult problems associated with 'intelligent' process control for CIM.

6.1 The 'objective' toolkit and object-based environments

The toolkit provides a means of supplying 'conventional' programs, written in *C*, with an object-centred environment contained within them. This statement requires some clarification.

The Smalltalk model (i.e. Smalltalk-80) is probably the most usual interpretation of a 'true' object-based environment. In this paradigm, an editor (browser), language translator and run-time system provide an integrated system for developing, testing and running programs. Smalltalk provides a 'virtual machine' for the user's code (which is interpreted at run-time), so, in a sense, all activities, including editing of objects, are a manifestation of this virtual machine.

A rather different view of this subject is held by by such workers as Cox (1985) and Schmucker (1986) who have extended base languages (*C* and Pascal, respectively) with an object-based declaration system into 'hybrids' such as Objective-C and Object-Pascal. In this type of environment, all object code is compiled (rather than interpreted) and then executed at the behest of conventional code in the base language. In this case, the job of the conventional code would be to provide some of the facilities that the Smalltalk virtual machine does, i.e activation/passivation of objects, and an entry point for invocation of the objects' methods. This also allows use of 'foreign' (non object-based) code as required - so it would be possible to build an application with one part in the base language and another which had been written in the objective style. The use of the word 'hybrid' to describe such languages arises from their mixed (object- and non object-based) style.

The above comments serve to highlight some differences between implementational styles for programming with objects, rather than illustrating how objects themselves are used. If specially created object environments like Smalltalk are at one end of the 'implementational spectrum' with the hybrid languages at the other, the work described in this Chapter (vide infra) is closer to the Smalltalk model in the way that objects function, but has, perforce, been implemented in a style somewhat closer the other extreme (relying on hybridisation techniques applied to a standard *C*). It can be claimed, however, that the design of the enhanced *C* system would allow, following further development, replacement of hybrid features with an integrated programming and run-time system (i.e. a Smalltalk-like environment). This contention is supported by my use of a special purpose definitional tool for objects, as opposed to the hybrid languages' reliance on a text editor and the declarative facilities of the compilers of the base languages to produce definitions of objects.

The work described below uses the hybridisation techniques of having (base language) start-up and initialisation code and then having access to objects from ordinary code. It differs, however, in how objects are declared and handled. Smalltalk is described as object-oriented (Goldberg and Robson 1986), as are object-based languages derived from like *C* or Pascal. I shall not use this term to describe the object-based system below. A more appropriate term might be 'object-centred'. This is because of the slight but important differences in the way objects are declared and treated in this system. The term object-centred implies that objects are not statically defined by a frame-like template but have, potentially, dynamic properties. The nature of the objective paradigm will be discussed in comparison of the toolkit developed for manufacturing applications with the above implementational spectrum.

6.1.1 Structure of an object-based system

A summary of the main features of object-based systems might be given as follows:

i. The constituent code and data supplying the behaviour (functionality) of the system are logically associated, in a structural sense.

ii. The nature of this association is that code and the data it acts on are formally segregated into separate entities whose control is specified individually. Each entity presents an 'interface' to the rest of the environment, through the medium of which, access is allowed to its functionality;

iii. Access to these entities (objects) is via specialised components, called methods, which define and form the interface.

iv. This access is in the form of requests to the object to perform some action. Such requests are dealt with by a global 'postage' service, called a messager, which decides what the message means and which particular type of object is being addressed. If the message is valid in this respect, the messager will attempt to find the activity which is required, and activate the method which can effect the action requested. In this way the requester, or 'consumer', of actions does not see how they take place: the 'supplier' (Cox 1985) is hidden.

v. The formal 'encapsulation' of data with code defines the object. Objects are then assumed to have definite characteristics and behaviour, internally specified and independent of the context of invocation, and sensitive only to invoking messages. In this sense, an object should be re-usable as a library item, freely transportable to a different application where it will still display the same abilities as before - a sort of 'context-free' entity.

vi. An additional characteristic, generally associated with objects, is of a taxonomic relationship between objects. Objects are represented as having a family-tree, where adjacent nodes relate to each other as parent-child or sibling-sibling. A child inherits the properties of its parents, i.e. methods and access to some data. All methods defined by a parent are, by default, available to the children, unless specifically re-defined.

vii. The taxonomic nature of the tree structure means that a child is an example of its parent's type, but will also be a specialisation of the parent: by virtue of the new properties it defines.

There are several other facets that are significant: in the extreme, all entities in a system are viewed as objects (in Smalltalk-80); although the description that defines the properties that an object has is really a template for producing objects and their behaviour. The name 'class' is given in Smalltalk to designate the kind of entity an object is. The individual members of a class (examples) are known as 'instances'. If the template (class) is an object itself, in its own right, then it should be an instance of some underlying class that defines classes. In this connection, the infinite regression of 'class is an instance of class...' is somewhat paradoxical, because classes do not, themselves, do anything in the domain of application, they exist as a way of producing and maintaining items (instances) that do things. This is not, strictly, the case: classes may have their own methods and data which are available for their maintenance and, for example, for creation and initialisation of instances. Another formal structuring of classes and instances is the possession of a set of variables (within instances) which are manipulated by instance methods, and therefore private - as opposed to variables defined for a class which are shared by all members of the class.

In Smalltalk, the variables are, in a sense, pointers to instances of classes. Primitive items like Integers (with the exception of SmallIntegers) are objects belonging to a class. In the object-centred toolkit, it was not necessary to implement variables in this way. In practice, only primitive items like pointers and numbers were used. It would of course be possible to make (instance) variables reference objects of some class. The toolkit is not object-oriented, as Smalltalk is; objects such as instance variables are type-free, an object may reference anything; indeed, it may reference several different objects, depending on which method is referencing them. It should be noted that Objective-C and Object-Pascal use records for class templates with the methods (functions) as pointer fields. Methods are activated by de-referencing the appropriate field of a record. The object-centred toolkit determines how a method is implemented at run-time, in the messager, and the binding of methods' code is done, not at the compile time of the methods themselves, but when the messager is linked. If dynamic loading were to be used, the binding of methods could be delayed until invocation at run-time. Classes declare methods when they are defined. All this does is enable some, as yet to be bound, functionality to the object.

6.1.2 An overview of the toolkit and its usage

The toolkit contains a number of program source files which can be used to support object-centred activities. The two principal tools are editors: the first being used to construct object 'bases'; the second being a prototype knowledge acquisition tool which allows the definition of the nature of the components (classes, methods etc.) and supports the acquisition of object-centred rules (as instances of rule classes or sets). In this latter capacity, the knowledge acquisition tool accepts an English-like statement of production rules governing relations between objects and translates them into the form used internally (see Appendix A6). The object base editor operates in two modes. New bases may be defined or existing ones edited.

When defining a new object base a new user must first decide which classes are to be declared. The methods accessed by these classes and their (implementing, 'effector') functions must also be decided. On invocation, the editor will prompt the user for a name by which the base will be accessed, if the editor command line contains no such indication. (The editor creates a new base with this name, checking first that the name is indeed new. This is to protect accidental over-writing of existing bases.)

Classes are named in turn, followed by the method and effector names. These names must be unique (within their group). The editor then prompts the user for details of each class. The inheritance list (via 'isa' links) is stated. A restriction placed is that a class may not name itself as a parent (which could allow cycles). It may only have named classes in the list. The exact inheritance must be stated. Naming of a parent does not mean that that parent's parent will be examined when tracing inheritance. The list shows exactly the order and extent for each class. A class may be at the present limit of the hierarchy (root). If desired, there may be more than one effective root class and multiple or incomplete inheritances are allowed. After the inheritance list has been stated, the class' methods are named. Child classes are defined after the methods. If there are no such classes the editor marks the class as a 'leaf'.

Having defined the hierarchical position for a class, the instances are now declared. The user must simply state how much space needs to be reserved for an instance and how many instances should initially be allocated. The editor allows the definition and naming of instances by the input of character strings and by special key sequences, storage of binary values, for real numbers etc., entered as text.
With the above stages finished, the data structuring part is complete. The user is now required to declare initial bindings of effector names. The editor (silently) compiles a file, containing these function names, which is used to ensure that the messager utility can access the effectors - without the need for the user to change toolkit components which use the object base.

For each method declared by a class, the editor prompts the user to specify which effector will be used. A table of these (class, method, effector) triples is saved. The editor assists the production of an executable application program for the object base by using the *UNIX make* utility. The user must place details of the sources for the effectors in a suitable file (for *make*-ing) together with the messager, startup and other required toolkit utilities. In any such program an entry point is required. This can access the object-centred part by messaging calls on objects (via the messager).

An already defined object base may be edited by invoking it as the argument to the editor program. Any object may be re-defined, new methods and effectors may be added. A new object-centred program can be made according to the new object definitions.

The knowledge acquisition tool can be used in two ways - as for the object editor. The purpose of the tool is to construct a (meta-)description of the object base already defined, so as to allow definition of instances for its classes. The program loads an object base, defined by the user, into memory. It then loads another (toolkit-supplied) object base. This second object base contains classes that define the syntax of rules, and skeletal classes whose instances are initialised to refer to the classes of the user object base. Thus the (meta-)class 'u–classes' has an instance for each of the user-defined classes. These instances are generated to retain details of the user classes' associations (with methods etc.) In a similar way the second part of the toolkit object base reflects (categorised by its meta-classes) aspects of the knowledge to be acquired. The (meta-)class 'u_constants' contains symbols that are known to be constants for the input of rules. The user may define these and state their internal representation for rule generation. The rule grammar already understood may be extended to include extra symbols, such as 'equals' (a relational operator) which would presumably be defined to generate the '=' symbol in a rule. This allows tailoring of the knowledge tool to the users needs. An expanded version of the user's object base is made from these definitions and saved. It can then be supplied to the tool as the source for knowledge acquisition. This is done by specifying which of the 'u_classes' instances (i.e. the classes of the user object base) is a rule class. Rules may then be defined for the various classes by typing in text.

Parsing of potential rules is guided by the meta-knowledge contained in the meta-description. This is in two forms: firstly, grammar of rules (represented in some of the pre-defined meta-classes and their methods) gives the parser certain expectations of the principal components (i.e. English words like 'then' and their relative positions in the sentence). Knowledge about the associations of the (object base) classes and their methods allows the parser to compare what appear to be references to methods with the object definition. Since rules are sets of messages which may form conditional tests or actions, certain types of words can be expected to appear in specified positions relative to each other. The parser looks at input strings, separating them into words, and tries to categorise them. If words are unknown to the program, sometimes the user may be able to supply a definition. The parser can attempt to identify a suitable categorisation and will accept a definition for the internal representation of a word for new constants, for example.

Rules that have been accepted can be placed into their proper class in the original object base definition by the knowledge tool. This is achieved by translating the parsed output into an internal representation. The text used for statement of rules can be saved in the file used by the rule trace mechanism.

6.1.3 Support of object-centred and knowledge-based programming by the toolkit

The diagram in Figure 6.1 shows the hierarchy of the classes used in the knowledge-based paint scheduler. (The additional, generic, trace facility is not shown, as are the, optional, additional rule classes to express all batching by rules.) The hierarchy of the object-centred part of the program is detailed. The steps by which such a system are assembled, are described more fully later in this Chapter.

In this section, the tools that support the object-centred 'functionalities' are described.

The central part of the toolkit is a utility performing some of the functions associated with the 'browser' of Smalltalk. This creates and modifies objects. The features required for object-centred programming are built by this program. The taxonomy of object classes is created by defining the children and parent of each class. The topmost class (in this case - 'task_manager') is the immediate ancestor of several classes. As a general rule, the methods belonging to this class, defining its abilities, are the most general, and, inherited, may be used by many of its descendant classes. In defining a class, the only additional features that are defined (after the methods) are the initial size (in bytes) a class should have reserved for its instances and the number of instances for which space, in memory, should be reserved when loading the object hierarchy.

Here it is appropriate to give an account of how classes are dealt with at the more concrete level of implementation in *C*. Quite unlike in Objective-C or Object-Pascal, where a record structure defining the class is compiled, the class composition is simply recorded as data in a file. Standard code for loading and dumping class contents uses this data. This standard code (for activation/ passivation) is one of the utilities linked in any object-centred application produced by the toolkit.

The above use of 'data' for holding details of classes does require some special features. The whole class structure, including instances, is held in four files (per application). At run-time, a global variable is used to access heap storage, via the **calloc()** function (*cf.* **new** in standard Pascal), and an array of pointers (one per class) is allocated. These pointers are then used, similarly, to allocate space for the 'class frames' held in heap space. These frames, amongst other items, contain more pointers, one of which is used to allocate an array of pointers which can reference character arrays. Instances are loaded into character arrays (really, strings) in memory.

Tree Structure For Class Objects

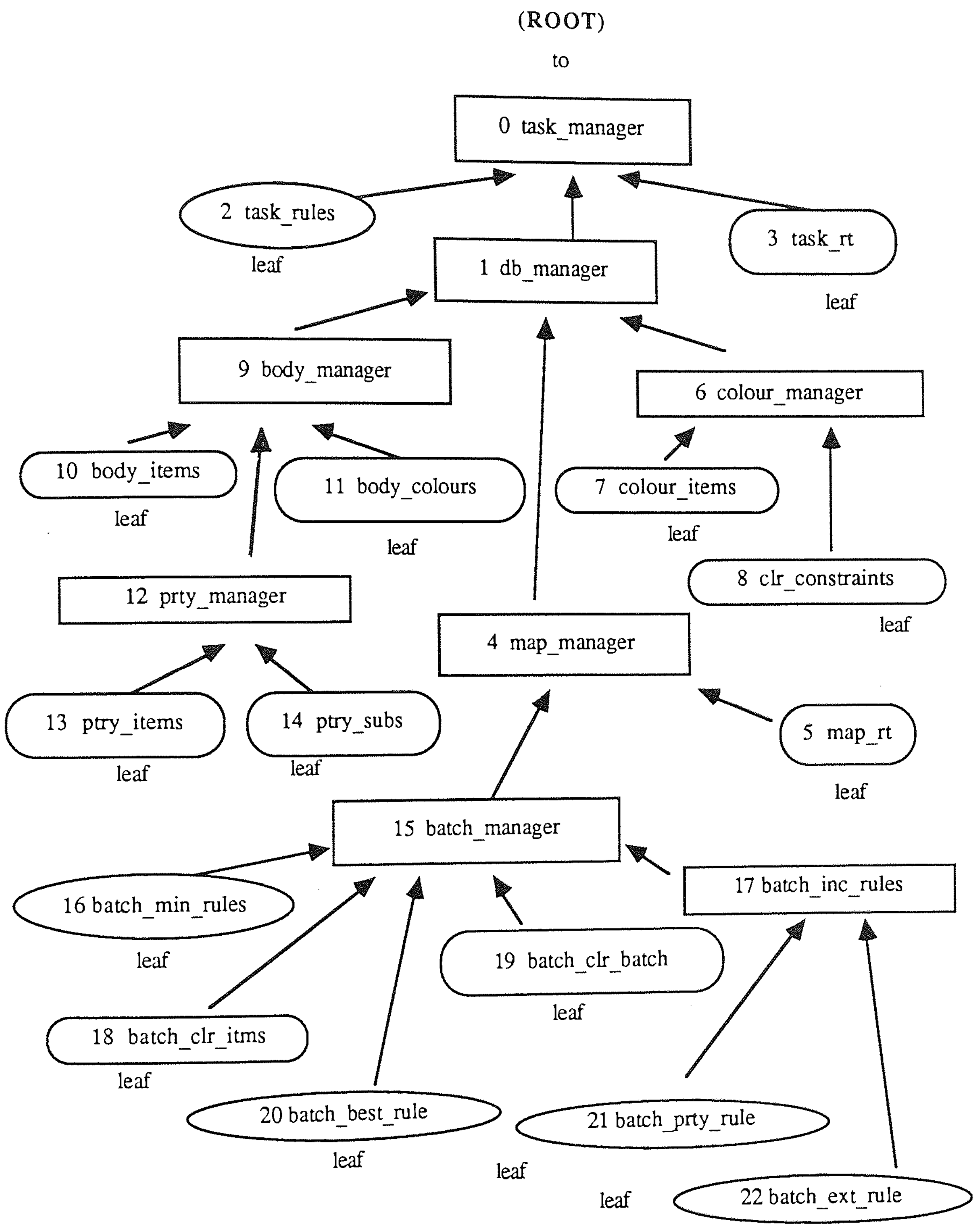


Figure 6.1

The base address of each instance can then be entered into these pointer arrays and the instances are now connected with their respective class frames. Thus, with several levels of indirection, the whole structure can be held in memory. This has the added advantage that (even dynamic) changes to classes can be accommodated without the penalty of re-compiling record structures that describes them. To facilitate loading of the contents of the hierarchy, sufficient space is allocated for the contents of all classes which are then read into memory. The above frame for the hierarchy (the collections of pointers and arrays of pointers) is connected by assigning appropriate values to these pointers, to indirect them into the reserved memory. Note that the structure of these pointer frames is compiled and is a property of the run-time toolkit. It is therefore, constant and independent of the nature of classes defined, providing a standard way of accessing a hierarchy of previously undefined structure. The toolkit provides a framework for the construction of an object-centred hierarchy.

The above use of global variables for the framework is somewhat undesirable - leading to the requirement for methods' source files to declare them, via **#include**-ing standard header files (with the *C* pre-processor), then relying on the compiler and linker to organise this (i.e. as it were, by default, they become external symbols and thus become globally available). This restricts to a small extent the identifiers that all code may use, at the same time as accessing the object-centred system. Such global variables would not be accepted if declared as external symbols in other code. This also means that these system variables are in the scope of all methods, and accessible, though it would be potentially catastrophic to the whole data structure for an inappropriate re-assignment of any such variable. This technique was devised to test the implementability of this type of object structure, but could be readily transformed into a more 'scoping-secure' (but less 'free') system. Discussion of such developments is left to Chapter 8.

6.1.3 Messaging

The messager - contained in one of the source files required for linking into the object-centred application - was written in ordinary *C*. Message interpretation is provided by a function. This function, however, is special in the way that it is compiled and linked. It must be compiled for each application program, rather than simply linked as for the source files containing the code for methods. Its job, as mentioned above, is to interpret messages sent to objects and ensure that the correct method is invoked. In this context, no distinction, *a priori*, is made between class and instance methods and, indeed, the class definitions do not record such distinctions. It is left to the methods themselves to do this - comment on possible improvement in this area is contained in Chapter 8. (Here, again, as the system 'evolved' these differences became apparent and, in practice, most methods developed used some kind of checking to establish the relevance of the message to them.) This was done, originally, to simplify and speed up messaging code since it is a natural bottle-neck of the system. The way the messager decodes the message is to identify two (mandatory) components and, optionally, a third. The first part shows which method, logically the name for the method, is required. In combination with the second part, which must be the identifier of a class, the particular method is identified.

Methods are initially defined as a potential property for a class to possess. The actual code ('effector' function), which 'implements' a method, executes the action implied by the method and, therefore, binds some code to the named potential to do something defined as a property of the class. It is declared (by name) after the whole object hierarchy is stated. (The properties of a class: instances, 'isa' links and children are defined with names of methods; the class states its interface with its environment at definition time; implementational details of the interface are not 'bound'. This is a subtle difference from the use of the term 'method' in Smalltalk. It should be noted that, given a definition of a class possessing methods, the actual implementation may be changed without changing this definition.)

Methods are bound to implementations by stating which effector is to be linked for that class. Many classes may declare a particular method, but their actual implementations in the interface depend on the effectors adduced. The combination of a method and a class name with an effector name completely defines a method. A table of triples of these names is built up and saved. It is from this table that the messager determines which method is appropriate on decoding of a message.

Effector names are 'linked' into the program by a technique depending, to some extent, on the convenience, in *C*, of invoking the pre-processor. So far, the above shows how effectors are notified to the system. The sole requirement for a function to be used, if an external symbol and thus not defined in the source file being compiled, is that it be available at link time, if it is of the (default) type returning integer. The compiler requires further elucidation if the type returned by the function does not resolve to integer. (Some compilers will allow sub-types of integer to be treated as integer for this purpose.) Forward declarations provide a way of doing this. This is important because effectors are always defined as returning **char *** (i.e string). A text-processing utility collects the list of the effector names required and enters them in a special header file. This file can then be used to define functions to the compiler. The messager source file contains an **#include** directive to the compiler to load this header file, allowing these functions to be referenced in the messager code. This is described more fully here after discussion of the interpretation of messages.

Identification of an appropriate effector by the messager is straightforward. The named methods in the 'messaging table' are listed in order, paired with their associated classes. If all the pairs of the method named in the message do not match for the class name as well, the messager examines the ancestors of the addressed class to identify a match for the class. In case of failure, a standard error message (the string "BADMESSAGE") with an accompanying error warning on the standard output device, is returned to the sender of the message. Assuming that a pair (method name/class name) is identified, the name of the effector needed will be the third member of the triple in the table for the matched method/class pair. A list of pairs of effector names and pointers to the code body of the function is kept. The way by which the identified code is now called is to de-reference the pointer. This is possible because of the nature of the function pointer as a variable with a value - which, however, must be known at link time.

In addition to the declaration of the effector names as of the appropriate type in the header file constructed by the system, a special global variable is allocated in this file. This variable is an array of pointers to hold the addresses of functions (i.e pointers to function returning **char ***). The size of this table must be declared to the compiler, and so another header file is **#include**-d in the messager source. The sole entry of this file is a definition of a constant for the size of the reference table for effectors. (The reason for the second header file being separate from the first is to allow re-definition of the table, independently of re-writing the first header.) Using this table, the list of effectors is declared, again, this time to initialise the values of the array - the name of a function, i.e. as a pointer to it, being taken as its address. The system, having written the header files, can now compile the messager so that it can reference the effectors that were only described by name. Note that the effectors themselves will be in source files, and may be separately compiled from the messager. The reason that the effectors will be accessible, when a complete program is assembled, is that the compiler will leave references to the effectors, in the messager, as unsatisfied external symbols which, however, are of a known (not default) type. The linker will be able to satisfy the function references in the messager by linking the previously 'anonymous' effectors now provided in the files for the method code.

All effectors should be provided in the source files as of type **Str** (defined for pointer to char - **char ***). This allows effectors to return a standard type, but does not restrict the contents of a return. Any item larger than **Str** may be accessed, by convention, by saving it in a static array of char and returning the array address. This allows effectively type-free returns from effectors.

Assuming that three effectors are required for methods, named a, b, and c, the text-processing enters the following definitions in the first header, say **header1.h**:

```
Str a();
Str b();
Str c();
```
(the parentheses indicate a function).

Three effectors are required, so the second header, e.g. **header2.h,** defines the table size thus:

```
#define PTRTABLE 3
```

In the first header, the table (**tab**, assuming that **tab** is defined as being of the type pointer to function returning **Str**) is declared as: **tab[PTRTABLE]**, followed by a list of initial values for the effector address, thus:

```
tab[PTRTABLE ] = { a, b, c };
```

This leaves the messager to access these definitions. This is done without re-writing the source; the file just needs to be compiled with the definitions that the system has just entered in the header files, **header1.h** and **header2.h**. In the messager source, then, the following two lines are all that are needed:

```
#include "header1.h"
#include "header2.h"
```

On linking the values of the effector addresses are supplied to this, initialised, table, giving run-time access to effectors to the messager via this global variable (**tab**).

The messager can now call effectors by referencing the pointer in the table. Assume that a message has been sent and the messager has found that the effector **b** is defined for the method referenced. In normal circumstances the function is called by stating its name (**b**) and making it part of a statement (e.g. **name = b(args)**, where **args** are the actual parameters and **name** a suitable variable of type **Str**). In this case, of course, the messager does not know the actual effector when compiled: it is a reference from its table to an address. It does know which table entry is indicated, say, in general, **x**. Thus, **tab[x]** points to the correct function address. In the case of **b**, this value is **1** (the second element). In C, the contents of what a pointer references are indicated by prefixing the variable with a star (*), and in the case of a function reference, a call by indirection is allowed by showing the expression is a function (with parentheses) which can then be put in the place of a 'normal' function name. To call **b**, a call of (***b**) () would be required (the left hand parentheses force prior evaluation of the * and **b** together). In the case stated, this would be (*(**tab[1]**)) () (again, parentheses show association of items for evaluation). Generally, the de-referencing statement in the messager to access any effector, will then be of the form '(*(**tab[x]**)) ()'. This can be seen to be independent of which set of methods is used: the differences are dealt with by editing the header files (i.e. **header1.h**, **header2.h**) and re-compiling the messager. Re-compiling is arranged by the system *(vide infra).*

Several other features of the messager are significant. The effectors return a value, by definition (**Str**) and so the statement calling the effector must appear with an assignment (strictly the compiler expects to see an 'lvalue' in the expression). If the assignment is to **name**, in the above example, the equivalent statement would be **name** = **b**(). A global variable (**Eval**) is provided for this purpose, so the convention is to send a message that would result in **Eval** = **b**(). By use of the pre-processor facilities, a distinctive form for the message was provided (enclosed in (* and *)). The message itself appears in the object-centred code in a prescribed form which can be described as '(* MethodName ClassName Selector String *)'. The nature of MethodName and ClassName have been described above: they are the first two components that allow the messager to find the method required. The convention for SelectorString is that it is of variable of type **Str**, i.e. it points to an array of characters, thus complex items can be 'passed' as part of the message. (The pre-processor re-arranges the message form in (* and *) to:

'Eval = msg(MethodName,ClassName,SelectorString);'. **'msg'** is the messager function which receives SelectorString as an actual parameter and supplies it to the effector selected: the effect is that the de-referencing call now contains SelectorString.)

An additional standardisation is enforced by the messager. The actual call to which the effector must respond is of the form: **'(*(tab[x])) (AddressedClass, RespondingClass,SelectorString);'**. The SelectorString is used to identify that an instance (method) is being addressed, for example. AddressedClass and RespondingClass are the original target of the message and the class that supplies the method, i.e. these are the same unless the method was inherited. (The identification of the actual classes involved was useful to start with and its existence is connected with the need for **self** and **super** mechanisms, as in Smalltalk. As will be described in Chapter 8, there are, probably, better ways to achieve these ends, and not 'open up' the classes by making their identities evident.) The messager responds to the sender by returning the result of the effector call.

Classes and methods are named as they are defined; when used in messages, they are, effectively, 'named' also. The form of the name is not the literal string, however, but a reference to it. A class, for example, is thus 'named' from its position in the class hierarchy (as are its instances from within the class). This may seem unattractive - but the use of a string as an identifier designating the class would (just as well) force binding of it into the code (declared at the compilation time of any messages). The advantage of using this indexing technique lies in the simplification of resolving class (and method and effector) references. It would take longer to use literal names, because the strings would then require extra processing to distinguish their meaning, which could slow down messaging (but see Chapter 8). This was intended to make the messager as efficient as possible, but it is noted that the appearance of the method code could be more readable if classes etc. were named by identifying strings. There is a difficulty in that the names of objects are stated in a different place and are dynamic, re-nameable. (A technique that could be used might be to compile reference tables of identifiers used in methods and associate them with corresponding identifiers defined elsewhere.) The index also provides an absolute reference to a class, method or effector (or instance, when associated with a specific class). Being bound to an index, these references can be resolved directly by the messager. Instances are identified in the selector string when referenced and named from their ordinal position of declaration (in their class). Distinction between class and instance methods relies on the method (and not the messager) finding the relevant instance's reference in the string argument (i.e. passed on to the effector). It is a convention that instances are denoted by the first resolvable item in this argument.

It is then necessary that a method sending a message should know the identity of the class to be addressed. If the method requests that some action take place, involving the sending class, it is normal to supply the identity of the sending class to the method via the selector string in the message. This allows a method to respond effectively to an 'anonymous' sender and, to some extent, allows classes to be more 'independent' of each other. (It is fairly easy to reduce this level of encryption; see Chapter 8. The style employed was used to implement a simple interfacing convention.)

6.1.4 Definition of instances

A simple means was used for the declaration of instances. All that is required is to state what amount of storage should be reserved for each instance and how many such items are to be loaded initially. The instance is (formally) unstructured, *a priori*, so it exists as though it were just space in memory. The user can define the contents of any notionally internal structures within instances by initialising them, as required. The program allows entry of text or binary items, or a combination of both. That is for any numeric item, such as a double length real number (**double**) or even a pointer field, the byte-wise values can be entered. The instance editor buffers all characters input and, following indication that a set of characters designates a non-text item, converts the string input into a suitable binary representation, according to type. Such binary numbers are eventually saved, 'passivated', to use Cox's term, into the file for instances.

6.1.5 Review of definitions

Once defined, an object hierarchy may be edited, extended or reviewed. The means of removing a defined class is to remove all references to it, as it is impracticable to remove it altogether (because of the addressing scheme) and simpler to re-define it as having no properties. Thus it wil have no children and no parent. Any former children must be edited to give them a new parent (e.g. the former grandparent). The class ceases to play a part in the structure, whilst avoiding complex changes to the hierarchy.

For review, the object editor presents any of the defined features, at request. Instances containing binary items are displayed in a special format. This format is intended to allow debugging of complex ('low-level') items on a byte-by-byte basis (essential for checking correct assignment and, particularly, alignment of numerical items). It should be remembered that a full, factory-wide, implementation of a, possibly real-time, process for FMS, using the object-centred paradigm would be concerned, at a low-level, with byte manipulation. (One extension, for users' convenience, might be to allow individual setting of bit fields).

6.1.6 Linking of a complete system

So far, the definition of the data and reference of code has been alluded to, without discussion of code composition. The code used is essentially of the base language *C*. The object-centred system has not been extended to providing special facilities for writing methods. The necessary messaging conventions are supported by the re-writing of header files and then compilation of the messager. The system assumes that all the code required for the effectors is available. This code is produced in the normal way for *C* programs and compiled, but will, of course, be linked to the object-centred application via the toolkit. Source files for effectors must have several lines in their text to enable the object-centred capabilities. These lines must specify (**#include**) necessary header files and define the effectors that are named to the toolkit for methods. Use is made of the *UNIX* utility '**make**' to assemble the correct executable program. This utility uses a 'makefile' to specify how a program is composed. The source files detailed in it are accessed to provide the desired output. It is assumed that the user knows how to write a makefile, which can be done by editing a skeleton example. The dependencies of source files on headers, stated in a makefile, cause the compilation of any source that uses a header more recent than it - thus the messager source will be compiled, following the re-writing of the definitions as stated above, i.e. **header1.h** and **header2.h**. The makefile needs to list the sources for the effectors. If these require compilation, the utility will do this and then link the object-centred application program. The application program must also include calls on the standard routines to initialise the object framework and load the objects (activation), and for passivation. Start-up code is provided in a file, as are several other utilities, which must be included in the makefile. It is for reasons of this design that the system is described as a toolkit. It is intended to provide flexible support for object-centred programming, rather than create, at this stage, a totally 'enclosed' environment.

Calls on the object-centred system contained in the application merely need to use a standard form of message. It is recommended that this be restricted to one entry point for each object-centred activity required, and that the entry point should be a method of the highest level object. The non-object code should not really access the object-centred part other than by its methods. (In Chapter 8, a mechanism to ensure such orderly interfacing is described.)

6.1.7 Tracing messages

To provide object programming without a mechanism for debugging would be inefficient. Several mechanisms are available to support useful tracing of activity. Firstly, it is possible to include a standard interrupt-handler which, if enabled, allows the re-defintion of the Control-C (^C) character, typed from the keyboad, as an entry to this routine. ^C (in BSD 4.2 *UNIX*) causes a software signal ('SIGINT') to be originated in the terminal device and sent to the attached process (i.e. the application program). SIGINT normally kills the process, but can be trapped by the program. If the toolkit's interrupt-handler is included in the program, all ^C characters are trapped. If tracing has not been enabled, the interupt-handler only allows ^C to have its usual effect if the user indicates that he wants to exit from the program - otherwise control is returned to the point where ^C was detected. Where tracing is enabled, ^C activates the tracing review mechanism. When a user requests tracing, all the message traffic is echoed to the screen. This means that the message and its meaning (as interpreted by the messager) are displayed; which method is invoked, if any, and the result of the call on the effector. If this information is not sufficient for the user to understand what is happening, ^C permits access to other facilities. A menu of possible actions is displayed. These allow examination of the contents of the memory; to check the state of the contents of objects, or the definitions given. It is also possible to exit from the trace (where it was entered). Each tracing step through the messager is regarded as belonging to particular level. The first message is at level nought, the second at one and so on. As the depth of function calls (through the messager) unwinds as it returns, the level decreases by one. In certain circumstances the call can be 'unwound' prematurely, before the effector is accessed. The messager then returns a 'dummy' value for the effector which was bypassed. The level is reduced and a 'non-fail' value results, as if the effector had been invoked and returned without error. A further option is to jump back to the previous level above the last call of the messager, i.e. omit the current message and resume at the next highest level.

The above means of avoiding all the effects of message interpretation can be used to arrange that the desired action of an effector, which was not called, occurs. This is done by using the object editor in the trace review mechanism. The contents of objects can be reviewed and then set to the required value (that the action of the avoided effector would have set them to).

6.1.8 Support of knowledge-based programming

Two major facilities associated with knowledge-based approaches are provided in the toolkit. Rule-based programming is often thought of as being central to AI. It is really the declarative aspects of rules and object definitions that are catered for in the toolkit. An object definition is, in a proper sense, a declarative means of defining a model (of the domain). If the model of the application reflects knowledge of the domain, then the objects declare such knowledge fairly explicitly. The procedural aspects of 'knowledge expression', methods in this paradigm, are usually simple enough for them to represent easily, comprehensible, logical actions on data. With a trace mechanism allowing the (gross) effects of a method to be explicated, this knowledge is, to a great extent, very explicitly represented.

Another aspect of knowledge is that of predication of ~~of~~ one item on another, e.g. 'if a then b'. Incorporation of formal rules into the object-centred system relies on the messaging convention. Rules, such as 'if a then b', are represented as instances of a class (of rules), which can be typed as strings, at the appropriate places in the list of the instances. In principle, rules are not supposed to be ordered, but in this case, they clearly must be because the list of instances is ordered by the system. It is up to the writer of rules to avoid (less 'declarative') effects dependent on rule ordering. For expression of rules, two interpreters have been provided in the toolkit. Rules must follow a given syntax for these interpreters. The instances of a rule class store the rules in a compressed form (for direct evaluation by the interpreters). A utility, *vide infra*, is available to assist the user in writing a readable and more English-style form of the rule, which it then translates into this internal form. This has the advantage of pre-digesting rules, so that their interpretation can be made faster.

In a class of rules, all instances are expected to be rules and will be interpreted as such. The class denotes a rule set, in effect. The 'traditional' form for a rule: a set of conditions, followed by a set of actions that can be asserted if the condition is valid, is observed. The syntax for rules does not preclude their use in a backward-chaining mode but, in the application developed, it was found to be appropriate to base expression on the forward-chaining basis of evaluating a condition and then asserting the action part.

The conditions and actions of a rule are a set of expressions, the conditions being composed of tests. The test corresponds to the sending of a message, which would have a result value (from the effector, or on error, messager) which is taken as its value (for a logical expression) or is compared with a constant value in the rule. If this test is logically true then the condition (or part thereof) is true. In the same way, actions are sets of expressions from which messages are derived. The activation of a message may result in some change to the object image in memory and is therefore imperative. (Conditions are *not* constrained to causing only non-destructive effects in objects, however.) The test must reference a method, which it does by stating the name of the method and addressed class and providing an optional parameter in the form of a string. The optional parameter then becomes the selector string for the message.

The interpreter is called on the class of rules to be evaluated. It parses rules into their constituent parts and then sends messages to objects for their expression. The interpreter may be instructed to operate in several ways. If, for example, it is asked to fire rules, it will start at the first rule in the class and attempt to evaluate its condition(s). If the conditional part evaluates to true, then the action part is invoked; this will be parsed in a similar manner and any messages arising are sent. More than one action is permitted to exert the effect of a rule. Rules may also be evaluated without firing; the number of (valid) rules consulted may be restricted to one or the whole set may be expressed.

The use of rules composed of potential messages means that they have the same control mechanism for mediation of (rule) expression as method invocation (i.e. messaging) in the rest of the object-centred system. An interpreter is an effector expressing a method. To invoke rule expression, a message is sent to the rule class (which is, then, sensitive to such interpretation). The type of rule expression implemented in the interpreters is of a single-level. Depth-first search, for example, is not provided in the toolkit; the interpreter just evaluates the rule (or set of rules) once on invocation. More complicated strategies for rule evaluation could be achieved by setting up another method to manage repeated invocations of the basic-level interpreter expressing the individual rules. There are several points to note in this context. All rules are of a single kind (with one possible extension) which does not state how it is to be interpreted (apart from having a defined syntax). It is possible to chain rules by stating (in a message) that the result of a test in one rule is obtained by interpretation of other rules. This technique seems close in intention to that of James and Frederick (1985). The significance of this is that in the FMS environment, which prompted this work, the knowledge was of the kind that establishes individual facts, that is, does not involve chaining of rules of the same (class) type - or, establishes facts in one context that depend on rules in other contexts. This mechanism controls relevance of rules and provides a contextual means for their appropriate expression.

The stated syntax of the rules implies that all rules are of this simple type. In effect, there is another type of rule which is the same, except that it contains an extra part, prefixed to the condition(s). The prefix specifies that the rule applies to all members of the class indicated, i.e. 'quantified over that class'. This allows rules to be generalised, e.g. ones that apply to colours (as a class) are stated as 'for all colours ...'. This syntactic device allows the rule to be applied over the instances of the class. A (slightly) different effector, invoked as a different method from the usual interpreter, will then pick up the quantified class and attempt to apply such rules once for every instance of the class (according to the mode of invocation). This can be useful, for example, when special attention must be given to particular instances. If 'colours' are referenced and the instance 'green' is important, a rule sensitive to 'green' can be used, thus: 'for all colours, if instance of colour is green then ....'; the action of this rule can then be restricted to one instance out of the whole class, and different rules in the same class can select for other colour instances. By extension of this mechanism, a quantified interpretation involving rules that specify a further level of interpretation, which is not quantified over a class, may pick up the original class and use the implied identification to allow appropriate expression: the quantified class is still visible to the second level of rules. These two intepreters are provided as effectors in a toolkit 'rule' source file.

6.1.9 Rule tracing

Special code is included in the toolkit to allow the interpreters to record rule expression. The mechanism for this is somewhat similar to that for message tracing. If this is enabled, the interpreters list details of the expression of rule classses that are sent intepreting messages. These are output to a file, for later examination. The identity of the rule class, which instances were examined and the outcome, what mode of interpretation and the result of the interpretation are all listed. After a run, the user may examine the trace, step by step, in either direction (see Chapter 8). This utility allows reasonable examination of the rules and should facilitate further understanding of how rules interact.

6.1.10 A test harness

A program to build a test harness for objects (**MAKETEST**) allows the source files for the methods of an object-centred definition to be linked with code for a test 'rig'. The resulting program can then load the object definition and messages can be sent to any object in the hierarchy to test its action. This allows a considerable degree of debugging of objects to take place,when coupled with the review/reset facility of the tracing mechanism *(vide supra)*. Suitable values can be assigned to objects and the effect of a method can be examined. Where a faulty method produces an error, a correction, to the inappropriate value of an item, can sometimes be made. Such corrections can then allow the method to proceed, causing more messages to arise, etc.

6.2 A prototype knowledge acquisition tool

The toolkit provides a utility to assist the user in defining instances of classes (i.e. rules) for a given class hierarchy. The intention is to allow the user to define (on top of a skeletal definition provided) all the symbols necessary for the system to be able to recognise English-like statements of rules. It can check and then translate them into the appropriate representation for the object-centred system.

The English syntax that is accepted for statement of the rules is somewhat restricted in form but is recognisably English. The function of a knowledge acquisition tool, in this context, might be said to be to capture 'semantic relevance' for the symbols it translates into internal tokens. Often, there may be a fairly direct translation from one symbol to one token using some pre-defined frame of association, giving a structure filled out with tokens. In this sort of translation there is a direct matching of symbol to token. In the toolkit, the scheme is slightly different; there being a difference in the nature of the knowledge capture. By definition, the symbols stated by the user, in inputting a rule, will have semantic relevance - because they have already been associated with a category of knowledge (e.g. a particular class of objects, which, to the user, have a special significance) or, by using expectations of what should be implicit from the known grammar of rules, the system can attach a likely categorisation to strings encountered.

The most useful aspect of the utility is not that it can accept and translate a suitable statement of a rule, but that it can also identify invalid rules, saying what is wrong. It can detect that a symbol may not be wrong, but not yet known to it, and allow the user to state what it refers to. The user may be able to mis-state a rule, e.g. by ambiguous or slightly incorrect use of names, the system may disambiguate it and suggest an appropriate meaning for the name.

6.2.1 A meta-description editor for defining metalevel knowledge

As with the object editor, the tool may be used to create a new definition or modify an existing one. Since this utility captures a deeper description of objects in order to generate items at a level beyond that of the object editor, which outputs an example of the object framework - a class hierarchy, it is referred to as a meta-description editor: that is, it is concerned with metalevel knowledge.

Two types of knowledge may be captured. Firstly, the user may specify more detailed descriptions of the nature of classes, and other items. If a type has been defined, i.e. of what nature its instances are, this can be recorded explicitly. The types of instance variables (i.e. primitive: double, character etc., or object reference: pointer, which would necessarily be the type accessing an object of a non-primitive type) can be stated and named. (A kind of metalevel knowledge about classes.) The system also constructs lists of relationships between method (names), classes and effectors. These lists are used to check that a named method is being correctly associated with a class, in a rule. In a manner similar to that for classes, effectors may also be 'meta-described'; their arguments may be typed (and named). This knowledge is useful for understanding the system. The knowledge acquisition tool is a prototype and is intended to assist documentation of this type of knowledge and also capture instances (rules) for a given object-centred definition. As such, the knowledge captured is meant to be the basis for a more sophisticated acquisition strategy that allows a less constrained statement of instances. The tool would accept much more ambiguous statements and use the meta-knowledge of the domain, as represented by the user's definitions, to elucidate coherent knowledge. It would be able to direct the user more accurately and could better 'guess' what was missing or irrelevant in a statement. Some work was done along these lines, but discussion of that is left to Chapter 8. Also, knowledge of what is expected in a method (i.e. what the named action does, e.g. assigns values or evaluates items, and what items the effector should deal with) could be used to provide semi-automatic writing of code.

The second activity of the utility is concerned with providing the means to capture instances (rules). Capturing class meta-knowledge, for example, puts it into a class hierarchy, as for any object definition. This means that there is a class (really a meta-class) for descriptions of any object-centred class definition. The (meta-) class for classes has, as its instances, the classes of an object-centred definition (the one for which rules are to be acquired). The utility loads details of the designated object definition into memory as instances of its own classes. It does this by loading, initially a skeletal object definition (called '**meta**'). **Meta** has classes, called u_classes, u_methods, u_effectors etc., which produce instances for each of the classes, methods and effectors of the object definition: i.e. it models the definition. The utility itself is in the form of an object-centred program whose methods relate to the construction of instances. Its class hierarchy has other members concerned with skeletal descriptions of components of instances (in this case rules). At present, rules only are catered for. Other types of instances could be accommodated by adding extra classes and methods to this part of the hierarchy.

The output of the utility is in the form of a meta-description hierarchy which can be used for further definitions. The (meta-)classes for the components of a rule have instances for constants, relational operators etc. On the first call of the utility (i.e. with no meta-description as argument) when a meta-description is initialised, the user may add to the definitions, given in **meta**, other symbols (strings) that should be recognised by the system. An example might be for the 'predicate' string that introduces a rule. The most obvious word might be 'if', for which the user might add a synonym such as ‘when’ to have the same effect. When definitions are introduced in this way, the user must state what token is to be understood as meant by the symbol. The strings ‘returns’ and ‘equals’ might be added to the language. The user would presumably indicate that the intended token is “=”. This is not a simple association of one with the other. For the purpose of the syntax, “=” is a ‘relational’ operator, and only in the context of a relational expression will this symbol be accepted for this meaning. When sufficient definitions have been introduced, the utility saves them in the categories to which they have been assigned. (New method names, for example, are not accepted, since a re-definition of the existing object hierarchy described by the meta-description would be implied.)

6.2.2 Capturing rules

When invoked with the name of a meta-description, the utility loads it and the user can now specify rules to be translated into the internal format. (If required, some new symbols may be defined, e.g. constants.) Firstly, the user must state which classes (of the original definition) are to be regarded as containing rules. This is not necessary if the utility has done this on a previous occasion. The particular rule to be captured is then indicated. Following an English statement of the rule, the system attempts to parse it. As described in Section 6.1, the internal form of rules is limited. The English grammar allowed is similarly restricted.

At the start, a predication symbol such as ‘if’, is required (prefixed by a phrase for quantification, in the case of such rules). A rule starting ‘for all colours if ....’ is quantified over that class. The intended class ‘colour’, in the case of the paint scheduler, would be unknown as a symbol, so the system will not be able to match it exactly, and looks for a near match, which will yield ‘colour_manager’. It will accept the string input for this class if it is indicated that ‘colour’ is an appropriate synonym. When no ‘match’ can be found among the known symbols, a warning is given and the user must choose the intended colour from a list. If none is acceptable, the rule will be rejected. In the case of a method, also, where no match can be made, the system informs the user that an attempt has been made to define an new item, and suggests that the original object definition may need extension. Should 'for_all' (or its synonyms) be missing (and the the first string not identified as a predication symbol) the 'class', or whatever is matched, will be rejected, because no quantification symbol was present and the next string was a 'predicator'.

In this way, the system can find errors and may disambiguate references. 'Valid' rules that are nonsensical cannot be rejected because they are semantically possible and syntactically correct. The system can recognise multiple conditions (and actions), separated by conjunctions (instances of the class u_conjunctions). If a relational operator is identified, then a constant or another expression is constrained to follow it. These expectations allow signalling of errors, both where a symbol is missing or of the wrong category.

Expressions are of the form <method> <class> and optional <selector>, followed by a 'relational' and then another expression or 'constant' (*vide supra* for the syntax for messages and their production from rules.) A similar approach to that of the messager is used for checking the relevance of the method and class references. Impossible combinations are rejected. A negated premise, such as of the form 'if a not the_same_as b' or its 'logical' equivalent ('if not a the_same_as b') is recognised. The internal representation of the rule is as the logical form ('..not a...') and the first form shown is inverted by the system. 'Not' used conjunctively, e.g. 'if a greater_than 2 not b greater_than 4' which is taken to mean that the expression is true for a > 2 and b <= 4, is understood as two relational expressions.

A more complicated re-arrangement is made for the selector string which is to be constructed in interpretation. It would make for very awkward English to force the selector string's symbols to be appended to the rest of an expression (as is the case for easy interpretation in the internal rule format). In the rule 'for_all colours if enquire colours less_than nought then ....' the phrase 'enquire 'colour_manager' is a clear enough message, derived from 'colours': 'enquire' is a known method, defined by 'colour_manager' and 'less_than' is a relational. So far so good. Unfortunately, the effector that is referenced requires more information than that. The 'enquire' method is not a class method and an instance must be referred to, say 'red'. This method also requires that a variable of the instance be accessed, say 'colour_code'. If, in the rule, these strings were mentioned in the same order as the messager receives the message the interpreter of the rules gives it, the intention would be rather obscured (with loss of declarative power?) e.g. 'for_all colours if enquire colours red colour_code .....'. The item 'red' would actually not appear because the interpreter of this quantified rule instantiates the rule for the instances of the class colour. The indication of the instance in the rule is by a place-holding name, e.g. 'current–item', which, it may be remembered, has a special significance, in that it identifies to the effector, ultimately, that the method is for an instance and not the class as a whole. The following shows the syntax that is allowed: 'for_all colours if enquire colour_code of current_item of colours less_than nought then ....'. This clearly associates 'current_item' with the class 'colours' (i.e. ultimately 'colour_manager'). To the author, this seems to be more readable: the purpose of the method is close, in the sentence, to the actual item that it accesses, the 'colour_code' variable. The meaning of the quantification of the rule over the class 'colours' is apparent: each time the rule is instantiated the item referenced is a particular instance of colours. Another point to note is that the strings 'of' are not represented in the internal rule syntax, but are meaningful in the English statement and assist parsing.

Where a previously undefined symbol is used, the utility will accept it as a synonym for another symbol (as instructed by the user) or, in the case of an apparent reference to a constant, for example, will recognise that such is expected and ask the user what token it represents. If the user cannot supply a suitable assignment, the system will indicate where and why there is an apparent error and reject the rule.

If a rule is acceptable, its form, as parsed into parts (conditional, action etc.)

is presented and can be confirmed by the user. The English statement and the translation are recorded. Following a session of rule acquisition, the utility can load the original object definition's instances and insert the new rules into their corresponding places, if wished. The utility is able to re-define the size of (rule) classes to accommodate the internal form of rules, if necessary. It can also write into the file used by the rule tracing mechanism, the English statements of rules, so that the explanation facilities use the latest text defined. Finally, the knowledge gained as meta-descriptions is, optionally, saved; the meta-description file is updated from the memory image of the description.)

6.3 An object-centred scheme for paint scheduling

The design techniques described above have been used to implement an object-centred version of the paint scheduler (see Chapter 1), called **START**. The knowledge base is incorporated in a modified version of the **PAINT** program. **START** consists of the front end of **PAINT**, with extensions for enabling the message and rule tracing activities, and uses the start-up code. It is, therefore, an example of the architecture suggested earlier (see Chapters 1 and 2) where a knowledge-based part resides in a conventional program. **START** uses database-handling, as in **PAINT**, to maintain the 'external' view of the factory state. Where **PAINT** would call Istel's algorithm to allocate colour, each batching decision in **START** is activated by sending a message to the object-centred knowledge base. (This request, for a scheduling decision according to the current state of the conveyor line, is mediated by a message of the form: 'load task_manager'.)

The message, 'load task_manager' must have a selector string as well. If the selector is "ALL", *viz* 'load task_manager: ALL', the knowledge base must load an image of the database into the objects of classes which model details of the map, colour, body and priority files (i.e. classes 'map_manager', 'colour_manager', 'body_manager', 'priority_manager' etc.). The effect of this message is two-fold: the database image is loaded; then, the colour allocation mechanism is invoked. The objects are initialised by a series of 'load' messages, e.g. 'load map_manager: ALL' etc., sent from the method. (The loading is done by invoking another method: 'load database_manager: ALL', which activates the 'load' methods of the objects reflecting each of the database files.)

The 'reverse' of this message is 'load task_manager: DUMP'. This allows the object-centred image of the database to update the database files, thereby allowing the state of the knowledge base to be saved. When the selector is of the form 'load task_manager: colour, batch', a different response occurs. This form is used on second and subsequent calls on the scheduler. (The image of the database should already be loaded and the relevant objects initialised). The meaning is that the 'batch' BIWs were achieved in 'colour' following the previous batching decision. As for the "ALL" selector, the 'db_manager' class receives a message to update the images of files because of the batch being achieved (i.e. counting down the appropriate requirements and increasing the achievements. 'Update db_manager: colour, batch' results in the 'db_manager' method sending out 'update' messages to the appropriate instance for the colour of the batch and objects of the 'body_manager' class. The map image is also updated. In principle, the updating of database images is achieved by the same algorithm as in the **PAINT** program, but access to identities of BIWs in the map, for example, is gained by requesting the appropriate map object (the 'old map' instance of the map_manager class) to say what body type is situated where. All values exchanged between classes are returned as the values of methods evoked from message-passing *(vide infra)*.

Figure 6.1 details the various classes in the object hierarchy. For each colour stated in the database file, there will be a corresponding instance of the class colour_manager; with variables indicating such items as total achievement and requirement, colour code etc.) The (constant) values for batching parameters (minpb etc.) are represented in a subclass ('colour_items') of 'colour_manager': one instance for each colour. In a similar fashion, the class 'body_manager' records body types and class 'priority_manager' any priority body/colour combinations specified. The organisation of the 'map–manager' class is slightly different. It has two instances: one is for the state of the map known prior to a batching decision, holding the code of each BIW in the map (database file) of the conveyor; the other is used to get a new image of how the map looks following a batching decision.

The process by which the body objects count down the requirement, and increase achievement, following an 'update' message, is essentially the same as that for **PAINT**. The algorithm consists of identifying each BIW that has been achieved in the map and marking down the particular body/colour totals. In the case of the object-centred system, messages are used to do this, for example, sent to the map requesting the identities of the various BIWs involved. The whole process is mediated by messages and no direct alteration of values is done, except by the objects that possess them themselves.

The requirement to restrict access to items may seem rather tedious and inefficient, but having privacy of data is the intent of such a system. It should also be noted that this knowledge base is meant to exist inside a real-time system and it is quite possible to have a situation where the knowledge of 'what is where' is subject to change. This is, for example, true of the BIWs nearest the sealer (which can be re-arranged). This is not, essentially a time-dependent form of non-monotonicity (in the sense that validity decreases with the age of the data) but one due to uncertainty in the accuracy of knowledge. The rationale for requiring map objects to release their information on request is to ensure that any such changes in the reliability of information could be catered for by explicit methods for handling non-monotonicity (as regards the view of an object from other objects). If body objects were allowed to access the map freely, without messaging, such control would be much harder to impose. It is not suggested that, in all cases, such an approach would deal with potential non-monotonicity. In the case of external changes in the database (or a series of such changes) which, for example, caused interruption of rule-based decision making, it is quite possible that a certain stage of reasoning had been reached that, after the change, was now invalid. In this case, it would be hard to retract certain assertions because they cause permanent changes in objects. When a rule fires, its assertions can cause an object to change a value. If the rule assertion now has to be retracted and a different line of reasoning followed, that is one dependency to change: it considerably more difficult to trace all the potential values whose previous values should now be reverted to (e.g. deKleer 1983), for example, because each item would have to be recorded as having a previous value, particular associations etc. This approach is logically complicated by the fact that objects would have to supply methods for value retraction. It would be unacceptable to demand that a global mechanism be allowed access to objects ignoring messaging conventions. The object scheme would be helpful, however, because the encapsulation of objects would allow specification (on an inherited/override basis) of just how each item should be justified etc. In many cases, the number of different methods for data justification could be limited: like families of objects would probably use the same method(s), so the total would be much less than linear in the number of classes. A reason maintenance scheme would, thus, be simpler to insert on a class-by-class basis.

In practice, such reason maintenance is not an issue for this knowledge base, although it might make the objects directly more responsive to external changes. Asynchronous events, such as the switching on/off of a paint colour, might affect the validity of the partial decision so far. If, after an interrupt, the decision-making is resumed, with appropriate tracing of dependencies and retraction of rule assertions etc., it may well take longer to resume than to abandon it altogether and start again, particularly if another interrupt occurs (which might interact with the first). One interrupt might imply undoing certain changes in data which would be relied on for undoing of the changes made necessary by the second interrupt. In such cases, there is an indeterminate state. It cannot, necessarily, be certain that this will not be the case, so any retraction of previous reasoning followed by resumption may have removed (now) incorrect items that would be required, in their incorrect form, for some future retractions to be possible. In the case of changes like constraining of colours, a check could be made, at the end of a decision, to see whether the change invalidated the colour chosen, i.e. if the selection were for 'yellow', then the asynchronous switching off of yellow could be noted and another colour chosen. This means that the time spent, after resuming the decision making, would have been wasted.

The above discussion makes reference to the encapsulation of objects and privacy of data. This is relevant in considering how rules mediate their effects and instances change their variables. In rule mediation, most messages are concerned with finding the value of some object, or setting it to a new value. (Methods for this are such as 'enquire' and 'set'.) Messages invoking instance methods, by convention, refer to the instance addressed in the selector; its first component. Where a particular variable is referenced, it is usually the second component. The third will refer to the value that the variable is to be set to. Another possibility is that a different transaction is to be done on the variable, for example, inverting it (negative to positive and vice versa).

On each invocation of the knowledge-based system, following updating of the database image, the method invoked ('load task_manager') causes the tasks which are required to select the batch etc. to be initiated. This is done by using a task loop. In the loop, the state of 'tasking' is tested, and while this has not reached a particular stage, in the body (of the loop) a message is sent to a subclass of the responding class (task_manager) to evaluate a rule set (of task rules). Initially, the value of the control item for the loop (the 'task' instance of 'task_manager') is 'selecting_colour'. The termination value is 'done'. An 'interpret task_rules by fire_first' message is then sent (to task_rules). The result of this is that the 'interpret' method of this class (the archetypal interpreter, inherited from its parent class, task_manager) is activated for its instances, which are rules, so that the first one with a conditional part that evaluates to true is fired, the others ignored.) In this case the rules are mutually exclusive. These rules are all of the form: 'if enquire task of task_manager returns state then .... and set task of task_manager to newstate'. In this example, 'state' is 'selecting_colour', 'selecting_batch' etc., i.e. any value that 'task' is allowed to have; 'newstate' is the next state in the performance of tasks, i.e. 'selecting_colour' becomes 'selecting_batch' ..... 'done'. When the last part of scheduling has been done the (second) action of the task rule fired will cause 'task' to be set to 'done', allowing the loop to end. The first actions of each of these rules causes a part of the scheduling process to take place. In effect, three activities are required for scheduling: firstly, selecting all those colours which could make a minimum batch - minpb (applying the principle of excluding for colours giving batches of less than a preferred size, e.g. because of lack of order cover in the map sequence); secondly, finding the largest batch possible for any colour that satisfied the first criterion (subject to the maxpb value); thirdly, selecting the 'best' colour, subject to the least-achieved criterion.

The rule set for tasks is an example of control knowledge. The expression of those rules causes a 'cascade' of effects which mediate scheduling. In some ways, the control scheme resembles a blackboard - the activation of one part of the system evoking activity in other parts - those parts activated will be rule sets, which are objects with associated control, and so there is a correspondence between blackboard and knowledge sources and this architectural feature.

In the case of the colour selection task, the message 'quantify batch_min_rules by fire_all' is sent to the rule set 'batch_min_rules' as a result of the task rule's first action, which is '.... then quantify batch_min_rules by fire_all ....'. The reason that the interpreting request is introduced by 'quantify' and not 'interpret' is because this rule set (class) is one which applies to a set of items (i.e. all the instances of a class) in this case the colours. All 'batch_min_rules' rules are of the form: 'for_all colours if .....'. The interpreter applies each of the rules, in turn, to the first instance, then the next etc., e.g. 'red', 'blue', 'green', 'yellow'. The first rule specifies that colours with a negative colour code (i.e. which have been switched off) are not considered, so should have a potential batch size of nil and the colour code should be marked as off ('for_all colours if enquire colour_code of current_item of colour_manager less_than nought then set batch of current_item of colour_manager to nil and set colour_code of current_item of colour_manager to minus'). The second rule looks at the colour code and if it is positive sets up the batch objects with the current colour details for the scheduling decision: 'for_all colours if enquire colour_code of current_item of colour_manager greater_than nought then copy current_item of colour_manager to batch_manager'. These rules show that the level of control is focused gradually. The first references are to the colour code, held in 'colour_manager' instances. If a colour is valid, current details, as updated by previous decisions and held in colour objects (images of the database file) are copied to the 'batch_manager' class, which is used as a 'temporary' image for partial batching decisions.

The second task rule will activate the (batches of up to maxpb) batching step: 'if enquire task of task_manager returns selecting_batch then quantify batch_inc_rules for fire_first and set task of task_manager to selecting_best'. (This will fire only when the first task has been completed.) Evaluation of 'batch_inc_rules' is by reference to colour objects, as for class batch_min_rules. The first of these rules specifies looking for a priority batch (i.e. one containing at least one priority body/colour combination in the batch of BIWs on the line). The expression of this rule's condition is to send an interpret message to evaluate the priority rules of class 'prty_rules' for each colour instance. If this rule succeeds for any colour (after all have been tried), the interpreter quits for this rule set, because it was invoked to find the first successful rule. The second rule checks that, for the colour instantiated, the minpb size of batch has been found. If this is so, the action of the rule is to extend the batch as far as possible (i.e. up to maxpb or lack of cover in the map). The third rule is a default one: if the previous batch can be extended (i.e. the leading BIW in the map has cover in the instance's colour) and the previous batch had this colour, the batch size is set to one. This rule calls for the evaluation of a rule set for extending batches ('batch_ext_rules'). If this returns a successful result, the action (set a batch size of one) can be asserted.

The expression of rules proceeds as in the above examples. A full list of rules and details of objects and their methods are given in the Appendix (see Chapter 7 for more examples of how rule expression works). The effect of rules is to produce permanent changes in the objects acted on. This is equivalent to updating the working memory elements of a production system. Eventually, enough knowledge has been accumulated to state the best decision. In this system, a set of batch sizes has been evaluated for the colour instances (strictly, 'copies' of them held in batching objects). The variables of these (batch) colour items show whether a priority has been found. If one has been found, then only 'prioritised' instances are considered. If an extension batch is found, only one colour (that of the previous batch) will have a batch greater than nought (i.e. one). A method (of 'batch_manager') is required to select the best from a number of alternative. This is an object-centred form of the algorithm used in **PAINT** (i.e. find the least a.p.).

It can be seen that the procedural expression of Istel's algorithm has been partly explicated by re-statement in rules and objects. The proportion of knowledge that is hidden (i.e. in procedural code) is much less. The methods to achieve tests and changes are fairly simple and, of course, result in messages (which can be traced). The nature of statement of messages is, thus, fairly declarative. This is a benefit over more conventional programming, where the variables, code and their interactions are probably hard to distinguish. If the strict use of encapsulation and independence of data in objects is adhered to, all changes in objects should be observable.

One aspect that is interesting is to note the nature of some classes and their instances. The task_manager class has but one instance, plus access methods. 'Task' is really a class variable in the Smalltalk sense: 'task_manager' requires no instances. The instances of the class might be represented by subclasses. This is no particular disadvantage, however (apart from the messaging convention for instances). One other way for a class to have class variables is represented by some methods' possession of private variables. These perform the function of class variables because they are accessible via their method (available to all members of the class). Their declaration is **static**, so they persist between invocations of their method and can be initialised.

Chapter 7 An evaluation of the knowledge-based approach

In this Chapter, a trace will be made through the decision-making process in the paint scheduler. The activity of rules and general message traffic will be highlighted to illustrate how the expression of knowledge is mediated. (The rules discussed are listed in Section 7.1.) Following this exposition, an example is given of how inappropriate rules can cause incorrect decision-making. The use of the description editor is illustrated in defining new rules that would expand the knowledge base in new ways. (Further and wider modifications to the knowledge base are discussed in Chapter 8.)

7.1 Rules (as accepted by the knowledge acquisition utility)

Class # Name

task_rules i. selecting colour:

```
        if (enquire task_manager) returns selecting_colour
        then quantify batch_min_rules by fire_all and
                update task_manager to selecting_batch
```

ii. selecting batch:

```
        if (enquire task_manager) returns selecting_batch
        then quantify batch_inc_rules by fire_first and
                update task_manager to selecting_best
```

```
                    iii. selecting best:
        if (enquire task_manager) returns selecting_best
        then interpret batch_best_rule by fire_first and
             update task_manager to done


batch_min_rules     i.  colour off:
        for_all colours
                if (enquire code of current_item of colours) less_than nil
                then set batch of current_item of batch_clr_itms to nil and
                     update code of current_item of batch_clr_itms to minus
                    ii.  copy colours:
        for_all colours
                if (enquire code of current_item of batch_clr_itms) not less_than nil
                then copy current_item of colours to batch_clr_itms
                    iii. nil batch:
        for_all colours
                if copy current_item of body_colours to batch_clr_itms and
                   set cover_done_flag of batch_manager to on and
                   interpret form_min_rules by fire_all for current_item and
                   (enquire batch_size of batch_manager for current_item) less_than
                   (enquire batch of current_item of colour_items)
                then update batch of current_item of batch_clr_itms to nil


batch_inc_rules     i.  priority:
        for_all colours
                if (interpret batch_prty_rule for current_item by fire_all) returns
                        priority_batch
                then update priority of current_item of batch_clr_itms to on
```

```
                    ii. maximum batch:
    for_all colours
        if enquire batch of current_item of batch_clr_itms not less_than
          (enquire minpb of current_item of colour_items)
        then interpret form_inc_rules by fire_all for current_item
                    iii. extend batch:
    for_all colours
        if (interpret batch_ext_rules for current_item by fire_first) returns
          cover_one
        then update batch of current_item of batch_clr_itms to one and
              valid batch_ext_rules for current_item
                    iv. batch one:
    for_all colours
        if (enquire code of current_item of batch_clr_itms) not less_than one
          and cover current_item of batch_clr_itms
        then update current_item of batch_clr_itms to one
                    iv. batch fail:
    for_all colours
        if (enquire code of current_item of batch_clr_itms) not less_than one
        then set batch_size of batch_manager to one and
              update achievement of current_item of batch_clr_itms to minus


batch_best_rule     i. prioritised:
    if best batch_clr_itms
    then set batch_best_rule
                    ii. normal batch:
    if valid batch_manager
    then set batch_best_rule
```

```
                    iii. previous colour:
        if copy batch_clr_itms
        then set batch_best_rule
                    iv. crisis valid:
        if valid batch_clr_itms for plus
        then set batch_best_rule
                    v.  crisis fail:
        if valid batch_clr_itms for minus
        then set batch_best_rule

batch_prty_rule     i.  quit colour:
        if (enquire batch of current_item of batch_clr_itms) returns nil
        then set batch_prty_rule
                    ii.  mark a p:
        if (enquire batch of current_item of batch_clr_itms)
           greater_than_or_equal_to
          (enquire minpb of current_item of colour_items)
        then quantify batch_clr_itms by high_value for achievement for
             current_item
                    iii. priority batch:
        if (batch current_item of batch_clr_itms for plus) greater_than nil
        then set batch_prty_rule

batch_ext_rules     i.  one only:
        if (enquire batch of current_item of batch_clr_itms) greater_than one
        then set batch_ext_rules
                    ii.  colour still on:
        if (enquire code of current_item of batch_clr_itms) not greater_than nil
        then set batch_ext_rules
```

```
                    iii. previous colour:
        if not match batch_ext_rules for current_item
        then set batch_ext_rules
                    iv.  cover one :
        if cover batch_ext_rules for current_item
        then set batch_ext_rules


    form_min_rules      i.  batched:
        if (enquire cover_done_flag of batch_manager) is off
        then set form_min_rules
                    ii.  constrained:
        if (enquire code of current_item of batch_clr_itms) less_than nil
        then set form_min_rules
                    iii. limit:
        if (enquire batch_size of batch_manager) equals
           (enquire minpb of current_item of colour_items)
        then set form_min_rules
                    iv. no cover
        if not cover for batch_clr_itms for current_item
        then set form_min_rules
                    v.  cover
        if cover for batch_clr_itms for current_item
        then update batch_size of batch_manager for plus
                    vi. check priority:
        if enquire prty_items for current_item
        then set priority of current_item of batch_clr_items
                    vii. continue minpb:
        if (enquire cover_done_flag of batch_manager) is on
        then interpret form_min_rules by fire_all for current_item
```

```
form_inc_rules          i. batched max:
        if (enquire cover_done_flag of batch_manager) is off
        then set form_inc_rules
                        ii. no colour:
        if (enquire code of current_item of batch_clr_itms) less_than nil
        then set form_inc_rules
                        iii. limit max:
        if (enquire batch_size of batch_manager) equals
           (enquire maxpb of current_item of colour_items)
        then set form_inc_rules
                        iv. no more cover
        if not cover for batch_clr_itms for current_item
        then set form_inc_rules
                        v. cover max
        if cover for batch_clr_itms for current_item
        then update batch_size of batch_manager for plus
                        vi. check prty max:
        if enquire prty_items for current_item
        then set priority of current_item of batch_clr_items
                        vii. continue maxpb:
        if (enquire cover_done_flag of batch_manager) is on
        then interpret form_inc_rules by fire_all for current_item
```

7.2 A trace of a painting run

The order profile as in Figures 7.1 and 7.2 was used as the starting point for a paint batching test. The leading part of the sequence of Bodies In White (BIWs) is shown in Figure 7.3.

Colours allowed	Maximum used	(Total colour items set)
32	12	70

Colour	minpb	maxpb	maxb	Required	Achieved	(Colour name)
1+	3	50	75	419	88	white
2+	5	34	53	249	66	midnight black
3+	4	40	60	570	158	scarlet
4+	4	20	58	496	135	azure
5+	3	35	80	245	28	lemon
6+	3	35	60	439	107	bronze
7+	5	15	40	294	50	sea green
8+	4	35	60	404	110	silk green
9+	3	40	55	419	82	beige
10+	4	47	60	336	88	sunset orange
11+	6	40	80	493	124	silver
12+	3	31	49	608	164	champagne

(+ signifies availability for a colour)

Colour order profile

Figure 7.1

```
Body codes allowed       Maximum used
   12                        10


Body      Tot. req.  Prty. req.   Achieved  Prty (ach.)     Name
  1         450          0          232         0           200 3 dr
  Colour   Tot. req.  Prty req.   Achieved  Priority (ach.)
    2+        103        0           34         0           b1 midnight black
    4+         76        0           60         0           b1 azure
    6+         63        0           72          0          b1 bronze
    8+         97        0           40          0          b1 silk green
   10+        111        0           26          0          b1 sunset orange

  2         875          0          220         0           200 5 dr
  Colour   Tot. req.  Prty req.   Achieved  Priority (ach.)
    1+         83        0            0          0          b2 white
    2+         92        0           18          0          b2 midnight black
    3+         78        0           32          0          b2 scarlet
    4+         83        0            0          0          b2 azure
    5+         68        0            0          0          b2 lemon
    6+         59        0            9          0          b2 bronze
    7+         48        0           20          0          b2 sea green
    8+         46        0           55          0          b2 silk green
    9+         88        0           15          0          b2 beige
   10+         55        0           27          0          b2 sunset orange
   11+         22        0           44          0          b2 silver
   12+        153        0            0          0          b2 champagne

  3          90          0           46         0           800 3 dr
  Colour   Tot. req.   Prty req.   Achieved  Priority (ach.)
   11+         90          0          46          0         b3 silver
```

Figure 7.2 (part)

```
  4         175          0           98         0           800 5 dr
  Colour   Tot. req.   Prty req.   Achieved  Priority (ach.)
    3+         41          0          40          0         b4 scarlet
    9+        134          0          58          0         b4 beige

  5         602          0          225          0          monterey
  Colour   Tot. req.   Prty req.   Achieved  Priority (ach.)
    4+        215          0          61          0         b5 azure
   12+        387          0         164          0         b5 champagne

  6         739          0           91          0    monterey estate
  Colour   Tot. req.   Prty req.   Achieved  Priority (ach.)
    3+        262        0            16          0         b6 scarlet
    6+        111        0            26          0         b6 bronze
    7+        123        0            15          0         b6 sea green
   11+        243        0            34          0         b6 silver

  7         712          0          113          0          direttore 3 dr
  Colour   Tot. req.   Prty req.   Achieved  Priority (ach.)
    1+        102        0            35          0         b7 white
    3+        105        0            32          0         b7 scarlet
    6+        138        0             0          0         b7 bronze
    8+        127        0            11          0         b7 silk green
    9+        138        0             0          0         b7 beige
   10+        102        0            35          0         b7 sunset orange
```

8	826	0	132	0	direttore 5 dr
Colour	Tot. req.	Prty req.	Achieved	Priority (ach.)	
1+	93	0	43	0	b8 scarlet
3+	29	0	38	0	b8 azure
4+	68	0	0	0	b8 lemon
5+	114	0	23	0	b8 bronze
6+	68	0	0	0	b8 sea green
7+	123	0	15	0	b8 silk green
8+	134	0	4	0	b8 beige
9+	59	0	9	0	b8 sunset orange
11+	138	0	0	0	b8 white

9	236	0	38	0	unterbahn 3 dr
Colour	Tot. req.	Prty req.	Achieved	Priority (ach.)	
1+	73	0	10	0	b9 white
2+	54	0	14	0	b9 midnight black
3+	55	0	0	0	b9 scarlet
4+	54	0	14	0	b9 azure

10	267	0	5	0	unterbahn 5 dr
Colour	Tot. req.	Prty req.	Achieved	Priority (ach.)	
1+	68	0	0	0	b10 white
5+	63	0	5	0	b10 lemon
10+	68	0	0	0	b10 sunset orange
12+	68	0	0	0	b10 champagne

Body order profile

Figure 7.2 (cont.)

Distribution biases:

up to minpb	minpb to maxpb	more than maxpb
1.000000	1.000000	1.000000

```
2  6  4  6  4  4  7  3  7  6  6  4  3  6  3  5  6  6  6  5  6  6
5  6  5  5  3  6  4  6  3  5  6  6  3  6  3  4  7  3  7  6  6  3
6  7  5  7  6  6  5  4  4  8  8  5  5  8  8  8  5  8  8  8  4  5
7  6  8  7  7  3  5  5  8  8  5  5  5  8  8  5  3  3  8  8  6  5
5  6  7  5  5  8  4  8  8  8  8  5  5  5  8  8  4  6  6  6  8  8
5  8  8  8  8  8  4  3  5  5  5  5  4  5  5  4  5  8  8  8  6  8
8  8  8  8  9  6  8  8  9  5  8  8  10 6  8  8  6  9  9  4  7  7
8  8  8  4  5  5  5  5  5  6  5  5  5  5  6  5  6  6  6  4  7  5
5  6  5  5  5  5  5  7  7  6  6  5  6  6  4  6  5  7  6  4  7  6
5  7  7  5  7  5  5  6  5  5  5  7  7  5  5  6  6  7  7  6  8  7
7  5  6  8  8  9  6  7  7  9  10 6  6  5  5  5  6  10 10 10 5
5  4  8  8  8  8  8  8  8  8  8  5  7  7  8  8  8  10 8  8  8  10
8  8  7  8  8  9  9  9  ........
```

Sequence of body types on conveyor line

Figure 7.3

Assuming that the initialising stages of processing, as described in Chapter 6, have taken place, the rule interpretation progresses as follows. Rules, as stated earlier in this chapter in Section 7.1, are referenced here by the class and instance names.

The state of the loop variable ('task') will be 'selecting_colour' on initialisation. Within the loop itself, a message is sent to 'interpret task_rules by 'fire_first'. This causes the call_rule_set effector to be activated on the specified instances of the class (task_rules). As discussed previously, the significance of 'fire_first' is that the first satisfiable rule only is fired (taking the instances of the class in turn). All these rules are sensitive to the state of 'task' and, in this case, the evaluation of the rule 'selecting colour' (the first tried) yields a satisfied conclusion: 'if enquire task_manager returns selecting_colour then ...'; an 'enquire task_manager' message is originated by the interpreter parsing the condition and the class replies with the value of 'task'. The action 'quantify batch_min_rules by fire_all ...' is enabled and a 'quantify' message is sent to 'batch_min_rules'. In both cases of interpretation, neither class possesses the method and they inherit 'interpret' and 'quantify' from an ancestor (task_manager). The quantification method of task_manager is implemented by the 'call_gen_rules' effector. As previously described, this method iterates over the members of the class specified in the quantification part of the rules being interpreted. All of the class batch_min_rules specify application over 'colours'. The first rule ('colour off') applies this quantification to the condition 'if enquire code of current_item of colours less_than nil'. As is evident from Figure 7.1, no colour is unavailable and when an 'enquire code' message is sent to each colour, in turn, the class responds with the colour code, which is always greater than nought. The relation 'less_than nil' is false and so the next rule in the class will be consulted for the colour. 'Copy colours' specifies a similar condition: 'if enquire code of current_item of batch_clr_itms not less_than nil', but in a negated form; so this is true for all colours tried. The same message results in the colour code being returned. This may seem wasteful but allows 'independence' of rules - which, paired, as in this case, form the logical equivalence of an if ... else sequence. For each colour, then, the action will be applied. The message 'copy current_item of colours to batch_clr_itms' is sent and causes the recently updated (following a previous batching decision) detail for the colour to be copied to the appropriate batch instance.

The third rule ('nil batch') has a condition with a trigger - 'if copy current_item of body_colours to batch_clr_itms' which causes a message to be sent to acquire more colour details for the batch item. The next part of the condition 'and interpret form_min_rules by fire_all' causes the interpretation of a set of rules for the colour being considered and tries to form a valid minimum batch. (This itself contains a trigger to initialise a control variable allowing expression of the rules - '... and set cover_done_flag of batch_manager to on ...'). The final part tests the batch size formed and compares it with the minpb value for the colour '... and set batch_min_rules for current_item less_than enquire minpb of current_item of colour_items then update batch of current_item of batch_clr_itms to nil'; the 'set' message in this case causes batch_clr_itms to reply the potential batch size formed and then the sending of an 'enquire' message to colour_items gains the relevant comparison value. The interpreter compares these two results and sends a message to any instance of batch_clr_itms where the comparison is true. The message requests the instance to now update its batch size to nil. This means, when all colours have been estimated, that the batch size is at least minpb or nought. (The alternative to using rules for an essentially procedural determination of batch size is a method 'batch' for batch_clr_itms, which uses an effector, 'form_batch', to compare the given sequence on the conveyor with the order cover for colours and bodies and find a valid batch. The rule would specify 'batch batch_clr_itms for current_item' instead of the 'interpret ...' part. This is an example where a method can recode the effect of rule expression). This rule's ('nil batch') action implements the principle that the minimum batch worth considering, in normal cases, is at least minpb BIWs. If rule interpretation is used, it is difficult to arrange that a value, not directly related to the rule, such as potential batch size, be returned as the value of the method (interpreter). The result of interpretation of a rule set should be logically true or false or, perhaps the last rule evaluated. In interpreting the rules, it is a convention, as with other parts of the object system, that a logically false result indicates failure to evaluate a rule, not that the rule set gave a 'true' or 'false' answer, which would be meaningless where rule expression is mediated by message-passing. If values other than fail/succeed are required, then a reference to the last rule evaluated in the set is allowed, since this should be unequivocal in meaning.

Expression of the batching rules ('form_min_rules') follows, as stated above, by 'set'-ting of the batch_manager instance variable ('cover_done_flag'). (When this flag is 'on', the rule set is enabled.) The first rule of this set ('batched') checks that the 'cover_done_flag' is 'on' and if 'off' cuts the rule expression - 'if enquire cover_done_flag of batch_manager is off then set form_min_rules'. The 'constrained' rule, as in other rules checks that the colour is available: 'if enquire code of current_item of batch_clr_itms less_than nil then set form_min_rules'. In this rule, again, the action is to cut rule expression (for the colour under consideration.) The 'limit' rule checks that the current potential batch size is less than minpb for the colour and quits if the batch has reached the limit - 'if enquire batch_size of batch_manager equals enquire minpb of current_item of colour_items then set form_min_rules'. The next rule ('no cover') looks for a match on the colour for the next BIW on the line (to add to the batch). If there is none, then the batch is complete and the rule class is 'cut'. (This is explained in more detail for the extension of batches, *vide infra.*) The rule ('cover') has the converse condition to the previous rule ('if enquire batch_size of batch_manager equals enquire minpb of current_item of colour_items then set form_min_rules') and, if a BIW can be added to the batch, sends an update message to the batch_manager instance variable 'batch_size'; in this case, the 'plus' selector does not set a particular value but requests that the current value be incremented. The 'check priority' rule is optional but can be used to establish that the current BIW under consideration is prioritised for the colour. It causes the same messages to be sent to the class prty_items' instances as the method that operates in the rule set 'prty_rules' ('if enquire prty_items for current_item then set priority of current_item of batch_clr_itms'). The final rule ('continue minpb') checks that the batch is not marked as complete ('if enquire cover_done_flag of batch_manager is on'). This is true for the first BIW for all colours (they have cover in type #2) and so the 'cover' rule succeeds and the others fail, except for the last ('continue minpb'). The result is that a potential batch of one is possible for all and the action of the last rule is, recursively, to call interpretation of the set again ('if enquire cover_done_flag of batch_manager is on then interpret form_min_rules by fire_all for current_item'). In the case of 'scarlet' the 'cover' finishes at the seventh BIW (type #7) but for all others the batch is 'complete' after the second or first BIW. Thus the third rule of the batch_min_rules class ('nil batch') will be enabled for all colours, other than 'scarlet', and the principle of ignoring batches smaller than minpb will be observed (setting them to nil). Interpretation of 'form_min_rules' will cease at the limiting body because the rule to continue expression will fail after the flag for completion is set. Rule expression unwinds (back to that for 'batch_min_rules') instead of recurring.

Processing of the batch_min_rules class is now complete. It was invoked from the action of the 'selecting colour' rule whose second action is now evaluated: '... and update task_manager to selecting_batch'. The effect is to request 'task_manager' to update 'task' to a new value. This value, 'selecting_batch', now will enable another task rule to fire ('selecting batch') on the next iteration through the task control loop.

The second task rule ('selecting batch') is fired and a 'quantify by fire_first' message is sent to 'batch_inc_rules'. The interpreter method treats the rule set in a similar way to 'batch_min_rules', except that only the first successful rule is to be applied to each of the instances of the quantified class: again, the quantification is over colours, specifically resolving to 'colour_manager'.

The first of the rules interpreted is 'priority', the condition of which is satisfied when a body has been prioritised for the colour in question. This is established by attempting interpretation of priority rules (class batch_prty_rule). The condition of the rule ('priority') causes the message 'batch_prty_rule for current_item by fire_all' to be sent. This means that 'current_item' is expanded into a reference to the instance of the relevant colour on each evaluation of the rule. The relational part of the rule is 'returns priority_batch'. This relationship is true when the interpretation of the set of rules referenced returns 'priority_batch'. That occurs when the last rule of the new set is successfully applied: i.e. that associated with 'priority_batch'.

Class batch_prty_rule has two rules, 'quit colour' and 'mark a p', which establish that a batch can be set (i.e. non-nil so far) and that its a.p. (achievement proportion) can be calculated, allowing a priority batch to be identified later. The condition of 'quit colour' is 'if enquire batch of current_item of batch_clr_itms returns nil'. This is applied to the colour that has been evaluated in the invoking rule (and passed to the interpreter). The result of potential batch determinations to date is shown in Figure 7.4.

Colour	Possible batch	Actual batch	Limiting BIW	Principle
white	2	-	6	minpb = 3
midnight black	2	-	6	minpb = 5
scarlet	2 6 4 6 4 4	2 6 4 6 (4 4)	7	minpb: 4; no cover: 7
lemon	2	-	6	minpb = 4
bronze	2	-	6	minpb = 3
sea green	2 6	-	4	minpb = 3
silk green	2 6	-	4	minpb = 5
beige	2	-	6	minpb = 4
sunset orange	2	-	6	minpb = 3
silver	2 6	-	4	minpb = 6
champagne	2	-	6	minpb = 3

Partial batch determinations

Figure 7.4

For all other colours than scarlet, this condition is not satisfied and the action is not fired. The message 'set batch_prty_rule' would be sent on successful unification for the colour. The 'set' method for this rule class is defined somewhat differently from other set methods. Since a rule is essentially a static item, the set has a different meaning from that of altering something. The effect is to mark the rule as having been fired and quit from further evaluation of the class (irrespective of how the class was referenced). The effector 'call_cut' is a dummy action representing the effect of a 'cut' operator in Prolog, for example.

In the case shown, this means that for scarlet only, a worthwhile batch (of at least minpb BIWs) is possible.The next rule is evaluated to screen out small (less than minpb sized) batches. For colours other than scarlet, it is necessary to remove them from consideration.

The second rule of this set ('mark a p') is only applied to the 'scarlet' instance. The same message to find the batch size is sent (originated from the condition, similar to the 'quit colour' rule: '*if enquire batch of current_item of batch_clr_itms* greater_than_or_equal_to enquire minpb of current_item of colour_items') but this time a comparison is made with the colour batching parameter, by sending the 'enquire minpb of current_item of colour_items' message. This rule applies only to scarlet, all other colours having at least one valid BIW but a batch size too small (less than minpb) e.g. silver, with two BIWs, failing the test, since at least six are required. The action, for scarlet, is to send a 'quantify batch_clr_itms by high_value for achievement for current_item' message. The meaning of 'quantify', as applied to to a non-rule class, is, like 'set', different from usual. It means that 'batch_clr_itms' should calculate an a.p. for any instance that has a valid batch size. Any instance for which this is not true has a value set for a.p. that makes it impossible to be selected ('high_value'). This value will always be so high that any a.p. calculated can never be higher, providing a default 'don't care' state for the colour and any 'valid' a.p. is always favoured. ('Least achieved' being most favoured.) The values for a.p.s are obtained by sending the requisite 'enquire' messages to colour objects, asking for their requirement and achievement values.

In the case of scarlet, then, the third rule ('priority batch') will give a positive result: 'if batch current_item of batch_clr_itms for plus greater_than nil then set batch_prty_rule'. The batch' message requests that the batch for the instance specified should be checked for priority items. The 'plus' selector indicates that if a priority does not exist within the batch, the test fails (as it will for all colours here). The action is not fired and so the return value for the interpretation of this class is "OK". This means that every (rule) evaluation was able to proceed and the result (returned after all evaluations)will not unify with the comparison looked for by the rule that invoked the interpreter (namely 'priority'). Where a priority is found, the effect is to send 'set' batch_prty_rules and the method invoked tells the interpreter that a cut has been made. In these circumstances, the usual value returned from the interpreting method ("OK" or "FAIL") is not replied and the last successful rule applied is identified. For this rule set, that would be 'priority_batch' and so the invoking rule's condition ('priority') is unified. It should be noted that 'batch for plus' is responded to by 'batch_clr_itms' by seeking a batch bigger than minpb for priorities is necessary (i.e. up to maxpb) and setting the batch size accordingly. This follows the Istel algorithm.

In the above example, the 'priority' rule always fails. If the opposite were true, the use of 'fire_first' as a selector in invoking interpretation of 'batch_inc_rules' would mean that a priority had been found and no more rules of this class were appropriate. The expression of this class would stop and control return to the invoking method (interpreting the task rule 'selecting batch').

The second rule of 'batch_inc_rules' (maximum batch) is now consulted. It causes 'batch_clr_itms' to be sent an 'enquire' message requesting the batch size for the instance being unified. This value is compared with that of the message 'enquire minpb of current_item of colour_items'. As discussed above, the situation now is that 'scarlet' is the only valid colour and 'if enquire batch of current_item of batch_clr_itms not less_than enquire minpb of current_item of colour_items less_than then quantify form_inc_rules by fire_all' succeeds for it. This means that the rules to form a maximum batch are applied. The rationale for 'form_min_rules' and 'form_inc_rules' is closely similar (maxpb being the limit rather than minpb). In this case, cover for 'scarlet' is obtained for the sequence up to the first type 7 body ('direttore 3 dr'). The result of this is that all colours but 'scarlet' (with six BIWs) are regarded as having a valid batch of less than minpb. The 'fire_first' criterion of the interpreter invoked means that further evaluation of this set is stopped and control now unwinds to the task_rule interpretation. This follows, of course, application of this second rule to each instance of the colour class specified (i.e. 'azure', 'lemon' ... 'champagne')

The third rule of the set is 'extend_batch', which invokes the interpretation of 'batch_ext_rules' ('interpret batch_ext_rules for current_item by fire_first'). The target rule set attempts to establish that a previous batch is extendable by one BIW (the leading BIW on the line). In the first of its instances ('one only') the batch_clr_itms class instance (for the colour being unified) is sent a message, 'enquire batch', and responds with the known potential batch size. (In the above example, this rule would not have been examined - but assume for the moment that no cover had been found for any colour; sufficient for a valid batch, so far.) The response, when this rule is evaluated as a default to the priority or normal batching decisions would mean that no colour had a valid batch, but this rule is intended to be read as independent of other rules and implements the batching principle that a batch can be extended past maxpb only when priority or normal batching fails; a 'partial' batch found (i.e. less than minpb) would be regarded as too small. It also means that the rule set 'batch_ext_rule' has the correct effect, in being regarded as having been applied in the right context: 'if enquire batch of current_item of batch_clr_itms greater_than one then set batch_ext_rules'. As for priority rules, the 'set' message means that a cut is applied to the rule set (on that attempted colour unificaton) and the exit value for interpretation ('one_only') is not 'cover_one', so the 'extend_batch' rule cannot succeed.

The second rule of 'batch_ext_rules' ('colour still on') checks that the colour has not been switched off (a requirement of the extending batch principle): 'if enquire code of current_item of batch_clr_itms not greater_than nil then set batch_ext_rules'. Any colour that had been switched off would have a negative code value and would be ignored (via 'set', the exit value of interpretation not satisfying the invoking rule).

The third rule ('previous colour') checks that the previously batched colour (obtained from a method returning the value of the global variable 'Prevcol') unifies with the instance being unified ('if not match batch_ext_rules for current_item then set batch_ext_rules'). When this rule is applicable, the cut is applied (and this rule interpretation has not succeeded). Only if the fourth rule ('cover one') succeeds is the required value ('cover_one') returned when the rule interpretation is cut ('if match batch_ext_rules for current_item then set batch_ext_rules'). The 'cover' method tries to match the leading BIW on the line with a body object defined to the colour in question and returns "OK" when there is such a body/colour combination with outstanding requirement for at least one more BIW. If the condition is true ("OK") the cut, applied in the action of the rule, ensures that the value for the successful rule is returned ('cover one') and the invoking rule ('extend batch') is satisifed. This would mean that messages to 'update batch of current_item of batch_clr_itms to one' and 'valid batch_ext_rules for current_item' would be sent, causing a record of the colour and singleton batch to be saved. The 'valid' message would then cause a cut to be applied to the class ('batch_inc_rules'). This would be true (in the absence of a valid batch) for any of the colours (the previous colour would have been one of them) because there was outstanding requirement for body type 2 (200 5 dr) in all colours and so one match would have taken place.

The fourth and fifth rules of 'batch_inc_rules' ('batch one' and 'batch fail') try to find a singleton batch (by default). 'Batch one' attempts to find a colour that has cover for the leading BIW on the line, whereas 'batch fail' is more general and tries to find the least achieved colour and schedule the leading BIW to it, even in the absence of cover. 'Batch one' causes the message 'cover current_item' to be sent to 'batch_clr_itms' for any available colour: 'enquire code of current_item of batch_clr_itms not less_than one'. If this is true, the action is to assign a singleton batch: 'update current_item of batch_clr_itms to one'. The condition tries to establish that the leading BIW has cover in the unifying colour. Again, this condition seeks to make the rule applicable independently of rules applied previously. For 'batch fail', the condition tries to find any colour that is not unavailable: 'for_all colours if enquire code of current_item of batch_clr_itms not less_than one then set batch_size of batch_manager to one'. The batch size for available colours (if this rule were expressed in the above circumstances, except that a 'valid' batch had not been found already, this would be true of all colours) is set at one. The second action, to send an 'update achievement of current_item of batch_clr_itms to minus' message, shows that the a.p. should be calculated for the instance and should be set to a negative value (as a flag that this rule has succeeded.)

At this stage, one rule of batch_inc_rules will have been applied for each colour and interpretation unwinds to the previous rule expression, i.e. of 'task_rules'. As for 'selecting colour', the 'selecting batch' rule has a secondary action to 'update' the value of the 'task' instance ('update task_manager to selecting_best').

The third task rule ('selecting best') is now enabled and its action is to 'interpret batch_best_rule by fire_first. This, as in the above cases, causes the specified rule set to be expressed.

The class 'batch_best_rule' contains the following rules (not quantified): 'prioritised' sends a 'best' message to batch_clr_itms. The method invoked looks for any batch_clr_itms instance which has been marked for priority and returns "OK" if one is found. The action part is to send a 'set' message to the instance (which defines a 'cut' as for all rules of this set). No further work is required to select a priority, the method picks the appropriate item as the most favoured one by sending a 'set' message to the 'batch_manager' class to inform it of the required colour and batch size. The a.p.s marked for prioritised items are selected from in order of priority.

In the case of the second rule ('normal batch': 'if valid batch_manager then set batch_best_rule') the method invoked compares all the a.p.s set and finds the lowest, i.e. most favoured colour. Note that 'scarlet' was found to be valid and an a.p. of $(6 + 135) / 496 \approx 0.309$ would be set. No other colour could be valid and so the values for 'sea green', 'azure' etc. would all be 'high_value', i.e 200000. Obviously, 'scarlet' is selected. The 'set batch_best_rule' message means, as in above cases, that the rule application is cut (and would allow detection of which rule had been successful.)

The 'previous colour' rule sends 'batch_clr_itms' a 'copy message' ('if copy batch_clr_itms then set batch_best_rule'). The method looks for the single colour with a batch size of one, set for the extended colour batch. If this method succeeds, the batch_manager class receives the identity of the colour (and singleton batch size).

The rules 'crisis valid' and 'crisis fail' send a 'valid' message to 'batch_clr_itms' with selector of 'plus'/'minus' respectively ('if valid batch_clr_itms for ...'). The method invoked then looks for a non-'high_value' for 'plus' (or a negative a.p. for 'minus') and finds the smallest (/greatest) value to determine the most favoured colour. In the case of a 'covered' BIW ('plus') the singleton batch size triggers calculation of a positive a.p. (and 'high_value' for nil batches) for comparison. The action is to cut the expression.

The (first) action of 'selecting best' is followed by updating of the task state (to 'done'). The control loop of expression of the class task_rules can now be terminated, because this variable now has the value for quitting (see Chapter 6).

The final action of the loop-containing method ('load task_manager', implemented by 'task_allocator') is to reveal the decision, by sending an 'enquire' message to the variables of class 'batch_manager', to state the colour and batch size chosen. It then packages these values in a string and exits. The return value for the method is then available outside the object-centred part for processing.

7.3 How does the Istel algorithm compare?

Given the order cover and sequences as in Figures 7.1 to 7.3, the object-centred solution and the original algorithm's implementation can be compared in the following discussion. (The implementation of Istel's algorithm contains the principles of paint scheduling as described in Chapter 4, Section 4.1). In tests, as the rule base was developed, the behaviour of the KBS was found to mimic that of the Istel algorithm implemented in the simulator. (The actual steps by which data files are updated etc. are not identical, but for each processing stage the same batches were produced given the same input sequence and body/colour profiles.

The colours are considered, in turn, from 1 to 12 (white to champagne). Any that are 'off' are discarded (represented in the **SIMULA** program by a 'c' suffix rather than '+'). In this case, none is constrained. The next principle applied is to form the largest possible batch for each colour, by checking the sequence against the cover profile for bodies and colours. This means that only scarlet is possible to form a valid batch (see Figure 7.4) because all other colours either have one BIW covered (e.g. bronze) or two (e.g. silk green), obeying the principle that batches of less that minpb bodies should be ignored. Given these potential batches (nil, or six for scarlet) the algorithm now checks for prioritisation. In this case no body/colour combinations have been defined as priorities. The prioritisation principle is applied as the batch is formed. The formation of the largest possible batch, subject to the maxpb limit, obeys the batch limiting principle (that no batch larger than maxpb for the colour be scheduled unless as a default.) The next principle to be applied is that of choosing the colour on the basis of relative achievement. The formula (batch size

+ achievement) / requirement is used to compare any non-nil batches. This means that an a.p. is calculated only for scarlet and the 'least achieved' principle means that this colour is selected for a batch size of six. (Again, the nil batch colours have a high value set for their a.p., i.e. 200000).

Where default allocation is required (but not in the above example), the first principle is to try and extend the previous batch, where this is of smaller size than maxb. In this situation, cover is checked for the leading BIW (i.e. code 2: 200 5 dr) for the previous colour. Again, this would be assignable, since this body type is available in all colours.

The other principles stated (for crisis) are somewhat informal, in that they attempt to provide an automatic response in situations where the scheduling program does not select a batch. The operator would normally be expected to supply a decision. For a 'valid cover' decision, any colour that has cover for the leading BIW is marked for a singleton batch and an a.p. generated: (achievement + 1) / requirement for the respective colour, and the lowest value of these is chosen. The last default attempt is to calculate the a.p.s for all available colours (i.e. irrespective of cover for the leading BIW) and take the lowest a.p. for a singleton batch.

7.4 The effect of inappropriate rules

To illustrate the difference that an inappropriate rule can make to the (object-centred) processing, assume that there is one slight change to the details as given in Figures 7.1 to 7.3. In this case, the colour chosen (scarlet) is, say, unavailable. The effect should be that the 'normal' batch rule fails for all colours and the 'extend batch' strategy comes into play. In this case the next BIW on the line can be painted in any colour and so the previous batch will be extended (unless the maxb limit has been reached; in which case, the 'crisis valid' rule would be applied. The singleton batch would be chosen on the basis of the a.p.s shown in Figure 7.5.)

Colour	Achievement proportion (a.p.)
arctic white	(88 + 1) / 419 ≈ 0.212
midnight black	(66 + 1) / 249 ≈ 0.269
scarlet (c)	-
azure	(135 + 1) / 496 ≈ 0.274
lemon	(28 + 1) / 245 ≈ 0.118
bronze	(107 + 1) / 439 ≈ 0.260
sea green	(50 + 1) / 294 ≈ 0.173
silk green	(110 + 1) / 404 ≈ 0.275
beige	(82 + 1) / 419 ≈ 0.198
sunset orange	(88 + 1) / 336 ≈ 0.265
silver	(124 + 1) / 493 ≈ 0.254
champagne	(164 + 1) / 608 ≈ 0.271

Crisis allocation

Figure 7.5

Lemon would be selected for the leading BIW on the line. Assume, now, that there were an incorrect rule, e.g. 'colour off' of batch_min_rules had been mis-stated *viz:* 'for_all colours if enquire code of current_item of colours *greater*_than nil then set batch of current_item of batch_clr_itms to nil and update code of current_item of batch_clr_itms to minus'. The change does not affect the interpretation of the task rules, but in 'batch_min_rules' all colours that are constrained are effectively disregarded and the (vital) setting of the batch_clr_itms instance variable for 'code' for scarlet is left as it was (previously available). The other colours are all marked as 'off'. This now means that 'scarlet' is now selected for a batch of six BIWs, just as before. No other type of allocation (than 'normal') would be possible, in any case, because the colours had all been marked as unavailable. If all colours were 'off', in this case, the rules would act 'correctly' and state that no colour were possible (the availability test being applied several times during rule expression).

7.5 Extended uses of the rule-based formalism

The original paint scheduler has been shown to be potentially lacking in flexibility in response (where, for example, the input is random or very non-random, see Chapter 5). It may be useful to express other constraints. A good example is a drop in paint quality for a few BIWs painted in a new colour, i.e. where the old colour masks the new, e.g. yellow after green or white after black. This would be expressed as an increase in rectification (not represented in the program) but effectively meaning that the efficiency of the a.p. as a criterion drops. This is because the smaller a batch is in the new colour, in these cases, the more it is subject to a high degree of decreased quality and, although its a.p. is lowered (favouring its selection) it should actually be a relatively less favourable decision. This is particularly hard in terms of some cost/benefit determination (such as a.p.) because external knowledge of the proportional decrease in quality is not available to the scheduler. Some other mechanism would need to be found, or an independent modification of the a.p., following determination, should be done to bias the decision away from the undesirable colour.

Extension of the objects and rules to accommodate such new knowledge would, preferably, not involve a major re-structuring. This can be done, but this is not always possible (see Chapter 8). The following example is given to show that a major strategic change can be effected within the existing framework.

The first step is to define a new rule class ('alter_a_p_rule') using the object editor. This states how a.p.s can be adjusted. It requires a sub-class ('a_p_coefficient') which has instances for each of the colours defined (e.g. for 'colour_manager'). For each of these instances, if a combination of old/new colours to be avoided is required, then for each of the *new* colours a set of sensitivities to *old* colours is defined. A coefficient to moderate the a.p. by is then stated (e.g. for multiplying the existing value by) for each combination. A 'valid' method is defined for the class, with a corresponding implementation by an effector ('moderate_a_p').

The code for the method works in a similar way to other ones in that it expects the instance required to be supplied. Say that 'azure' is being checked and no combinations for this colour with a previous colour are constrained. The method will find no combination and report a "FAIL". If a sensitivity is listed, it checks the previous colour ('Prevcol') for a match. If one is found, the coefficient is evaluated and the corresponding instance of 'batch_clr_itms' is sent a message 'enquire'-ing the value of a.p. determined. The a.p. derived is then transformed by the coefficient and the value is returned to the batch_clr_itms instance ('set') to update the effective a.p. The preferred way of doing transactions is to define an 'enquire' method to allow a rule to find out if a combination has been specified.

The definition of 'alter_a_p_rule' is as a rule set. In this case, one instance is required. This can be defined using the object editor (or the meta-description editor). Having defined a 'set' for the class, the rule 'if enquire alter_a_p_rule for current_item then set alter_a_p_rule for current_item' can be adduced.

The second part of the re-definition is to change two task_rules slightly and introduce a new rule: 'selecting batch' should now update the task state to a new value ('change_a_p') and the new rule should mimic its siblings, in calling for interpretation of a (new) rule class: 'if enquire task_manager returns change_a_p then quantify alter_a_p_rule by fire_all and update task_manager to selecting_best'. Now, batches formed can be moderated by considering the desirability of allowing a new batch in a colour sensitive to that of the old batch.

This is a quick way of implementing a moderation in an independent instance (of batch_clr_itms) without changing the existing objects to a large extent. ('Alter_a_p_rule' would probably be best defined as a subset of the topmost class - task_manager.) The rule stated should be used in 'fire_all' mode because, in practice, more rules than one would probably be adduced to provide a more detailed type of moderation.

Chapter 8 Discussion and summary

This final chapter is concerned with evaluation of the work and possible directions for future research. The initial project was to construct a representational scheme capable of supporting expression of the rather complex and disparate knowledge available of a particular control process. The result was a set of programs and a technique for structuring knowledge representation. The work will be discussed in this light. The scope of the research covered many areas of interest in Artificial Intelligence, although essentially an issue of knowledge representation. Many aspects arising would be starting points for whole lines of new research.

8.1 Results of the research

There are two main parts to the work. The first is an examination of an existing solution to a real manufacturing problem. The second presents an alternative approach to this problem, and, in the light of other work, as discussed in Chapter 2 and Section 8.2 below, may indicate a new way of handling such issues.

Istel's solution to the problem of the flexible paint-spraying requirement was assessed in Chapter 4 and the programs written to support the analysis suggested in Chapter 5. The theoretical analysis suggested that the algorithm might be based on an inappropriate assumption: that ideal behaviour results from the use of achievement proportions as a criterion in selecting a colour.

A simulation of the traffic on the conveyor line through the painting booth was created. This was able to record the changes in the state of the line and implemented the colour allocation algorithm as stated (Istel plc 1985). The control console of the factory was simulated, allowing the maintenance of a database of order requirements and control over the status of system variables. An analysis of the effect of possible types of input to the paint booth was carried out using a suite of programs (controlled by Unix scripts). The simulation was of a range of body sequences entering the paint booth, where the degree of randomness of different sequences was varied. The results of processing such sequences were recorded for a set of order banks of a range of degrees of freedom (of permissible body/colour combinations).

The results of the tests on processing different sequences were compared to evaluate the likely performance of Istel's algorithm under conditions of flexible response. The analysis suggested that the algorithm might give acceptable performance under limited types of input, but in general did not behave ideally, as mentioned in Istel's specifications. In particular, a sensitivity to some forms of input was significant, producing a (presumably) unacceptable level of singleton batches (i.e. failures of the scheduling algorithm).

This part of the work shows that the algorithm may not behave as it would be expected to under all conditions for use and its flexibility may be less than desired.

The second part of the work investigated the possible uses of AI techniques to capture the disparate knowledge available for paint scheduling. The architecture devised to handle the problem uses an object-centred scheme for a knowledge-based system which can fit inside an application and be invoked from it. A toolkit to facilitate knowledge representation was constructed, which supports the definition and modification of an object system. A utility to aid the definition of properly constituted rules was written and can assist a user towards stating the appropriate relationships between objects in the domain to form logical rules of behaviour of those objects. The form of rules accepted complies with the syntax and semantics of rule interpreters provided by the tooolkit. Using the formalism supported by the toolkit, the given knowledge for scheduling was captured in an knowledge-based system which was able to perform the paint scheduling task. The toolkit supports the tracing of activity within the knowledge base. The object-centred system appeared to be extensible, allowing new knowledge of scheduling to be added. This could provide a significant advantage of the formalism over the original conventional scheduler in long-term factory use.

The work on representation of knowledge shows that a solution to a reactive scheduling problem, such as that of paint allocation, can be found using AI techniques. The formalism derived to handle such fragmented, and sometimes strategically contradictory, knowledge appears somewhat unusual (when compared with more 'traditional' knowledge-based systems). There are several reasons why the approach taken was different from the more conventional approach (e.g. using an AI shell. The potential need to change allocation strategies, as more operating realities become apparent, encouraged the encapsulation of knowledge in discrete units. (This could assist the removal of some and addition of other new units, without drastic changes in the control structure of the program.) Object-centred schemes offer advantages in this direction. The need to capture and manipulate knowledge relating to changes in the state of the system means that much knowledge is essentially procedural. The way objects and messaging in this formalism are used means that the 'core' (i.e. mediating expression of knowledge) procedures to change states (or views of what is being done) need not seem complex because they are mediated by traceable and inherently comprehensible messages. The intention was to allow a context to be defined which would then control the expression of the (other) knowledge associated with it. The toolkit was written in a widely-used language and it was not necessary to use a specialised AI environment for the development and running of the scheduler.

The domain of reactive scheduling, where a complex behavioural effect (e.g. even production of cars) is controlled at one process stage (e.g. painting) is significantly different from other types of industrial scheduling (e.g. planning for shift production at one time per shift). The reactive scheduler must be 'aware' of states outside its immediate location . An 'isolated' (non-reactive) scheduler could, for example, have a fixed list of requirements, supplied from another part of the overall process, and act on the basis of those. The list would specify, in this case, the batches to be painted and in which order. All it would do would be to examine the number of items that had passed it since the list arrived and change the colour as each batch ended. If the colour became unavailable, it could do nothing, except sound an alarm, for example. To do more, sensibly, knowledge of what is best for other parts of the process must be recognised by the scheduler. In this way, the paint scheduler discussed in this thesis is rather different type of knowledge-based system from controllers such as PIDs which close a control loop using stored knowledge. For such PIDs, the knowledge is only of variables in their immediate domain. The cement-kiln controller of Haspel (1985) seems similar in this respect to such PIDs. (Feedback is an important determinator for later behaviour.) The opposite end of the 'scheduling' spectrum is represented by planners, e.g. that described by Fox *et al.* (1982). Here, the scheduling is done, *a priori*, to optimise activity happening later. The use of AI techniques is well established in this area.

The novelty of this paint scheduler derives partly from this use of AI in CIM. The way this object-centred scheme embodies knowledge-based systems techniques seems different from how other knowledge-based systems for FMS (and industry generally) allow the use of objects. (Rules are true objects, having no prescribed, large, interpreter, whose expression is mediated solely by message-passing. Definition of new procedures or predicates, in most systems, puts the knowledge embodied in such code *outside* the framework for representation. This means that the understandability of the new knowledge is that of code. In the tool-kit the behaviour of such a new item - as an invoked method - could be traced and would probably consist, largely, of messages, which have meaningful contents.)

Another important aspect of this work is the attempt to evaluate the worth of using a particular technique. It may be an interesting exercise to implement a knowledge-based system, but part of the knowledge engineering process should be to evaluate the characteristics of a problem so that the relevance of any technique used can be assessed. This was the motivation for the analysis and testing of Istel's algorithm, which was an empirical piece of work. In this case, the results supported the initial contentions and point to possible alternatives to the original solution which could give improved performance (over the Istel algorithm).

8.2 Process control knowledge and its representation

The main problem with the knowledge available from Istel's documentation seems to be commonly expressed in the AI literature; namely, that it is incomplete, inaccurate and often inconsistent. Feigenbaum (1979) mentions these difficulties in the context of 'knowledge engineering'. The problem studied is typical in these respects. The statement of the scheduling approach is certainly incomplete.There seems to be indication in the documentation that the operating circumstances may not really be quite as stated. There are inconsistencies in the principles of scheduling as stated. The following paragraphs seek to clarify these remarks.

A justification for the statement that the knowledge is incomplete is that the responsiveness of the algorithm (Istel plc 1985) is to many things that are not explicitly represented. One example is the batching parameterisation. The intention is clearly to bias things so that batches are generally of a certain size. This is not actually what is explicitly represented (in the algorithm). No cost/benefit is attached to making a batch of one particular size, as against another in the same colour. If a batch of *n* BIWs is highly desired, then batches of sizes $n + 1$ and $n - 1$ should be less desirable. This is not stated - any effect eventually producing such a state of affairs is a side effect of the batching process, not a clear choice. The minpb/maxpb 'desired' range of batch sizes just biases the batch in favour of maxpb. It is the comparison of achievement proportion that decides which colour, and therefore what batch size, is chosen. The relative importance of batch size and colour choice are thus commingled. This point is taken up again later.

The comment about inaccuracy stems from the above. If the minpb/maxpb values, for example, are principally *colour* parameters, to guarantee the correct size for batches due to colour constraints, for quality etc., then they should not be set so as to produce batches of a particular size if these values do not reflect colour constraints. It has been suggested that one use of maxpb is to ensure that, say, an average batch size of twenty BIWs is scheduled. (It is likely that the best batch size, from the point of view of paint quality, is much greater.) Other inaccuracies are introduced by the use of a.p.s, as discussed in Chapter 4. It is clearly a bad thing to stop a less popular colour being painted, for reasons of small demand, when it is at least as under-painted as a more popular colour, simply because a small batch, in the less popular colour, would be required on the basis of having to have a small (favourable) a.p. to be selected. This happens because when two colours, a relatively rarely ordered one and a popular one, are alternatives, then unless the former has a small size of maxpb compared with the latter, where a large batch of either could be allocated, the latter will tend to be selected because it is much less achieved, even when the ratio of achievement:requirement for both is similar - the batch size is then the dominant factor. The calculation of (batch + achievement) / requirement favours the colour whose absolute value for requirement is larger. The result of this situation can be a 'reflux' state where some body types in some colours never get painted - being 'difficult' to achieve.

The 'inconsistency' is highlighted by such considerations. If the program is to be flexible and ensure even, i.e. proportionate, production across colours, then 'priority' rules go right across notions of flexibility; a flexible algorithm should not require a prioritisation capability, which, indeed, reduces the degree of flexible response. Other inconsistencies lie in the way in which one strategy is applied, e.g. painting 'properly-sized' batches of the most favoured colour, followed by another, i.e. crisis allocation, should the first fail. The apparent intention is to find the 'best' decision within the limits of the resolution pertaining. Many criteria obtain, e.g. batch size, paint quality, production balancing, or order profile, but they are not combined explicitly with some attributable cost/benefit function.

Given these difficulties then, one purpose of the research was to find a mechanism for representing the knowledge, even if somewhat incoherent in statement, as in this case. This was deemed necessary for a general approach to handling such manufacturing problems, for which this case is seen as a good example in the rather under-examined but significant area of flexible *reactive* process control. The issues of flexible response and 'adjustment' of knowledge to reflect new insights, without requiring major restructuring, were central for any scheme adopted. As was discussed in the early chapters of this account, there is a special need in representing knowledge for this type of domain and this concern has been reflected in the research: in a manufacturing environment, the knowledge is typically of changes in the state of the system, which are inherently action-based and non-defeasible (non-reversible) in a logical sense. The changes cannot be retracted once made. Thus, the component parts of a knowledge base must be sensitive to context and must also be allowed to be modified permanently. This means that the search for a solution is increasingly constrained and does not backtrack. This led to the use of forward reasoning and forward expression (data-driven rather than goal-driven) of rules. Another aspect of the knowledge is that it is 'bitty': a set of 'independent' laws of scheduling, which do not 'fit' into each other, has been adduced. Somehow, there must be a connection between them. This led to two distinct features in the architecture - autonomous items responsible for their own maintenance, as 'objects'; and rules, stating some effect which is achieved, which cannot easily be chained in a logical progression by the implicit action of an interpreter. The object-centred scheme and the expression of rules, mediated by message-passing, developed naturally from these constraints.

An important feature of the object-centred design is the simplicity of the programming paradigm. Once the differences between it and more conventional programming styles are appreciated, it is simpler way of structuring data and writing code. The toolkit is not intended for use by a novice, since, currently, some writing of *C* code is required. A more competent, but still relatively inexperienced programmer should be able to write, test and debug applications easily. Any complexity involved in methods is more in the design of their objects and they, themselves, do not need to be intricate.

It was felt that a central requirement was for a simple and uniform representation, that would still allow extension. The use of methods and message-passing to mediate rule expression is a simple feature. It is, however, flexible. All rules used (which do reflect a wide range of knowledge types: control, declarative etc.) have been accommodated within the same syntactic framework. (It is easy to invent new styles for knowledge representation, where appropriate, without disrupting the existing scheme - new classes may simply define new methods for a different form of expression.) Conditions and actions are solely expressed by message-passing between known objects. The object structure really should reflect some model that the designer has envisaged and, therefore, should be amenable to comprehension, hopefully to all who examine it. The rules generally state relationships and could be described as very declarative. The main achievement of the system is in expressing procedural knowledge in an accessible manner. No special means is required to state such metalevel knowledge as control rules. These will, normally, be part of an class concerned with control of some group of objects and should not need any special features to be recognised as such. Simple methods probably have a a suitable 'grain-size' for procedural knowledge to be 'explicitly' understood. If rules express relations between objects of a declared nature, via those methods, then the procedural knowledge has been explicated. Procedural knowledge does not then have to reside in complex predicates manipulated by rules. This criterion, of explication, has been successfully addressed to a considerable degree.

Given the uniform and prescriptive nature of rules, and objects in general, a limited range of modes of expression have been used to implement the representation of manufacturing knowledge. Does this mean that the range of expressivity is limited to these? This question can be clarified a little by considering what the user is forced to do. The object editor only requires a framework for message-passing and inheritance to be stated - with the minimum storage to be reserved on initialisation. The user is at liberty to express certain parts of the application in a manner quite outside the object-centred paradigm. The extent to which rules express the knowledge is determined by the user: an example of this is the formation of a batch. It is quite possible to use a rule-based representation for this or to use a single method (i.e. totally procedural). The user can add new means of expression as desired. The flexibility of the system allows extra parts to be added in a much less restricted manner (in the sense of avoidance of harmful interactions). If the underlying classes have been defined appropriately then new knowledge that does not fit into the existing hierarchy can probably be assimilated into new classes.

There is an extra dimension to this in terms of 'engineering' a solution. The scheme makes for top-down design (due to its hierarchical structure) and there is a fair chance that models of a solution to a problem, represented in objects, are made to behave in a manner that mirrors the designer's view of their world relationships. Where inconsistencies appear, after definition of the structure, it is likely that either the model has been incorrectly transcribed into the application or the model itself is faulty. If the former case is true, then the tracing facilities for inter-object transactions, inherent in the architecture, may help to expedite resolution of the problem. When a model is incorrect, the likelihood is that certain objects will appear to behave at variance with their supposed activity, because the 'model' does not account for the required behaviour. The tight definition of objects as encapsulated items means that the potential for unexpected interactions, on addition of new items, is limited and, hopefully, less difficulty arises in understanding them when they do occur.

In terms of explication of knowledge, the metalevel description process to assist definition of rules is helpful. It forces a user to concentrate on exactly what objects mean in the representation in relation to each other. The way knowledge is stated, can to some extent be checked for consistency. If a rule *should* be correct but is rejected, for reasons other than that of unknown referents, there is evidence that the model underlying it is faulty.

8.3 Modifications to the toolkit and further work

There are two possible directions for future research and work on this implementation of an object-centred architecture. The first concerns general improvements to the programs constituting the toolkit; the second is to use the architecture in better ways to improve its expressiveness. It would be convenient to produce a 'similar' problem from the same (wide) domain of the application studied here and show that the toolkit could be used to produce an intelligent control system for it. This, however, does not give any but the most superficial evidence that the technique is effective in general. Rather than attempt to use a doubtful 'proof by analogy' a discussion is included to demonstrate the potential flexibility of the approach in handling the same problem in a very different way: namely trying to represent constraints as cost/benefit parameters (i.e. relative rather than absolute), called here 'utilities'.

8.3.1 Possible improvements to the toolkit

There are several areas where the toolkit could easily be strengthened and made more 'user-friendly'. These were omitted, not because they were time-consuming, but more because they would not assist demonstration of the feasibility of the scheme and because the toolkit was not intended to be in the required form for marketing, at this stage.

Perhaps the most obvious improvement is to allow references to system objects by name and not by some slightly obscure symbol, e.g. an index. As was stated, this had more to do with a quick implementation and testing of the speed of messaging than any other consideration. It would be better to allow the user to use the same names for classes etc. as are defined for the object editor. In association with this a small text-checking utility could be written to analyse the method source files and extract the messages from them to compare with known strings. This could be done as each method is defined for a class but would then require that the code be written prior to the definition of the class. If this were done, a companion utility could be added to amend identifiers, altered in the definition, in the accompanying methods.

Having said that it is a simple matter to do the above, some changes would have to be made in the messager. If, instead of a series of indexes and strings, the message consisted of one string stating the class, method and rest of the selector, the messager would have to do a little more work. The syntax would still be "method, class, ...", preferably separated by a comma or, for Smalltalk enthusiasts, perhaps, a colon. The messager would then have to look up the names found against a table of class indexes so as to get the necessary access. This is probably best done by some type of hash table of names, particularly if large numbers of identifiers are to be used. (This was not done for the similar purpose of understanding names in rule acquisition, simply to make the process easier, since this was evidently desirable in any subsequent re-organisation of messaging along these lines.) This leads to more difficult decryption of messages when class names are part of the tail of the selector. since names of different objects are not constrained to be unique when of a different 'type' (i.e. methods must have unique names, but a class and method may have the same name and instances of different classes may share a name). One way of doing this might be to insist that some special identifier be added to a class name when it appears in the selector tail so as to distinguish it from a possible instance of the same name (or even constant or variable). A better way of allowing this might be to adopt the mechanisms detailed in the next paragraph. These are concerned with making the objects more independent and secure. The remarks above apply also to removing the need for methods to resolve a class from an instance method. If the messager is allowed to evaluate the message to the point of resolving these two, then a standard form of selector can be created, for transmission to the supplying method, that leaves the method without the job of parsing this distinction.

On the issue of security and making the objects more self-contained from the viewpoint of external items, several changes could be made. At present, the object framework is a global structure, whereas it only need be visible directly to the messager. The definitions to establish it could be put together in the messager source file and not in scope elsewhere. Such aspects as a class needing to know its identity within the hierarchy or that of its parent could be changed. The definition of access functions (and globally available) linked into the program could make the system more tightly controlled. An examples of this is the searching by a class of its children. It does not know its identity. Its methods are informed of its code, by the messager, and when it wishes to make a reference to itself it calls a **self()** function (exported by the messager source) which returns its name, which can then be used in messages. To access its descendants it can call **child()**, with an argument to specify which one it wants. A **super()** mechanism would allow access in a similar manner to a parent. In this way, the code would not have to specify absolute names of objects that should not really be known to the class. To allow one class to identity itself to another, a mechanism used in Smalltalk could be helpful. The context of a method's invocation could be accessed by a method receiving a message (from another method), e.g. one class requesting another to carry out some action involving it could be enabled by the receiving class using a function, **context()**, to find out the sender of the message, so that it could then send appropriate messages. In all these cases it is the messager that knows the real identities and relationships involved and the code does not have to reflect this directly. One very important aspect is the possible improvement in legibility of methods afforded by these changes.

The nature of class variables is somewhat indistinct in this scheme. To make for more evident objects of this type, it would be possible to define, for the class 'pool', objects in a similar way to that for instances. With the above mechanisms for streamlining messaging, access to class variables would be easy and definition of class methods could be done at the same time for instances. Notwithstanding this, the view taken in this work is that class methods and variables are not a desirable feature, because a class is principally a template for instances and should do nothing except define its members. Creation of new instances is generally a feature of class methods, but often these are special methods. In this work it has been possible to avoid the separate definition of such objects and seems to make the system more comprehensible.

The facilities for rule tracing are somewhat restricted in nature. One improvement might be to make the rule trace interactive like the method tracing. That would allow explanation facilities to be used while the process ran. One feature that might be helpful would be to allow the user to request details of rules that did not fire or only rules that were fired. Another possibility would be to allow the trace mechanism to show which rules were consulted (for quantified rules applied to particular classes) or to trace the changes in certain variables as a result of rule activity. This latter extension would require a rather complicated approach to object tracing. It would probably be possible to establish a record for a variable or group of variables. This record would keep a history of states, which could then be consulted. The tracing potential is supplied by an added class ('trace_manager') which fits into the object hierarchy so it would be in 'contact' with the objects in question.

The issue of rule acquisition is important for this scheme. Some additional work has been done on freeing the syntax accepted. The intention was to allow slightly incorrect statements or less detailed rules to be accepted and interpreted into suitable rules. This of its nature is a subject beyond the scope of this work and more within natural language processing. The results that were obtained showed that the hierarchical nature of the system directed acceptance of rules, thereby reducing the potential ambiguity in understanding input.

An example of the sort of free syntax that could be handled was this type of statement: 'if colour not available then set colour off ....'. To allow this to be understood the string 'off' had to be defined and related to a particular variable, using the meta description system to define such associations. 'Colour' would be tentatively associated with 'colour_manager' and a search of its instance variables revealed that 'code' matched with the associations of the constant 'off'. Similarly 'available' was related to 'code' and the meaning could be inferred. The translation was into the internal form of the rule. This rule was felt to be of less value and meaningfulness than the more correct 'if enquire code of colour_manager returns ....', so this work was left incomplete. It would seem to belong with a more comprehensive tool which assisted the writing of methods given the exact relationships of objects and necessary items for methods to recognise.

8.3.2 More advanced and flexible solutions to control problems

One of the difficulties associated with the existing solution to the scheduling problem is the rigid way in which the constraints were stated. This meant that the rules and structure of the solution were somewhat convoluted. In this part, the limits to extensiblity of this solution will be discussed. An alternative scheme will then be discussed which seems more flexible and should be able to reflect knowledge gained in a more direct manner.

What sort of new principles, then, could not be expressed in the existing structure without major re-organisation? A realistic example might be the sensitivity of production to what part of the week it is at any time. It seems possible that certain types of cars should be produced more on one day than another. Some rules that could be expressed are such as: 'if it's Monday morning then don't paint yellow', or 'if it's after 3.30 on Thursday then paint 25% less red'. The expression of these might be via a method that could enter the time in a variable of a class, for the purposes of comparison, using system rules like 'if enquire time of dates less_than monday_pm then set code of yellow of colour_manager to minus' and 'if enquire time of Thursday of days greater_than 3.30 then update coefficient of red of modify_a_p to 0.25 by minus', respectively. The first rule sets the colour yellow to be unavailable, and could be inserted in a new class (like the coefficient modification example given in Chapter 7) to be evaluated as a task. The second example refers to the modification of achievement proportion as suggested in Chapter 7 for constraining bad sequences of colours, acting in the same way, but belonging to a different class. The first rule just updates the colour status in a similar way to the 'colour off' rule of the class batch_min_rules.

These examples of rules that can be assimilated look quite new in scope. They do add new flexibility to the system. This is not, however, as flexible as it seems. There is one aspect that the system, as a whole, neglects: because such knowledge has not been included. The deficiency is in the sensitivity to body order profile. A much more useful type of rule is one that selects for particular bodies and possibly colours as well: 'if it's Friday then don't paint yellow for monterey and don't paint 800 3 dr'. This type of rule is very hard to implement, but a reasonable one. It is difficult because the whole allocation process look primarily at colours and secondly at body types. One way to to express it would be to do the batching twice, the first time selecting those batches which break these new constraints and switching the appropriate colours off before applying the usual batching rules. This however means writing rules which constrain colours, but refer to batching decisions about including a BIW or not. Logically, a 'correct' translation of this new knowledge into rules acting in context, i.e. at the time of increasing the batch to include a particular BIW, should mean new rules, in form_min_rules and form_inc_rules, being written or new classes for them being consulted by these batching rules. It is hard to do this without making the new rules reference a new set of constraints (prohibited combinations etc.) which are not relevant to the existing rules. The result is a mess in terms of object style, since these new rules have to be expressed, or not, depending on yet another constraint - time, in probably a new class of objects. This is not a shortcoming of the object system, and the fact that these sorts of constraints can be expressed shows the innate strength and flexibility of the scheme. It highlights instead the limitations of using constraints to express relative preferences, which is, after all, what the a.p. is supposed to do. Note that in some of these examples given, the a.p. is not modified, rather some possibilities are simply excluded.

To explore the flexibility of my approach to handling complex decisions in a reactive manner in manufacturing, a different formulation of the problem can be made in terms of the direct attribution of the worth of an overall decision to contributory partial decisions. The full implementational details of how this use of 'utilities' would be made will not be given. Indications of the major aspects of using this technique will be discussed.

What I am suggesting is a series of aspects that might be considered in making a decision. These might be such factors as the likely degradation of paint quality with batch size, poor sequences of colour, recognition of orders that have been pending for a long time in the rolling schedule. These are all different in how they affect the colour and batch size chosen but should be related directly to each other in combining to influence a decision.

In the case of the conveyor, a possible scheme might be to consider the stations on the line, starting from that at the paint booth identification point. The algorithm appropriate is to build a batch, as before, one BIW at a time, but viewing the stations as having a set of attributes, e.g. the BIW present, possible colours to be assigned to the BIW at the station. The manner in which colour attributes are assigned depends on the propagation of evidence for and against a colour from one station to the next. I am advocating a set of criteria which allow the colour to be judged, e.g. non-availability would constitute very strong evidence against assigning a potential to the BIW to take the colour, and would be propagated from that point on. Instead of looking at a time for a colour batch, all colours would be considered as having a cost and a benefit for the BIW being considered. Thus all relevant evidence could be adduced as each component of the decision is made, and preferring one colour over another for a particular body type could be done explicitly. Context-sensitive control rules that modified evidence accumulated for colours could be applied at the point where they would be most relevant. It is not clear what rules stating references might be, but it seems likely that some numerical attachments for evidence would be made as each criterion came into play. The advantage is that explanations of components of decision would be available, since successful rules modify the evidence collected, and direct comparison could be made between evidence for different candidate decisions to see why one comes out more strongly supported than another. I envisage that no direct addition of values, plus and minus, as for confidence factors (Shortliffe 1984) would be made, but rather that similar evidence for different candidates would be adduced and the numerically stronger value would dominate. This leads to the difficult question of combinations of different types of evidence (see the discussion in Cohen 1985 for the obvious problems involved). It is likely that experience of the use of the automatic paint booth would enable rules to be stated to discriminate between more and less important evidence, thus using a combining system to distinguish similar alternatives. The advantage of the object-centred structure is that it facilitates association of like items in a mutually-controlled unit. This allows the user to add on bits of knowledge in a coherent manner.

It might be difficult for operators etc. to state the relevant values to bias decisions accurately. The possibility of generating these by examination of recorded examples exists. If a particular control scheme is decided upon and the values for utilities are unknown, then it would be possible to start with a situation where no preference was stated, except that absolute constraints had the same effect as described above in making certain decisions invalid. The system would be tried on various known cases and the batching decisions made on the basis of the overall calculation that the model is set up to use. The result of the decisions could then be compared with the ideal that the model is supposed to produce. The utilities in each category could then be modified slightly so as to discriminate more between alternative colour and batch size decisions. If the changes produce a change toward better overall performance then they should be fixed. If they fail to produce a favourable result, then they should be undone and another set of changes tried. By repetition of these changes, possibly more suitable values for utilities could be arrived at. Since there are likely to be more than a few categories of utilities, it might be best to start with those that make the largest contribution to a decision and progress towards the more 'finely-tuning' ones later. If this trial and error were not to stabilise the values, or if it were, but they then could not be used for other examples, something important would be missing from the model; alternatively the model would be faulty.

The above example of how to produce flexibility depends on the correct modelling of the problem. The architecture described in this thesis is capable of representing many different types of knowledge and seems inherently quite extensible. It is simple to use and capable of capturing and handling the various contexts essential for a knowledge-based approach to writing applications for control in manufacturing.

Abbreviations

ACM	*Association for Computing Machinery*
AI	*Artificial Intelligence*
BCS	*British Computer Society*
Coll	*Colloquium*
Conf	*Conference*
IEE	*Institute of Electrical Engineers*
IEEE	*Institute of Electrical and Electronic Engineers*
IFAC	*International Federation for Automation and Control*
IJCAI	*International Joint Conference on Artificial Intelligence*
Int	*International*
J	*Journal*
NCAI	*National Conference on Artificial Intelligence*
Proc	*Proceedings*
SIGART	*Special Interest Group of the ACM on AI*

Aikins J.S., 1979

'Prototypes and production rules: an approach to knowledge representation for hypothesis formation', *Proc of the Sixth IJCAI*, 1-3.

Aikins J.S., 1983

'Prototypical knowledge for expert systems', *Artificial Intelligence* **20** 163-210.

Avouris N.M., van Liederkerke M.H. and Argentesi, F. (1988)

'An intelligent system for management of chemical emergencies', *First European Conf on Information Technology for Organisational Systems* 990-6, May, Athens, Greece.

Barbuceanu M., 1987

'Integrating declarative knowledge programming styles and tools in a structured object AI environment', *Proc of the Tenth IJCAI*, 563-8 Milano, Italy.

Barclay-Adams J., 1984

'Probabilistic reasoning and certainty factors', in Buchanan B.G and Shortliffe E.H. (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.*.

Bernstein M., 1987

'Finding heuristics for flowshop scheduling', in *SIGART Newsletter* 99, 32-3, January.

Bobrow D.G. and Stefik M., 1983

The LOOPS Reference Manual, Xerox Palo Alto Research Center, CA USA.

Briot J-P. and Cointe P., 1987

'A uniform model for object-oriented languages using the class abstraction', *Proc of the Tenth IJCAI*, 40-3, Milano, Italy.

Brown J.S., Burton R.R. and deKleer J., 1982

'Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II and III', in Sleeman D. and Brown J.S. (eds.), *Intelligent Tutoring Systems*, Academic Press, London, UK.

Buchanan B.G. and Feigenbaum E.A., 1978

'*DENDRAL* and meta-*DENDRAL*: their applications dimensions' *Artificial Intelligence* **11**, 5-24.

Buchanan B.G and Shortliffe, E.H., 1984

Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project, Addison-Wesley, London, UK.

Buchanan B.G., Smith D.M., White D.C., Gritter R.J., Feigenbaum E.A., Lederburg J. and Djerassi C., 1976

'Applications of Artificial Intelligence for chemical inference 22.Automatic rule formation for Mass Spectroscopy by means of the meta-*DENDRAL* program',*J American Chemical Society* **88**, 6168-78.

Bundy A., Silver B. and Plummer D., 1986

'An analytical comparison of some rule-learning programs', *Artificial Intelligence* **27**, 137-81.

Chester D., Lamb L. and Dhurjati P., 1984

'Rule-based computer alarm analysis in chemical process plants', *Proc of the Seventh Annual Conf on Computer Technology, MICRO-DELCON*, 122-9.

Clancey W.J., 1979

'Tutoring rules for guiding a method dialogue', *Int J of Man-Machine Studies* 79, 25-49.

Clancey W.J., 1983

'The epistemology of a rule-based expert system - a framework for explanation', *Artificial Intelligence* **20**, 215-51.

Clancey W.J., 1985

'Heuristic classification', *Artificial Intelligence* **27**, 289-305.

Clancey W.J. and Letsinger R., 1981

'NEOMYCIN: reconfiguring a rule-based expert system for application to teaching', *Proc of the Seventh IJCAI*, 829-36.

Cohen P.R., 1983

'Heuristic reasoning about uncertainty: an artificial intelligence approach', *PhD.Thesis*, Stanford University, CA, USA.

Cohen P.R. and Greenberg M.R., 1983

'A theory of heuristic reasoning about uncertainty', *Artificial Intelligence Magazine*, 17-24, Summer.

Collinot A. and le Pape C., 1987

'Controlling constraint propagation', *Proc of the Tenth IJCAI*, 1032–4, Milano, Italy.

Cooper D.W., 1984

'*TIMM*: The Intelligent Machine Model', *Proc of the IEEE Aerospace and Electronics Conf, NAECON*, Dayton, Oh, USA.

Corkhill D.D., Gallagher K.Q. and Murray K.E., 1986

'GBB: a generic blackboard development system', *Proc of the Fifth NCAI*, 1008-14, Philadelphia, PA, USA.

Cox B.J., 1986

Object-Orientated Programming: An Evolutionary Approach, Addison-Wesley.

Davis R., 1977

'Meta-level knowledge: overview and applications' *Proc of the Fifth IJCAI* 1-3.

Davis R., 1979

'Interactive transfer of expertise: acquisition of new inference rules', *Artificial Intelligence* **12**, 121-57.

Davis R., 1980(a)

'Meta-rules: reasoning about control', *Artificial Intelligence* **15**, 177-222.

Davis R., 1980(b)

'Meta-rules: reasoning about rules', *Artificial Intelligence* **15**, 223-3.

Elleby P., Fargher H.E. and Addis T.R., 1988

'Reactive constraint-based scheduling', *Proc of the IEE Coll on Artificial Intelligence in Planning for Production Control*, May, London, UK.

Erman L.D, London P.E. and Fickas S.F., 1981

'The design and an example use of *Hearsay-III* ', *Proc of the Seventh IJCAI*, 409-15.

Evers D.C., Smith D.M. and Staros C.J., 1984

'Interfacing an intelligent decision maker to a real-time control system', *Proc of SPIE, International Society of Optical Engineering, 485 Applications of Artificial Intelligence,* 60-4, Bellingham, WA, USA.

Fikes R.E. and Nilsson N.J., 1971

'STRIPS: a new approach to the application of theorem proving to problem solving', *Artificial Intelligence* **2**, 189-208.

Fox M.S., Allen B. and Strohm G. 1982

'Job-shop scheduling An investigation into constraint-directed reasoning' *Proc of the First NCAI*, 155-8.

Fox M.S., Lowenfeld S. and Kleinosky P., 1983

'Techniques for sensor-based diagnosis', *Proc of the Eighth IJCAI* 158-63

Francis J.C. and Leitch R.R., 1985

'Intelligent knowledge-based process control', *Proc of the IEE Int Conf on Control*, 483-8, Cambridge, UK.

Gallanti M., Guida G., Spampinato L. and Stefanini A., 1985

'Representing procedural knowledge in expert systems: an application to process control', *Proc of the Ninth IJCAI*, 345-52, Los Angeles, CA USA.

Georgeff M.P., 1979

'A framework for control in production systems', *Proc of the Sixth IJCAI*, 328-34.

Georgeff M.P., 1983

'Communication and interaction in multi-agent planning', *Proc of the Second NCAI*, 125-9.

Georgeff M.P., 1986

'The representation of events in multi-agent planning', *Proc of the Fifth NCAI*, 70-5, Philadelphia, PA, USA.

Georgeff M.P. and Bonollo U., 1983

'Procedural expert systems', *Proc of the Eighth IJCAI*, 151-7, Karlsrühe FRG.

Gilmore P.C., Lawler E.L. and Shmoys D.B., 1985

'Well-solved special cases', in Lawler E.L., Lenstra J.K., Rinooy Kan A.H.G. and Shmoys D.B. (eds.), *The TRAVELLING SALESMAN PROBLEM*, Wiley and Sons, Chichester, UK.

Goldberg A. and Robson D., 1985

Smalltalk-80. The Language and its Implementation, Addison-Wesley, Menlo Park, CA, USA.

Gordon J. and Shortliffe E.H., 1984

'The Dempster-Shafer theory of evidence', in Buchanan B.G., and Shortliffe, E.H. (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project..*

Harris R. and Zinober A.S.I., 1988

'Practical microcomputer solutions of the cutting stock problem', *Proc of the IEE Coll on Artificial Intelligence in Planning for Production Control* May, London, UK.

Hasling D.W., Clancey W.J. and Rennels G., 1984

'Strategic explanations for a diagnostic consultation system', *Int J of Man-Machine Studies* **20**, 3-19.

Haspel D.W. and Taunton J.C., 1985

'Application of rule-based control in the cement industry', *Alvey Report Expert Systems Group*, 16-24.

Herrod R.A. and Rickel J., 1986

'Knowledge-based simulation of a glass-annealing process: an AI application in the glass industry', *Proc of the Fifth NCAI*, 800-4 Philadelphia, PA, USA.

Hewitt C., 1979

'Control structure as patterns of passing messages', in Winston P.H. and Brown R.H. (eds.), *Artificial Intelligence: An MIT Perspective*, The MIT Press, Cambridge, MA, USA.

Iline H. and Kanoui H., 1987

'Extending logic programming to object programming: the system LAP' *Proc of the Tenth IJCAI*, 34-9, Milano, Italy.

Istel plc, 1985 **

Technical Specification for the Paint Scheduling Sub-system, Istel plc, Redditch, UK.

James J.R., Taylor J.H. and Frederick D.K., 1985

'An expert system architecture for coping with complexity in computer-aided control engineering', *Proc of the IFAC, Computer Aided Design for Control Engineering*, København, Denmark.

Kämmerer W.F. and Allard, J.R., 1987

'An automated reasoning technique for providing moment-by-moment advice concerning the operation of a process', *Proc of the Sixth NCAI* 809-13, Seattle, WA, USA.

King R.E. and Karonis F.C, 1988

'Synergistic multi-level expert systems in computer integrated manufacturing', *First European Conf on Information Technology for Organisational Systems*, 1012-4, May, Athens, Greece.

Kunz J.C., Fallat R., McClung D., Osborn J., Votteri B., Nii H.P., Aikins J.S., Fagan L. and Feigenbaum E.A., 1978

'A physiological rule based system for interpreting pulmonary function test results', *Working Paper HPP-79-19*, Heuristic Programming Project, Department of Computer Science, Stanford University, CA, USA.

Kowalski R.A., 1979

'Algorithm = Logic + Control', *Communications of the ACM* **22** 424-35.

LeClair S.R., 1986

'The application of artificial intelligence technology to process control' *Proc of the 1986 Rochester Forth Conf*, 125-33, Rochester, NY, USA.

Lenat D.B., 1979

'On automated scientific theory formation: a case study using the AM program', in Hayes J.E. (ed.), *Machine Intelligence* **10**, Ellis Horwood UK.

Lenat D.B., 1983

'*EURISKO:* a program that learns new heuristics and domain concepts' *Artificial Intelligence* **21**, 61-98.

Lenat D.B. and Brown J.S., 1984

'Why *AM* and *EURISKO* appear to work', *Artificial Intelligence* **25** 269-94.

le Pape C., 1985

'*SOJA:* a daily workshop scheduling system', *Proc of the Fifth Technology Conf of the BCS Specialist Group on Expert Systems*, 195-211.

Mamdani E.H., 1982

'Rule-based methods for designing industrial process controllers', *Proc of the IEE Colloquium on applications of knowledge-based or expert systems* , London, UK.

Mark W.S., 1977

'The reformulation approach to building expert systems', *Proc of the Fifth IJCAI*, 329-35.

Michalski R.S. and Chilausky R.L., 1980

'Knowledge acquisition by encoding expert rules versus computer induction from examples: a case study involving soybean pathology', *Int J of Man-Machine Studies* **12**, 65-87.

Mitchell T.M., 1977

'Version space; a candidate elimination approach to rule learning', *Proc of the Fifth IJCAI*, 305-10.

Mitchell T.M., Utgoff P.E., Nudel B. and Banerji R., 1981

'Learning problem-solving heuristics through practice', *Proc of the Seventh IJCAI*, 127-34.

Neches R., Swartout W.R. and Moore J., 1985

'Explainable (and maintainable) expert systems', *Proc of the Ninth IJCAI* 382-9.

Nelson W., 1982

'*REACTOR*: an expert system for diagnosis and treatment of nuclear reactor accidents', *Proc of the First NCAI*, 296-301.

Newman P.A. and Kempf K.G., 1985

'Opportunistic scheduling for robotic machine tending', *Second Conf on Artificial Intelligence Applications, IEEE Computer Society*, 168-73, Miami Beach, FL, USA.

Odette L.L. and Dress W.B., 1987

'Engineering intelligence into real-time applications', *Expert Systems* 4 228-39.

Park J., 1986

'Toward the development of a real-time expert system', *Proc of the 1986 Rochester Forth Conf*, 133-43, Rochester, NY, USA.

Rieger C. and Stanfill C., 1980

'Real-time causal monitors for complex physical sites', *Proc of the First NCAI*, 215-7.

Rubinoff R., 1985

'Explaining concepts in expert systems: the CLEAR system', *Second Conf on Artificial Intelligence Applications, IEEE Computer Society* 416-21, Miami Beach, FL, USA.

Sacerdoti E., 1975

'The non-linear nature of plans', *Proc of the Fourth IJCAI*, 206-14.

Schefe P., 1980

'The Fuzzy Set Fallacy' , *Proc of the AISB Conf on A I.*

Schmucker K.J., 1986

An introduction to object oriented Pascal, Heydon.

Shaw M.J. and Whinston A.B., 1985(a)

'Automatic planning and flexible scheduling: a knowledge-based approach', *IEEE Int Conf on Robotics and Automation*, 890-4, St. Louis MS, USA.

Shaw M.J. and Whinston A.B., 1985(b)

'Task bidding and distributed planning in flexible manufacturing', *Second Conf on Artificial Intelligence Applications, IEEE Computer Society*, 184-9, Miami Beach, FL, USA.

Shortliffe E.H. and Buchanan B.G., 1984

'A model of inexact reasoning in medicine', in Buchanan B.G and Shortliffe E.H. (eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project.*

Shortliffe E.H., Davis R., Axline S.G., Buchanan B.G., Green C.C. and Cohen S.N., 1975

'Computer-based consultations in explanation and rule acquisition capabilities of the MYCIN system', *Computers and Bio-medical Research* 8, 303-20.

Sloman A., 1985

'Real-time multiple-motive expert systems', *Proc of the Fifth Technology Conf of the BCS Specialist Group on Expert Systems*, 213-24.

Smith D.M., 1983

'Industrial applications of artificial intelligence', *TI-MIX National Symposium.*

Spruell J.A., 1981

'Computer system for automatic paint and process control', *TI-MIX National Symposium.*

Sullivan M. and Cohen P.R., 1985

'An endorsement-based plan recognition system', *Proc of the Ninth IJCAI*, Los Angeles, CA, USA, 475-9.

Swartout W.R., 1981

"Explaining and justifying expert consulting programs', *Proc of the Seventh IJCAI*, 815-23.

Swartout W.R., 1983

'XPLAIN: a system for creating and explaining expert consulting programs' *Artificial Intelligence* **21**, 285-325.

Sweickert R., Burton A.M., Taylor N.K., Corlett E.N., Shadbolt N.R. and Hedgecock A.P., 1987

'Comparing knowledge elicitation techniques: a case study', *Artificial Intelligence Review* **1**, 245-63.

van dyke Parunak H. and Irish B.W., 1985

'Fractal actors for distributed manufacturing control', *Second Conf on Artificial Intelligence Applications, IEEE Computer Society*, 653-60 Miami Beach, FL, USA.

Velthuijsen H., Lippolt B.J. and Vonk J.C., 1987

'A parallel blackboard system for robot control', *Proc of the Tenth IJCAI* 1157-9, Milano, Italy.

Weiner J.L., 1980

'BLAH, a system which explains its reasoning', *Artificial Intelligence* **15** 19-48.

Williams C., Allen B.P., Haley P.V. and Wright, J.M., 1985

'ART: the automated reasoning tool', *Proc of the First Annual Artificial Intelligence and Advanced Computer Technology Conf*, 77-82, Long Beach, CA, USA.

Winograd T., 1975

'Frame representations and the declarative/procedural controversy', in Bobrow D. and Collins A. (eds.), *Representation and Understanding*, Academic Press, New York, NY, USA.

Wright M.L., Green M.W., Fiegl G. and Cross P.F., 1986

'An expert system for real-time control', *IEEE Software* 3, 16-24.

Zadeh L.A., 1979

'Approximate reasoning based on fuzzy logic', *Proc of the Sixth IJCAI* 1004-10.

** for reasons of confidentiality the full title of this document has been withheld

Appendix A1 Results of paint batching: a part example

Example results of the first bias set of order profile #4 (16 × 16 matrix)

Results of replicate 1

135 batches: 0 singleton, 0 bodies NFP, 0 BIW bodies

Batch failure rate: 0.00

Colour batch details

Colour	Singles	Desired size	Large batches	Total bodies	Deviation from ideal
1	0	7	0	92	0.146
2	0	7	0	112	0.131
3	0	7	0	77	0.117
4	0	6	0	92	0.141
5	0	11	0	139	0.129
6	0	8	0	83	0.106
7	0	11	0	153	0.106
8	0	12	0	118	0.117
9	0	8	0	144	0.100
10	0	8	0	167	0.125
11	0	9	0	181	0.131
12	0	9	0	235	0.140
13	0	12	0	176	0.117
14	0	5	0	83	0.135
15	0	7	0	71	0.117
16	0	8	0	77	0.121

Results of replicate 2

139 batches: 12 singleton, 0 bodies NFP, 0 BIW bodies

Batch failure rate: 8.63

Colour batch details

Colour	Singles	Desired size	Large batches	Total bodies	Deviation from ideal
1	0	7	0	87	0.138
2	7	9	0	104	0.122
3	0	6	0	67	0.101
4	1	7	0	87	0.133
5	0	10	0	131	0.122
6	0	8	0	89	0.114
7	2	13	0	175	0.121
8	0	6	0	111	0.110
9	0	10	0	165	0.114
10	0	7	0	175	0.131
11	0	8	0	172	0.125
12	0	8	0	195	0.116
13	0	9	0	178	0.119
14	0	5	0	66	0.107
15	2	7	0	99	0.162
16	0	7	0	99	0.156

Results of replicate 3

132 batches: 0 singleton, 0 bodies NFP, 0 BIW bodies

Batch failure rate: 0.00

Colour batch details

Colour	Singles	Desired size	Large batches	Total bodies	Deviation from ideal
1	0	7	0	65	0.103
2	0	9	0	120	0.141
3	0	6	0	84	0.127
4	0	7	0	84	0.128
5	0	12	0	124	0.115
6	0	9	0	92	0.117
7	0	12	0	171	0.119
8	0	6	0	130	0.129
9	0	7	0	155	0.107
10	0	12	0	172	0.129
11	0	10	0	168	0.122
12	0	8	0	229	0.137
13	0	9	0	178	0.118
14	0	6	0	75	0.122
15	0	5	0	77	0.126
16	0	7	0	76	0.119

Results of replicate 4

142 batches: 0 singleton, 0 bodies NFP, 0 BIW bodies

Batch failure rate: 0.00

Colour batch details

Colour	Singles	Desired size	Large batches	Total bodies	Deviation from ideal
1	0	11	0	86	0.136
2	0	8	0	112	0.131
3	0	8	0	76	0.115
4	0	5	0	79	0.121
5	0	9	0	136	0.126
6	0	7	0	93	0.119
7	0	12	0	165	0.115
8	0	8	0	118	0.117
9	0	9	0	173	0.120
10	0	10	0	152	0.114
11	0	9	0	148	0.107
12	0	10	0	232	0.138
13	0	12	0	172	0.115
14	0	4	0	82	0.133
15	0	9	0	92	0.151
16	0	11	0	84	0.132

Appendix A2 Class structures

A symbolic representation of the class structure is obtained from the listing facility of the *editor* and included below.

In the ascending (parental) direction the inheritance is shown. The lists of requests for the class are given next (in nominal order of code). The third item is the list of child classes. In the listing the names *LEAF* and *ROOT* signify lower and upper boundaries for the hierarchy. *ROOT* is a nominal proto-ancestor of all classes. No defined class is higher up than a child of *ROOT*. *LEAF* marks the fact that a class has no children. A structure may be defined which is then found to be deficient. New children may be added to classes and new parents defined, allowing portions of the structure to be kept. The action table of requests must then be re-defined. Class properties of requests, instances may be re-defined.

A2.1 Classes and their messages and encoded methods

Code Class

[0] *task_manager*

Entry point for colour allocation. Two management activities control updating of knowledge base image of database files and colour allocation process. One instance: *task* *agendum*, a state record for the use of tasking rules. Subclasses: [2] *task_rules*, the object class declaring the tasks recognised in colour allocation;

[3] *task_rt*, a class provided for future real-time aspects.

The request [0] *load* on [0] (=> [2]*task_allocator()*) is the entry point for colour selection (i.e. from *paint_batch()*). It causes external changes to be marked on the knowledge base image (of the database). [0] sends itself an *enquire* (=> [3]*task_stat_read()*) message to query the task agendum about the tasking state. It also sends *update* (=> [4]*task_allocator()*) to itself to change the record of task state (in the agendum). Various messages to other (sub) classes originate from [0].

[1] *db_manager*

Parent and control class of the database classes (i.e. map, colour, body and priority). It is used as a routing point to ensure that changes are made consistently across all the database images. This class has no instances currently.

Subclasses: [4] *map_manager*, the control point for the map file image;

[6] *colour_manager*, the control point for the colour file image;

[9] *body_manager*, the control point for the body file image.

load (=> [5]*db_mngr_load()*) causes initialisation of the knowledge base image of the database from its various files (with the message *ALL* - this method is used also, with a *DUMP* message, to signal saving of the current image state). Messages to achieve this are sent to the subclasses. *update* is recognised (=> [6]*db_mngr_update()*) and controls the updating of these classes in a similar fashion.

[2] *task_rules*

The three instances of this class determine the colour and batch selection activities. *interpret* is accepted and invokes an interpreter (=> [0]*call_rule_set()*) to examine them in turn *(vide infra)*.

[3] *task_rt*

This is a place-holder for real-time aspects of the tasking activity.

[4] *map_manager*

Has two instances for map file contents. The first is used as the scheduling map, whilst the second is used as a buffer for new (incoming bodies) to re-populate the map. The 'visible' region of the paint line is thus the first instance's contents.

Subclasses: [5] *map_rt*;

[15] *batch_manager*, the control point for the batching classes.

load (=> [7] *map_mngr_load()*) initiates (/saves) the class and *update* (=> [8] *map_mngr_update()*) allows an interface with the database map file sequence (actually the batch sequence file - the map is used by external routines). *enquire* (=> [26] *enq_map()*) reports the body code at a given position in the queue: if the message arrives at this class through inheritance (the target was a descendant of [4]) the meaning is taken as compare a specified body code for a match at the given position.

[5] *map_rt*

A place-holder for real-time aspects of map handling.

[6] *colour_manager*

Holds the image of the database file for colours. An instance is held of each colour defined, representing: colour code (as in the file, an integer with a negative value indicates a constraint, i.e. non-availability); the current batch size found so far in this invocation of colour selection; the number of permissible body/colour combinations for the colour; the last item is a floating point number to allow the achievement to be calculated.

Subclasses: [7] *colour_items*

[8] *clr_constraints.*

load (=> [9]*colour_mngr_load()*) is received from [1] and causes the loading (or dumping, according to the exact message) of details from the colour file. *update* (=>

[10]*clr_obj_update()*) is accepted also from [1] and the message contains details of what colour is to be counted down and by how much. This class writes directly on its subclasses [7,8] in loading and updating because the details entered change at the same time as those of the parent.

[7] *colour_items*

As above an instance exists for each colour. This class is used to hold batching parameters and production details: code; MINPB; MAXPB; MAXB; requirement and achievement values are recorded.

[8] *clr_constraints*

A place-holder for future attachments to colours.

[9] *body_manager*

This is the equivalent, for body details, of [6]. It is the control point for the priority file image and the parent for its controller ([12]). It has an instance for each of the defined body types which record the code.

Subclasses: [10] *body_items;*

[11] *body_colours.*

Control of the body file image is exerted by receipt of *load* messages (=> [11] *body_manager_load()*) from [1].

As for [6] the parent writes details on its child classes to keep their information up to date. *update* also from [1] (=> [11] *body_manager_load()*) uses the same effector but the form of the message is different and for *update* specifies the details to update.

[10] *body_items*

Has no subclasses and no requests of its own.

[9] manipulates it and it has an instance for each body type: code; total and priority requirements; total and priority achievements and the number of colour combinations for the code are detailed. A list of those colour codes follows these details.

[11] *body_colours*

As with [10], class[9] maintains this class. As [10] is loaded, the equivalent body/colour combination details are recorded in a list of instances. The external variable, *Tcol*, is the number of instances, i.e. the sum of the possible combinations. Each instance has details of colour and body codes; total and priority requirements and total and priority achievements.

No requests are handled by this class.

[12] *prty_manager*

This class contains two subclasses: [13] *prty_items*;

[14] *prty_subs*.

It, with its subclasses, forms the knowledge base image of the priority file in the database. It recognises *load* (=> [13] *prty_mngr_load()*) and *update* (=> [14] *prty_mngr_upd()*) arising from [9] which control the initialisation and upkeep of priority items.

[13] *prty_items*

The instances of this class represent the defined priority combinations. Details held are colour and body codes; requirement and achievement for the combination.

[14] *prty_subs*

A place-holder for future developments.

[15] *batch_manager*

Holds transient items for batching. Its instances are used to control batching. At any time their contents reflect items under current consideration. One is for the colour. The next shows the body type. A third marks the present position in the map of the body of interest. The fourth is used to indicate what type of batching is used (i.e. priority, crisis etc.).

Some subclasses are rule sets: [16] *batch_min_rules*;

[17] *batch_inc_rules*;

[20] *batch_best_rule*;

the others are batching details: [18] *batch_clr_itms*;

[19] *batch_clr_batch*.

load (=> [15] *batch_mngr_load()*) from [4] controls the initialisation of the class for batching decisions. *enquire* (=> [30] *enquire_cover()*) checks that cover exists for the current colour/body possibility. *best* (=> [31] *pick_batch()*), selecting the optimal batching decision, and *set* (=> [25] *batch_class_set()*), causing the expansion of MINPB batches arising from rule interpretation.

[16] *batch_min_rules*

A rule class which can be interpreted, which allows specification of the appropriate class for which the rules apply. On receipt of the appropriate message *quantify* (=> [1] *call_gen_rules()*) its instances are evaluated.

[17] *batch_inc_rules*

As for [16].

[18] *batch_clr_itms*

There are no subclasses of this class. Its instances are controlled by its parent, [15]. They contain details of colour items copied from the colour classes allowing, on each colour selection run, a temporary version of colour items which may be adjusted as the decision is made. Colour class items are changed on updating and are 'read-only' for allocation purposes.

[19] *batch_clr_batch*

As for [18], with instances reflecting other batching items.

[20] *batch_best_rule:* this and the other classes are for rules.

[21] *batch_prty_rule*

Rules of this class respond to the interpret message (=> [0] *call_rule_set()*) and the instances are interpreted without general application over another class.

[22] *batch_ext_rule*

As for [21].

[23] *batch_best_rule*

As for [21].

[24] *form_min_rules*

This and the next class specify optional rules to replace batching methods. Interpretation is as for [16].

[25] *form_inc_rules*

As for [16].

Appendix A3 Messages found in methods

Source		Request	Class	Effector	Argument
external	0.	load	task_manager	task_allocator	(various)
(0) call_rule_set			none		
(1) call_gen_rules			none		
(2) task_allocator					
	1.	load	db_manager	db_mngr_load	{ALL|DUMP}
	2.	update	db_manager	db_mngr_update	“class”,“batch”
	3.	enquire	[src] task_manager	task_stat_read	{0 |1 | 2}
	4.	interpret	task_rules	call_rule_set	“fire first”
	5.	valid	batch_manager	batch_result	“NOARG”
(3) task_stat_read			none		
(4) task_stat_upd			none		
(5) db_mngr_load					
	6.	load	map_manager	map_mngr_load	“ALL”
	7.	load	colour_manager	colour_mngr_load	“ALL”
	8.	load	body_manager	bdy_mngr_load	“ALL”
	9.	load	prty_manager	prty_mngr_load	“ALL”
(6) db_mngr_update					
	10.	update	colour_manager	clr_obj_update	“colour”,“batch”
	11.	update	body_manager	bdy_mngr_load	“colour”,“body”, “item”
	12.	update	map_manager	map_mngr_update	“batch”

(7) map_mngr_load				
12.	valid	*[self]*	objfile_open	"self,ALL"
(8) map_mngr_update		none		
(9) clr_mngr_load				
13.	valid	*[self]*	objfile_open	"self,ALL"
14.	load	batch_clr_itms	set_batch_bco	
(10) clr_obj_update		none		
(11) bdy_mngr_load				
15.	valid	*[self]*	objfile_open	"self,ALL"
16.	update	prty_manager	prty_mngr_upd	"colour","body", "batch"
17.	update	body_colours	bdy_clr_update	"colour","body", "batch"
(12) bdy_clr_update		none		
(13) prty_mngr_load		none		
(14) prty_mngr_upd				
18.	valid	*[self]*	objfile_open	"self,ALL"
(15) batch_mngr_load		none		
(16) set_batch_itms		none		
(17) batch_minpb				
19.	enquire	batch_clr_itms	enq_batch_details	"colour,0"

20.	enquire	batch_clr_itms	enq_batch_details	"colour,1"
21.	enquire	colour_items	enq_clr_const	"colour,1"
22.	valid	*[self]*	enq_map	"map,posn."
23.	enquire	*[self]*	enquire_cover	"NOARG"
24.	set	batch_clr_batch	update_cover	"colour,body"
25.	update	batch_clr_itms	set_batch_itms	"colour,batch"

(18) bdy_itms_cpy none

(19) bdy_clr_cpy none

(20) enq_batch_detail none

(21) quant_batch_itm

26.	enquire	colour_items	enq_clr_const	"current_item", "requirement"
27.	enquire	colour_items	enq_clr_const	"current_item", "achievement"

(22) enq_bdy_clr none

(23) update_cover none

(24) form_batch

28.	cover	*[self]*	enq_map	"colour,body"
29.	enquire	prty_itms	enq_prty_one	"colour,body"
30.	enquire	batch_clr_itms	enquire_cover	
31.	enquire	colour_items	enq_clr_const	"current_item", "minpb"
32.	enquire	colour_items	enq_clr_const	"current_item", "maxpb"

33.	set	*[self]*	set_batch_itms	“colour”, “proportion”,“value”
(25) batch_class_set		none		
(26) enq_map		none		
(27) batch_cb_update		none		
(28) enq_clr_const		none		
(29) set_batch_bco				
34.	copy	*[self]*	body_clr_cpy	“destination”, “current_item”
(30) enquire_cover		none		
(31) pick_batch		none		
35.	enquire	colour_items	enq_clr_const	“current_item”, “requirement”
36.	enquire	colour_items	enq_clr_const	“current_item”, “achievement”
37.	set	batch_manager	batch_class_set	“colour,value”
(32) call_cut		none		
(33) batch_result		none		
(34) set_batch_cut				
38.	set	batch_manager	batch_class_set	“cover,priority”

(35) enq_batch_one

39.	cover	*[self]*	enq_map	"position,1"
40.	enquire	batch_clr_itms	enq_batch_detail	

(36) enq_prty_itm

41.	enquire	batch_manager	enq_batch_one	
42.	enquire	batch_clr_itms	enq_btch_detail	"current_item", "colour"
43.	enquire	map_manager	enq_map	"0"
44.	enquire	map_manager	enq_map	"1"

(37) enq_batch_inst	none
(38) enq_prty_one	none
(39) objfile_open	none
(40) enq_bdy_item	none
(41) match_old_clr	none
(42) cover_leading	none

Appendix A4 Rules

A4.1 Syntax and the rule interpretation effectors

Two rule interpreters, which use the same syntax, are included in the o-c program. The rules they examine are in an encrypted form.

The message [5] *quantify* results in the interpretation of rules which are in the same form as those which [3] *interpret* examines but contain a prefix showing the range of quantification (a class).

Rules are in the form of a conditional part followed by an action statement (or statements). The effectors used (*call_gen_rules()* and *call_rule_set()*, respectively) may be used in one of four ways: evaluate one or all conditional parts of rules, fire one or all rules. The rules are always grouped in a rule set which constitutes a class, and therefore associated with a context.

Conditions are made up of one or more message expressions which evaluate to a boolean result. Conjunctions and/or disjunctions are allowed. The expressions may be a simple boolean proposition or be a comparison between two expressions. An expression is either a constant or a request whose return value is taken as the value for the expression.

Actions are, similarly, made up of one or more message expressions, conjoined by 'and'. If the conditional part evaluates to true the rules will be fired, as appropriate, by calling the action part. The requests in the expressions are made and their return value is taken as the value of the interpretation. Expressions are 'fired' and while their return is not a *FAIL* value the list in the action part is evaluated. If *FIRE ALL* is the mode of usage the interpreter fires all actions whose conditions succeed. If none fail the interpreter returns the value of the last evaluated expression.

The key parts of a rule are thus: I request .... T action, where I(f) introduces the conditional part and T(hen) the action. Rules must contain lower case letters and digits only, plus the relational and equality operators and the decimal point. Upper case letters are taken as being reserved characters and are inserted by the rule editor. The upper case letters used for separating expressions are, in addition to I and T, O for 'inclusive or', A for 'and', N for 'not'. These are logical operators considered as conjoining or disjoining expressions. A is used to separate statements in the action part. N is used as a prefix to the expression to mean that its value should be logically negated. Expressions involving A are evaluated while their combination is valid, i.e as soon as a condition has failed on the left hand side of an A then it is taken as false. If the left hand side of an O is true then all other expressions disjoint to that one by O are ignored (until T or A is reached). N is taken as 'and ... not' so will be ignored where an expression without would be ignored.

Message expressions are of the form: <request code number>, <class code number>, <message string>, introduced by the letters R, C and P respectively. The string part, P.... is optional. The parsing of conditions (and actions) recognises the expressions and each is sent in turn. A single expression is false if it returns *"0"*. If the expression is followed by the equality or relational operators its value is stored and compared with that of the expression to the right of the operator(s) which may be a numerical or string constant. Less than,'<', less than or equal to, '<=', equals, '=' , greater than or equal to,'>=', and greater than,'>' are recognised orderings. The meaning of the equality operator is taken as numerical unless both sides of the equation are strings, in which case the relation is true if the result of *strcmp()* taking the expressions as arguments is false.

These methods are properties of *task_manager* (archetypes for rule interpretation). If the interpreter method succeeds, a result of *"OK"* is returned. This accords with the recommended usage for o-c code.

Appendix A5 Messages found in rules

Rule Source		Request	Class	Effector	Argument
task_rules (2)					
(i)	1.	enquire	task_manager	task_stat_read	“colour”
	2.	quantify	batch_min_rules	call_gen_rules	“fire_all”
	3.	update	task_manager	task_stat_read	“batch”
(ii)	"1				
	4.	quantify	batch_min-rules	call_gen_rules	“fire_all”
	"3				
(iii)	"1				
	5.	interpret	batch_best_rule	call_rule-set	“fire_first”
	"3				
batch_min_rules (16)					
(i)	6.	enquire	colour_manager	enq_clr_const	“code”
	7.	set	batch-clr-itms	batch_set_itms	“batch”
	"7				“colour- ”
(ii)	8.	enquire	batch_clr_itms	enq_batch_details	“colour”
	9.	copy	body_items	batch_cpy_clr	“class”
(iii)	"8				
	10.	copy	body_colours	set_batch_bco	“class”
(iv)	11.	batch	batch-manager	batch_minpb	“current_item”
	12.	enquire	colour_items	enq_clr_const	“minpb”
batch_inc_rules (18)					
(i)	13.	interpret	batch_prty_rules	call_rule_set	“fire_all”
	14.	set	task_manager	call_cut	
(ii)	"8				“batch”
	"12				“minpb”
	15.	enquire	batch_manager	enq_batch_inst	“batch”
	16.	batch	batch_clr_itms	form_batch	“batch”
(iii)	17.	interpret	batch_ext_rules	call_rule_set	“current_item”
	18.	valid	batch_ext_rules	set_batch_cut	“current_item”
(iv)	"8			“current_item”	
	19.	valid	batch_clr_itms	batch_enq_one	“current_item”
	"7				“batch,one”
(v)	"8				“current_item,colour”
	"8				“current_item,achievement”
	20.	set	batch_manager	batch_class_set	“batch,one”

batch_best_rule (20)						
	(i)	"15				
		24.	set	batch_best_rule	call_cut	
	(ii)	"15				
		25.	best	batch-manager	pick_batch	
batch_prty_rules (21)						
	(i)	"8				"current_item,nil"
		"7				"current_item,nil,minus"
	(ii)	21.	enquire	batch_prty_rules	form_batch	"current_item,nil"
		22.	quantify	batch_clr_itms	quant_batch_itms	
						"current_item,nil,priority"
	(iii)	23.	batch	batch_clr_itms	enq_prty_one	"current_item"
		"8				"priority,one"
batch_ext_rules (22)						
	(i)	"8				"current_item,nil,one"
		26.	set	batch_ext_rules	call_cut	
	(ii)	"15				"current_item"
		"8				"batch,one"
		"8				"colour,current_item"

must be and how many are to be defined initially. For each of these items the indicated number of examples is now entered. If an instance file exists for the description the user is warned. The final item in any category is indicated by typing an asterisk.

Each example is accepted as a character (alphanumeric) string with some special codes. Upper case A(nd), I(f), N(ot), O(r) and T(hen) are assumed to indicate logical elements of a (n English-like) rule. The asterisk is taken to introduce a numeric field. The field must be terminated by a character defining its type; h/s for shorts (unsigned/signed), u/i for (unsigned/signed) integers, p/l for (unsigned/signed) longs, f for float and d for double. If the field is incorrectly formed the user is warned. The appropriate bytes are put in the instance file the numbers being held in numerical not ASCII form. The tally of characters left for the instance is reduced accordingly and the number is displayed (in ASCII form) terminated by a tilde.The float value 3.01 would appear on the screen as '*3.01~ '. The user can now have the names of object classes displayed with their associated orderings. The date of creation, name of author (up to 15 characters) and comments (up to 39 characters) are saved. Each object class is allowed a chronicle for updates etc. but at creation these are assumed to be valid for all classes, thus one record only is held. The record allows the creation date and date of last change to be recorded. These are held as 8 character fields in the form dd.mm.yy obtained from the *UNIX* utilities which read the real-time clock.

A6.2 Definition of the hierarchy and inheritance

The definition of the object hierarchy is now stated in order of the named classes in terms of the three essential attributes: sub-classes subsumed, messages responded to (methods) and super-classes (all identified by the ranking order of the subsumed class or action, *0* to *number of classes - 1*). No checking is done that the assignments are sensible (e.g. not self-referencing) but all codes must exist. Edges of the structure are identified by typing an asterisk. For the sub-class' attribute this means the item is a *leaf*, stored as -2, and for super-classes a *root, -1*. These attributes are held as lists, the sizes of which are declared before the contents are itemised. The lists for sub- and super- classes form a tree structure. For sub-classes the list represents all those objects which are immediate descendants. In the case of a superclass the head of the list is the object's parent,the next item the grandparent and so on unto the tail which is the root. It is not enforced that there be a single 'tree' since the messager can deal with multiple structures. This is done so that parts of the program that are outside the knowledge base, for example, may also use the class structure. Inheritance of properties is obtained by method references. If a referenced class does not, itself, have the required method the messager will supply the method/effector if it can find response to the message listed for an ancestor class. Although there is a redundancy of information, each object tracing its entire descent, the messager's tracing of inheritance is simplified.

Several actions are now taken to split up the description file into more manageable files and complete the definition of the object structure.

The files necessary for re-compilation of the system (in order to allow the messager to reference the effector functions) are edited. The header file, **DEFTAB.H**, is produced from a standard file, **DEFTAB.OLD.H**, by replacing the defined size of the message table (1) with the number of effector functions defined (ASCII form). The file specifying the table itself is set up from **METHODS.OLD.H**. This file, **METHODS.H**, has the type definitions for each effector (the names of the functions defined) followed by an initialised definition of the table itself. The table is an array of the size detailed in **DEFTAB.H** of references to functions so that each effector can be accessed eventually by a subscript rather than its name.

A6.3 Attachment of methods and the messager

The connection of message responses with effectors is done next. A file is constructed for this with the name of the definition file suffixed by **'.act'**. For each action that the user has attached to a class the intended effector is stated. Triples of the codes are recorded on the file. Where more than one triple is present for an action the action code is replaced by the number of entries. (The size, in bytes, required for the table and the number of entries are recorded at the start of the file, the triples following. The same indexing information is also held in the initial definition file.)

The class structure information is abstracted from the definition file and used to create a class file (definition file name with suffix **'.clss'**). The structure of the new file differs from the definition, the comments being re-ordered by class rather than similar fields all together *(vide supra)*. Some indexing information is copied to the start of the file.

The final re-organisation is to abstract the identifiers named by the user into a file for names (definition file + suffix **'.nam'**). The names of (all) classes, followed by methods and effectors are copied to this file.
Once the definition is done the program exits by calling the *make* utility to construct the required executable system. The defined effectors are assumed to be declared in the files comprising the specification for *make*, as is the messaging function.

The messager works by standard *C* conventions in that it requires an archetypal argument list. The arguments are, the code of the action desired, the code of the object class and a pointer to allow access to further information made available by the caller.
The first time the messager is called it forms an array showing the number of messaging triples that precede the first entry for each action code (in the action, object, effector table saved as above). This is done by looking up the action code field for the first entry of each code, *vide supra*. The messager does not initialise an

image of this action table in memory, this being done as part of the start-up procedure for the system. As mentioned above, the action table references all object/action attachments so the desired action can be located in one of two ways. Firstly, the action code will have a number of triples listed for it. If the object class named as target appears on the list the relevant effector function has been found. Should this fail the second attempt is to locate one of the object classes in the list on the list of superclasses of the target object. This gives the desired result because an effector located in this way belongs to the triple containing the object class nearest in the object hierarchy to the target class. The superclass list states the inheritance of the target in order of most recent ancestor first so looking in turn for a match on this list with the list of triples causes the most recently defined effector to potentiate the method. If an action cannot be effected the messager prints a warning and returns a standard string to the caller. If successful the messager calls the effector giving it the pointer as the argument.

Argument passing conventions are limited by *C* so that argument lists of caller and called should match. It is possible to use the *varargs* system to write functions that can accept varying argument lists but this is inherently non-portable. It is possible with the above convention to provide sufficient messaging ability so *varargs* and other extensions have been avoided.

A6.4 An object-centred paint scheduler

By using the *edit* utility the class structure, methods, and their effector function names, and class instances are declared, as above. The messaging table (of action requests - request and class - bound to their effectors) is fixed. The four files produced for each knowledge base definition are loaded, dumped and displayed by code from source file '**startup.c**'. All knowledge bases must include this code as well as that from '**msg.c**' which contains the messager. In addition, various header files ('**.h**') must be *included* for linking of the final system.

A tracing facility is contained in '**msg.c**' which is activated by a global variable *Trace*, assigned externally.

Quit interrupts (e.g. ^C) are trapped by a signal-handler which is installed at the first call on the messager. If tracing is installed, on such an interrupt the user is notified of the latest messaging call and can then a) select viewing of the knowledge base, i.e instances, via *startup* code; b) edit class instances, using code from the *editor*; c) quit tracing at the current level (i.e level of messaging); or d) abandon the program. When tracing is 'off' the user may quit or chose to continue.

The level of tracing is marked by the messager: each call creates a 'lower' level (i.e one lower) and each unwinding raises the level by one. If the current level of tracing is 'quit' from the messager breaks from executing the message and returns a value of "OK". Thus the call is obviated and execution should be allowed to continue. "OK" is a pre-defined non-fail status value for the object-centred system and the recommended value for satisfactory messages which do not have to return a special value. Return of "BADMESSAGE" indicates that the messager has not been able to process the message. User code can use this to trap invalid requests.

A message-passing test harness in '**harness.c**' can be used to check individual messaging calls. This file provides an entry-point for code (i.e. *main()*) and storage for necessary global variables (e.g. those used in **msg.c, startup.c**).

To make a test harness program the utility *maketest* accepts the names of the source files (less the '.c' suffix) containing the required code (i.e. for methods) and links them (compiled '.o' images) with the *harness*, *startup* and *messaging* code. The knowledge base to be tested is then given to the test program produced and the user can send messages to the classes desired and see their effect. The format is as in a proper program: the user specifies the code number or name of the request, that of the class and a string to be passed in the message. *Trace* is set for tests and so interrupts will allow examination of the knowledge base as in a real program. The exit value of the messager and each effector function is available also for inspection. This is intended for incremental testing of messaging relationships.

A6.4.1 Paint allocation

The object-centred version of the colour selection program contains a modified version of **PAINT** which loads the database and then runs the knowledge base start-up sequence. Bodies and colours are then processed from the usual files using a call on the object-centred sub-system to produce appropriate allocation decisions. The object-centred system may read the database files once only when initiated and maintains an image of the database by updating changes notified to it in successive messages sent to it. An exception to this is that the map sequence file is read to re-populate the knowledge base image of the assembly line each time a colour decision is required. This means that if an external change occurs, e.g. updating of order cover, the knowledge base will be re-initialised and will not need to read database files often.

A6.4.2 Messaging and system messages

The entry-point to the object-centred system is by a standard request to the messager i.e *(load task_manager)*. Messages are of the form *lval = msg(a,b,c)*, where *lval* is a string and *msg* the messaging function which returns it as the value of the message. *a* and *b*, respectively indicate the request and class of the message, *c* specifies the content of the message itself (as a string). The content of the string shows the method, in the above example, (whose effector is *task_allocator*) of *task_manager* the receiving class, whether the knowledge base is to be loaded *("ALL")*, dumped *("DUMP")* and/or a colour decision is required. This is done by *c* being composed of a list of items separated by full-stops, e.g. "0.0.0", i.e. three noughts. In this case the first nought indicates that the database image is to be loaded before a decision is made. The following two show that no change in that image is needed because the previous (i.e. non-existent) batch of nominally colour 0 was of size 0. In the case where a previous decision had been made these two items would show the colour that had been actually been used for the the previous batch and the size of that batch. (This may not accord with the previous allocation decision.) This is because the database image is updated when the allocation system is called (again) and not immediately after the decision. The knowledge base does not keep track of external events for itself but uses the information supplied on each invocation to maintain the integrity of its data. As mentioned above this is important if access to the database is to be restricted, i.e if the database-handling is done by an asynchronous task, e.g. a console control program.

As mentioned above the *task_manager/task_allocator* method allows invocation of the object-centred system. As a result of the *load* the sub-classes of the target class receive messages to load their database file image. The *db_manager* class is the controller for the database images and distributes messages to the map *(map_manager)*, colour *(colour_manager)*, and body *(body_manager)* classes to load their images of respective database files. The priority file image *(prty_manager)* is a property of the body class. As above the message to the object-centred part states the effect of processing its last decision. From this information which *db_manager* receives from *task_manager* it routes updating details to its subclasses. The *map_manager* receives an *update* message and reads the appropriate sequence of bodies from the sequence file to replace those processed in the previous batch, advancing the map. The effector used is *map_mngr_update()*. The current sequence file position is kept by this function. For the *colour_manager* the *update* message (=> *update_clr_obj()*) causes the specified colour item to decrement its count of required bodies and increment its 'achievement' (by the specified batch size). In a similar manner *body_manager update*s its data and sends the appropriate *update* requests to its classes (which include the priority file image - *prty_manager* and its subclasses).

In this way control filters down through the object hierarchy. In these actions failure can be detected in the return value of the message (usually *"FAIL"*). If this happens or the colour selection gives an invalid result the value of *FAILALLOC ("0.0.0")* is returned through the messager to the invoking code. A successful call would be of the form *"P.Q"* where *P* is the integer colour code and *Q* the suggested batch allocation.

A6.4.3 Rule-based processing

The above discussion outlines a scheme by which a series of requests with messaging causes a cascade of effects. The various tasks required for colour selection are done in a similar way.

Following database image updating the *task_manager/task_allocator* method *(load)* has a loop which exits when the colour task is done. Inside this loop various sets of rules are evaluated causing colour selection to take place. The control point for this loop is the *task agendum* instance of *task_manager*. While this is not in the *DONE* state the loop continues. The *enquire* method of the *task_manager* class (=> *task_stat_read()*) allows examination of this agendum. An *interpret* request is sent to the *task_rules* subclass and this causes its instances to be interpreted as rules.

These rules will succeed on the appropriate state of the *task agendum* instance. Their actions are, respectively, to cause colour selection, batch selection and best batch selection. The message to interpret the rules contains the mode of evaluation, in this case *"fire first rule with valid conditions"*. This interpreter (=> *call_rule_set*) examines each of the known instances of a class in turn. In each case further rules are evaluated. The action statements of the rules cause changes in the state of the knowledge base and so processing converges on a result for colour selection.

The partitioning of the task's execution into task rules would allow an arbitrary set of such rules to be used if needed, i.e if the selection were to be made more complex. The three rules represent criteria used in the original conventional program, but are encrypted in the algorithm. The intention of this expression was to declare these 'principles' explicitly. The contexts for relevance of these 'facts' are stated together with the facts themselves. In practice the ordering of the rules used is not necessary to their validity. Efficiency prompts their declaration in the order given. Were other rules adduced to represent a more complex set of tasks these original rules should still be valid were the nature of the overall task to remain the same. Their meaning is not impaired if the action side's requests become implemented in a different way. The subject of rule interpretation is discussed further *(vide infra)*.

A6.4.4 Control of information

Access to data in classes is usually limited to the methods of the class (or those of an ancestor). Thus requests made are often of the form 'what is the value of instance x' and the answer is returned. Thus a class does not have to allow other classes to read its data. This is true of updating or re-setting of values also. An example is *map_manager*: Its first instance is the current 'visible' part of the map file. One form of *enquire* request causes it to supply the code of a body type at a specified position in the map.

6.4.5 The conventional algorithm represented in the object-centred system

The conventional algorithm checks first for batches containing a priority body/colour combination. Batches must be of MINPB size at least. The first rule for selection is stated in the documentation as being to select at least MINPB-sized batches. The object-centred system follows this direction in its first task rule. What is not stated unequivocally in the documentation is that, having looked first for a priority batch, the work done to find one, in the several colours, is not wasted should the search fail and the selection process go on to look for a least achieved colour. It is not clear how this would be done. (Particularly as the size batching rules are different for these two directives.) The solution chosen to resolve this is to find a MINPB batch in as many colours as possible and then (task rule two) to find the largest batch looking also for priorities.

The rules of selection applying to crisis allocation are also included in the rule set referenced in task rule two. Task rule three succeeds when the task agendum is in the state, batch done. It references a rule set to pick the best batch. In these rule applications the batching details and batching state are held in instances of the batch class *(batch_manager)*. The final selection of batch depends on the values in these instances. Either a valid batch is found (which may be priority) or a single body is indicated as allocated (by the rules of crisis allocation of the Istel algorithm). Note that the colour and batch details are returned to the invoking code for the object-centred system as described; failure of the tasks results in the exit value *FAILALLOC*. As with the task rules the various rules examined are only ordered for convenience, their validity is not dependent on order of examination.

A6.5 Definition of object meta-knowledge

The purpose of defining meta-knowledge of the knowledge contained in an object hierarchy is to assist in acquiring instances of the classes already defined and other aspects of the object-centred structure. A program, **METACODE**, allows further information to be codified, pertaining to an object structure.

Just as **EDITEX** allows two modes of use, **METACODE** can be used to create a meta-description from a given object definition, when invoked with no argument, or, when a name is given, to edit an existing one. The output of the program is a set of files corresponding to the object definition, named for the object files with a suffix '**u_**'. The initialisation of meta-descriptions is, effectively, an amalgamation of the given object definition with another object definition, which describes special aspects for modelling meta-knowledge.

A6.5.1 Initialisation of a meta-description

The current version of the program uses a definition called '**meta**'. This has classes to represent constant items, relational symbols and grammatical forms. The grammatical forms and their methods state the syntax for rules (according to the standard methods in the source file '**rules.c**'. The constants and other symbols are stated with the internal representations they should have, e.g. *selecting_colour* has the internal value 0. In initialisation the classes, methods and effectors are extracted from the object definition and recorded in the meta-description. At the same time, the user can indicate what the instance variables mean by assigning identifiers to them and stating what size and type of item represent. A concordance of methods to classes is extracted so that the program can later detect possible messages that classes could receive. The arguments of effectors can also be described in this way. This is a form of *metalevel* knowledge, since it can be used to guide acquisition of new knowledge. The information from '**meta**' and these details are combined to form the initialised meta-description corresponding to the object definition.

A6.5.2 Acquisition of new knowledge

Having initialised a meta-description, the user can invoke **METACODE** with the meta-description as argument. This allows access to two forms of modification of the meta-description. Further additions to structural knowledge, e.g. new constants may be defined, or an acquisition system can be used. At this stage, the 'forms' of knowledge included in '**meta**' are for instances of classes, specifically rules (as the most complex abstraction used in the object system). The methods included allow an external form of the rule to be stated and an attempt to translate it into the internal representation required is made.

The user, currently, must state which classes are to be considered as rule sets, unless the meta-description has already defined this facet. It might be possible to categorise these automatically by assuming that instances containing only printable characters are rules, but this is not a particularly useful distinction. The system then decides the total number of rules by examining the object definition inherited in the meta-description. The user can state an external form for the rule. This is the form adduced for explanations using rule tracing. The form of rules is prescriptive and uniform and rules must have a 'predicate' part, which is a symbol like 'if', optionally preceded by a quantifier (a phrase declaring the class for which the rule applies over all its instances, consisting of some symbol like "for_all colour_manager"). This must be followed by a phrase constituting a conditional part; a 'dependent', such as 'then', and an action part. The conditional and action parts consist of one or more expressions which resolve to a message. The individual conditional expressions may be boolean or have a relational symbol followed by a message expression or constant. In the case of an action the expression is simply imperative. The internal form of a message must contain a request name and a class. The parsing methods then, using the object structure and messaging to record the components of phrases in classes relating to the grammatical parts, check the completeness and consistency of the rule. The message selectors are specified in the expressions by such forms as 'set code of red of colour-manager to value': i.e. the colour instance *red*'s variable code should be *set* to *value*.

If parts are missing or inappropriate, the parser informs the user, who can supply values for new symbols such as relational names. A valid rule can be saved in the meta-description and, if desired, the original object definition can have the new rule inserted in it.

A6.5.3 Other uses of meta-knowledge

Some of the knowledge, e.g. of effectors, is not used in this version of **METACODE**. It could be used to allow more flexible acquisition strategies. Also, the inclusion of other skeletal forms for meta-knowledge in 'meta' could generate other types of instances. This is discussed in Chapter 8.